

Generic Programming with Dependent Types

Thorsten Altenkirch, Conor McBride and Peter Morris

School of Computer Science and Information Technology
University of Nottingham

1 Introduction

In these lecture notes we give an overview of recent research on the relationship and interaction between two novel ideas in (functional) programming:

Generic programming Generic programming allows programmers to explain how a single algorithm can be instantiated for a variety of datatypes, by computation over each datatype's structure.

Dependent types Dependent types are types containing data which enable the programmer to express properties of data concisely, covering the whole spectrum from conventional uses of types to types-as-specifications and programs-as-proofs.

Our central thesis can be summarized by saying that dependent types provide a convenient basis for generic programming by using *universes*. A universe is basically a type $U : \star$ which contains names for types and a dependent type, or family, $El : U \rightarrow \star$ Which assigns to every name $a : U$ the type of its elements $El\ a : \star$. Universes have been already used by Type Theory to capture the predicative hierarchy of types, first introduced by Russell to prevent set-theoretic paradoxes:

$$\star_0 : \star_1 : \star_2 : \dots \star_i : \star_{i+1} : \dots$$

which can be represented as

$$\begin{aligned} U_i &: \star \\ El_i &: U_i \rightarrow \star \\ u_i &: U_{i+1} \end{aligned}$$

such that $El_{i+1}\ u_i$ is isomorphic to U_i .

Here we are interested in the application of universes to programming, which leads to consider a wider variety of smaller universes, less general but more usefully structured than the ones above.

Structure of the paper

We start our discourse with a quick introduction to dependently typed programming (section 2) using the language Epigram as a vehicle. Epigram is described

in more detail elsewhere, see [22] for a definition of the language with many applications, [10] for a short and more recent overview and [21] for an introductory tutorial. Epigram is not just a language but also an interactive program development system, which is, together with further documentation, available from the Epigram homepage [20].

As a warm up we start with a very small, yet useful universe, the universe of finite types (section 3). The names for types in this universe are particularly simple, they are just the natural numbers.

We soon move to bigger universes which include infinite types in section 4, where we introduce a general technique how to represent universes which contain fixpoints of types — this section is based on [23]. We also discuss the tradeoff between the size of a universe and the number of generic operations it supports.

While the universes above are defined syntactically, we also present a semantic approach based on *Container Types* (section 5), see [1–4]. However, here we will not study the categorical details of containers in detail but restrict ourselves to using them to represent datatypes and generic operations in Epigram.

As an example of a library of generic operations we consider the generic zipper [16] (section 6), which is a useful generic tool when implementing functional programs which use the notion of a position in a data structure. As observed by McBride [19], the zipper is closely related to the notion of the derivative of a datatype, which has many structural similarities to derivatives in calculus. This topic has been explored from a more categorical perspective in [5, 7]; the presentation here is again based on [23].

We close with our conclusions and possible directions for further work (section 7).

2 Programming with Dependent Types

Epigram is an experimental dependently typed functional language and an interactive program development system. It is based on previous experiences with systems based on Type Theory whose emphasis has been on the representations of proofs like LEGO [17]. The design of the interactive program and proof development environment is heavily influenced by the ALF system [18].

Epigram uses a two-dimensional syntax to represent the types of operators in a natural deduction style. This is particularly useful for presenting programs with dependent types — however, we start with some familiar constructions, e.g. we define the Booleans and the Peano-style natural numbers as follows:

$$\text{data } \text{Bool} : \star \quad \text{where } \text{true}, \text{false} : \text{Bool}$$

$$\text{data } \left(\frac{}{\text{Nat} : \star} \right) \quad \text{where } \left(\frac{}{\text{zero} : \text{Nat}} \right) ; \left(\frac{n : \text{Nat}}{\text{suc } n : \text{Nat}} \right)$$

We first give the *formation rule*, `Nat` is a type, which doesn't require any assumptions, then we introduce the *constructors* `zero` is a natural number and

`suc` n is a natural number, if n is one. These declarations corresponds to the Haskell datatypes:

```
data Bool = True | False
data Nat = Zero | Succ Nat
```

The formation rule may contain assumptions as well, e.g. in the case of lists:

$$\underline{\text{data}} \left(\frac{A : \star}{\text{List } A : \star} \right) \text{ where } \left(\frac{}{\text{nil} : \text{List } A} \right) ; \left(\frac{a : A ; as : \text{List } A}{\text{cons } a \text{ as} : \text{List } a} \right)$$

Like in the corresponding Haskell declaration,

```
data List a = Nil | Cons a (List a)
```

the type-valued arguments to the constructors are left implicit. However, as we will see, the subject of implicit arguments becomes more subtle when using dependent types. Hence, Epigram offers an explicit notation to indicate where implicit arguments should be expected. The definition above can be spelt out in full:

$$\underline{\text{data}} \left(\frac{A : \star}{\text{List } A : \star} \right) \text{ where } \left(\frac{A : \star}{\text{nil}_A : \text{List } A} \right) ; \left(\frac{A : \star ; a : A ; as : \text{List } A}{\text{cons}_A a as : \text{List } a} \right)$$

Here, the underscore signals an implicit argument. Under normal circumstances, Epigram's *elaborator* will be able to infer values for these arguments from the way these constructors are used, following a standard unification-based approach to type inference. Previously, we also omitted the declaration of $A : \star$ in the premise: again, this can be inferred by the elaborator as well, with the natural deduction rule acting like Hindley-Milner 'let', implicitly generalising local free variables.

We define functions using `let` and generate patterns interactively using the `case` tactic. E.g. we define boolean `and`:

$$\underline{\text{let}} \left(\frac{a, b : \text{Bool}}{\text{and } a b : \text{Bool}} \right) ; \text{and } a b \leftarrow \text{case } a \left\{ \begin{array}{l} \text{and true } b \Rightarrow b \\ \text{and false } b \Rightarrow \text{false} \end{array} \right\}$$

Epigram is a total language, hence all programs terminate.¹ To implement a recursive program, we have to indicate over which variables we want to recur and the recursive calls have to be structurally smaller in this argument. E.g. as simple examples consider addition and multiplication of Peano natural numbers:

$$\underline{\text{let}} \left(\frac{x, y : \text{Nat}}{\text{plus } x y : \text{Nat}} \right) ; \text{plus } x y \leftarrow \text{rec } x \left\{ \begin{array}{l} \text{plus } x y \leftarrow \text{case } x \left\{ \begin{array}{l} \text{plus zero } y \Rightarrow y \\ \text{plus (suc } x) y \Rightarrow \text{suc (plus } x y) \end{array} \right\} \end{array} \right\}$$

¹ The condition that the use of universes has to be stratified is present in the language definition but absent from the current implementation. As a consequence, the machine will accept a bogus non-terminating term based on Girard's paradox.

$$\underline{\text{let}} \left(\frac{m, n : \text{Nat}}{\text{times } m \ n : \text{Nat}} \right) ; \text{times } m \ n \Leftarrow \underline{\text{rec}} \ m \{ \\ \text{times } m \ n \Leftarrow \underline{\text{case}} \ m \{ \\ \text{times } \text{zero } n \Rightarrow \text{zero} \\ \text{times } (\text{suc } m) \ n \Rightarrow \text{plus } n \ (\text{times } m \ n) \} \}$$

We introduce a definition with a let-declaration and use the two-dimensional rule syntax to declare its type. In Epigram the types of all top-level declarations have to be given explicitly. The program can then be developed by:

\Leftarrow (**by**) refinement into subproblems using programming tactics like rec, case or view.
 \Rightarrow (**return**) by supplying an answer.

It should be noted that these operations produce a term in Epigram's underlying Type Theory which only uses standard elimination constants to perform recursion or case analysis. The recursion tactic is quite flexible it is straightforward to implement simple structurally recursive programs like the Fibonacci function or the Ackermann function.

So far we haven't used dependent types explicitly. A standard example for a dependent type is the type of vectors of a given length:

$$\underline{\text{data}} \left(\frac{n : \text{Nat} \quad X : \star}{\text{Vec } n \ X} \right) \\ \underline{\text{where}} \left(\frac{}{\text{vnil} : \text{Vec } \text{zero } X} \right) ; \left(\frac{a : A \quad as : \text{Vec } n \ X}{\text{vcons } a \ as : \text{Vec } (\text{suc } n) \ X} \right)$$

Here $\text{Vec } n \ X$ is the type of vectors of length n of items of type X .

We can now implement a safe version of the head function, whose type makes it clear that the function can only be applied to non-empty lists:

$$\underline{\text{let}} \left(\frac{ys : \text{Vec } (\text{suc } m) \ Y}{\text{vhead } ys : Y} \right) ; \text{vhead } ys \Leftarrow \underline{\text{case}} \ ys \{ \\ \text{vhead } (\text{vcons } y \ ys) \Rightarrow y \}$$

More generally, we can implement a function which safely accesses any element of a vector. To do this we first define the family of finite types, with the intention that $\text{Fin } n$ represents the finite set $\{0, 1, \dots, n - 1\}$, i.e. exactly the positions in a vector of length n .

$$\underline{\text{data}} \left(\frac{n : \text{Nat}}{\text{Fin } n : \star} \right) \underline{\text{where}} \left(\frac{}{\text{fz} : \text{Fin } (\text{suc } n)} \right) ; \left(\frac{i : \text{Fin } n}{\text{fs } i : \text{Fin } (\text{suc } n)} \right)$$

Here fz represents the 0 which is present in any non-empty finite set, and $\text{fs } i : \text{Fin } (\text{suc } n)$ represents $i + 1$, given that $i : \text{Fin } n$. Note that $\text{Fin } \text{zero}$ is intentionally left empty and we can write programs with empty patterns whenever we encounter a hypothetical element of $\text{Fin } \text{zero}$. The following table enumerates

the first 4 instances of **Fin**:

Fin 0	Fin 1	Fin 2	Fin 3	Fin 4	...
	fz ₀	fz ₁	fz ₂	fz ₃	...
		fs ₁ fz ₀	fs ₂ fz ₁	fs ₃ fz ₂	...
			fs ₂ (fs ₁ fz ₀)	fs ₃ (fs ₂ fz ₁)	...
				fs ₃ (fs ₂ (fs ₁ fz ₀))	...
					...

We implement the function **proj** which safely accesses the i th element of a vector by structural recursion over the vector. We analyze the index given as an element of **Fin** n and since both constructors of **Fin** n produce elements in **Fin** (**suc** m) the subsequent analysis of the vector has only to consider the **vcons** case. The patterns are automatically generated by the Epigram system.

$$\underline{\text{let}} \left(\frac{xs : \text{Vec } n \ X \quad i : \text{Fin } n}{\text{proj } xs \ i : X} \right) ; \text{proj } xs \ i \leftarrow \text{rec } xs \left\{ \begin{array}{l} \text{proj } xs \ i \leftarrow \text{case } i \left\{ \begin{array}{l} \text{proj } xs \ \text{fz} \leftarrow \text{case } xs \left\{ \begin{array}{l} \text{proj } (\text{vcons } x \ xs) \ \text{fz} \Rightarrow x \end{array} \right. \\ \text{proj } xs \ (\text{fs } i) \leftarrow \text{case } xs \left\{ \begin{array}{l} \text{proj } (\text{vcons } x \ xs) \ (\text{fs } i) \end{array} \right\} \end{array} \right. \end{array} \right. \end{array} \right.$$

Exercise 1. Implement the function **transpose** which turns an $m \times n$ matrix represent as an m vector of n vectors into an $n \times m$ matrix represented as a n vector of m vectors:

$$\underline{\text{let}} \left(\frac{xy : \text{Vec } n \ (\text{Vec } m \ X)}{\text{transpose } xy : \text{Vec } m \ (\text{Vec } n \ X)} \right)$$

Epigram provides very few predefined types: the empty type **Zero**, the unit type **One** and the equality type $a = b : \star$ for any a, b not necessarily of the same type. The only inhabitant of equality is **refl** : $a = a$. Epigram's function types $A \rightarrow B : \star$ for $A, B : \star$ are a special case of the dependent function type $\forall a : A \Rightarrow B$, where $B : \star$ under the assumption $a : A$, that the assumption isn't actually used. Lambda abstraction is written $\lambda x : A \Rightarrow b$. The domain information can be omitted in λ and \forall , if it can be inferred from the context. Several abstraction of the same kind can be combined using $;$, i.e. we write $\lambda x; y \Rightarrow c$ for $\lambda x \Rightarrow \lambda y \Rightarrow c$. The rule notation is just a convenient way to define functions, e.g. the type of **vhead** can be written explicitly as $\forall_m ; _Y \Rightarrow \text{Vec } (\text{suc } m) \ Y \rightarrow Y$.

There are a few additional standard type constructors which we will use: we define a type for disjoint union corresponding to **Either** in Haskell:

$$\underline{\text{data}} \left(\frac{A, B : \star}{\text{Plus } A \ B : \star} \right) \underline{\text{where}} \left(\frac{a : A}{\text{Inl } a : \text{Plus } A \ B} \right) ; \left(\frac{b : B}{\text{Inr } b : \text{Plus } A \ B} \right)$$

Epigram hasn't currently a predefined product type, hence we define it:

$$\underline{\text{data}} \left(\frac{A, B : \star}{\text{Times } A B} \right) \text{ where } \left(\frac{a : A \quad b : B}{\text{Pair } a b : \text{Times } A B} \right)$$

We also introduce the dependent product or Σ -type:

$$\underline{\text{data}} \left(\frac{A : \star \quad B : A \rightarrow \star}{\text{Sigma } A B : \star} \right) \text{ where } \left(\frac{a : A \quad b : B a}{\text{tup } a b : \text{Sigma } A B} \right)$$

Exercise 2. Define the first and second projection for Σ -types using pattern matching:

$$\underline{\text{let}} \left(\frac{p : \text{Sigma } A B}{\text{fst } p : A} \right)$$

$$\underline{\text{let}} \left(\frac{p : \text{Sigma } A B}{\text{snd } p : B (\text{fst } p)} \right)$$

3 The Universe of Finite Types

We have already implicitly introduced the first example of a universe, the universe of finite types. The names of finite types are the natural numbers which tell us how many elements the type has and the extension of such a type is given by the family `Fin` given in the previous section, which assigns to any $n : \text{Nat}$ a type `Fin n` with exactly n elements. We will now identify basic operations on types within this universe, namely coproducts ($0, +$), products ($1, \times$) and leave exponentials (\rightarrow) as an exercise. This reflects the well known fact that the category of finite types is bicartesian closed.

Coproducts

The coproduct of two finite types $m, n : \text{Nat}$ is simply their arithmetical sum `plus m n : Nat`, which we have defined previously. Coproducts come with injections and an eliminator which gives us case analysis. We will use Epigram's views to implement a view on coproducts in the finite universe. As a consequence we can use Epigram's pattern matching to analyze elements of `Fin (plus m n)` as if they were elements of an ordinary top-level coproduct (`Plus`).

We are going to parametrize the injections `finl` and `finr` explicitly with the type parameters $m, n : \text{Nat}$ leading to the following signature:

$$\underline{\text{let}} \left(\frac{m, n \quad i : \text{Fin } m}{\text{finl } m n i : \text{Fin } (\text{plus } m n)} \right)$$

$$\underline{\text{let}} \left(\frac{m, n \quad j : \text{Fin } n}{\text{finr } m n j : \text{Fin } (\text{plus } m n)} \right)$$

Intuitively, **finl** will map the elements of $\text{Fin } m$ to the first m elements of $\text{Fin } (\text{plus } m \ n)$ and **finr** will map the elements of $\text{Fin } n$ to the subsequent n elements of $\text{Fin } (\text{plus } m \ n)$. These ideas can be turned into structural recursive programs over m : in the case of **finl**

$$\begin{aligned} \text{finl } m \ n \ i &\Leftarrow \text{rec } m \{ \\ \text{finl } m \ n \ i &\Leftarrow \text{case } i \{ \\ \text{finl } (\text{suc } m) \ n \ \text{fz} &\Rightarrow \text{fz} \\ \text{finl } (\text{suc } m) \ n \ (\text{fs } i) &\Rightarrow \text{fs } (\text{finl } m \ n \ i) \} \} \end{aligned}$$

we analyze the element $i : \text{Fin } m$ mapping the constructors **fs**, **fz** in $\text{Fin } m$ to their counterparts in $\text{Fin } (\text{plus } m \ n)$. To implement **finr** we follow a different strategy:

$$\begin{aligned} \text{finr } m \ n \ j &\Leftarrow \text{rec } m \{ \\ \text{finr } m \ n \ j &\Leftarrow \text{case } m \{ \\ \text{finr } \text{zero } n \ j &\Rightarrow j \\ \text{finr } (\text{suc } m) \ n \ j &\Rightarrow \text{fs } (\text{finr } m \ n \ j) \} \} \end{aligned}$$

We analyze the type name $m : \text{Nat}$ to apply m successor operations **fs** to lift $\text{Fin } n$ into $\text{Fin } (\text{plus } m \ n)$. It is worthwhile to note that the above implementations of **finl** and **finr** only work for the given implementation of **plus** which recurs over the first argument. Had we chosen a different one, we would have to either have chosen a different implementation of **finl** and **finr** or would have to employ equational reasoning to justify our implementation. We tend to avoid the latter as much as possible by carefully choosing the way we implement our functions.

How can we compute with elements of $\text{Fin } (\text{plus } m \ n)$? One way to answer this question is to provide an eliminator in form of a case-function:

$$\text{let } \left(\frac{m, n \quad X \quad s : \text{Fin } (\text{plus } m \ n) \quad l : \text{Fin } m \rightarrow X \quad r : \text{Fin } n \rightarrow X}{\text{fcase_m_n_s_l_r} : X} \right)$$

However, Epigram offers a general mechanism which allows the user to extend the predefined pattern matching mechanism by providing a view, i.e. an alternative covering of a given type which is represented as a family:

$$\text{data } \left(\frac{m, n \quad i : \text{Fin } (\text{plus } m \ n)}{\text{FinPlus } m \ n \ i} \right) \text{ where} \\ \left(\frac{i : \text{Fin } m}{\text{inL } i : \text{FinPlus } m \ n \ (\text{finl } m \ n \ j)} \right) ; \left(\frac{j : \text{Fin } n}{\text{inR } j : \text{FinPlus } m \ n \ (\text{finr } m \ n \ j)} \right)$$

To use the `FinPlus` view for pattern matching we have to implement a function which witnesses that the covering is exhaustive:

$$\underline{\text{let}} \left(\frac{}{\mathbf{finPlus} \ m \ n \ i : \mathbf{FinPlus} \ m \ n \ i} \right)$$

```

finPlus m n i <- rec m {
finPlus m n i <- case m {
finPlus zero n i => inR i
finPlus (suc m) n i <- case i {
finPlus (suc m) n fz => inL fz
finPlus (suc m) n (fs i) <- view finPlus m n i {
finPlus (suc m) n (fs (finl m n i)) => inL (fs i)
finPlus (suc m) n (fs (finr m n j)) => inR j } } }

```

We can now use the `view` gadget to do pattern matching over `Fin (plus m n)`, e.g. to implement `fcase`:

```

fcase_m_n s l r <- view finPlus m n s {
fcase_m_n (finl m n i) l r => l i
fcase_m_n (finr m n j) l r => r j }

```

This may look slightly disturbing to the ordinary functional programmer, these pattern aren't linear and even worse `finl` and `finr` are defined functions — how can we pattern match over those. Indeed, Epigrams pattern are only *for show*, the operational behaviour is generated by the way we have refined the programming problem using the gadgets on the left of `<-`. The patterns are for readability and they also indicate which variables are bound.

Products

Given type names `m, n : Nat` their cartesian product is denoted by the arithmetic product `times m n`. Elements of `Fin (times m n)` can be constructed using pairing:

$$\underline{\text{let}} \left(\frac{m, n \ i : \mathbf{Fin} \ m \ j : \mathbf{Fin} \ n}{\mathbf{pair_m_n} \ i \ j : \mathbf{Fin} \ (\mathbf{times} \ m \ n)} \right)$$

The intuitive idea is to arrange the elements of `Fin (times m n)` as a rectangle and assign to `pair i j` the `j`th column in the `i`th row. This is realized by the following primitive recursive function which uses the previously defined constructors for coproducts, since our products are merely iterated coproducts:

```

pair_m_n i j <- rec i {
pair_m_n i j <- case i {
pair_(suc m)_n fz j => finl n (times m n) j
pair_(suc m)_n (fs i) j => finr n (times m n) (pair i j) } }

```

Indeed the **pair** $i j$ just computes $j + i * n$, however our implementation verifies that the result is less than $m * n$ simply by type checking.

As in the case for coproducts we extend pattern matching to cover our products by providing the appropriate view:

$$\begin{array}{l} \underline{\text{data}} \left(\frac{m, n \quad i : \text{Fin} (\text{times } m \ n)}{\text{FinPair } m \ n \ i : \star} \right) \\ \underline{\text{where}} \left(\frac{i : \text{Fin } m \quad j : \text{Fin } n}{\text{finSplit } i \ j : \text{FinPair } m \ n (\text{pair } i \ j)} \right) \end{array}$$

As before we show that this view is exhaustive:

$$\begin{array}{l} \underline{\text{let}} \left(\frac{}{\text{finPair } m \ n \ i : \text{FinPair } m \ n \ i} \right) \\ \text{finPair } m \ n \ i \Leftarrow \text{rec } m \{ \\ \quad \text{finPair } m \ n \ i \Leftarrow \text{case } m \{ \\ \quad \quad \text{finPair } \text{zero } n \ i \Leftarrow \text{case } i \\ \quad \quad \text{finPair } (\text{suc } m) \ n \ i \Leftarrow \text{view } \text{finPlus } n (\text{times } m \ n) \ i \{ \\ \quad \quad \quad \text{finPair } (\text{suc } m) \ n (\text{finl } n (\text{times } m \ n) \ i) \Rightarrow \text{finSplit } \text{fz } i \\ \quad \quad \quad \text{finPair } (\text{suc } m) \ n (\text{finr } n (\text{times } m \ n) \ j) \Leftarrow \text{view } \text{finPair } m \ n \ j \{ \\ \quad \quad \quad \quad \text{finPair } (\text{suc } m) \ n (\text{finr } n (\text{times } m \ n) (\text{pair } m \ n \ i \ j)) \\ \quad \quad \quad \quad \Rightarrow \text{finSplit } (\text{fs } i) \ j \} \} \} \\ \quad \quad \quad \Rightarrow \text{finSplit } (\text{fs } i) \ j \} \} \} \} \end{array}$$

Note that we are using the previously defined **FinPlus** view to analyze the iterated coproducts. We can use both derived pattern matching principles to show that products distribute over coproducts

$$\underline{\text{let}} \left(\frac{m, n, o \quad x : \text{Fin} (\text{times } m (\text{plus } n \ o))}{\text{dist } m \ n \ o \ x : \text{Fin} (\text{plus } (\text{times } m \ n) (\text{times } m \ o))} \right)$$

$$\begin{array}{l} \text{dist } m \ n \ o \ x \Leftarrow \text{view } \text{finPair } m (\text{plus } n \ o) \ x \{ \\ \quad \text{dist } m \ n \ o (\text{pair } i \ j) \Leftarrow \text{view } \text{finPlus } n \ o \ j \{ \\ \quad \quad \text{dist } m \ n \ o (\text{pair } i (\text{finl } n \ o \ j)) \Rightarrow \text{finl } (\text{times } m \ n) (\text{times } m \ o) (\text{pair } i \ j) \\ \quad \quad \text{dist } m \ n \ o (\text{pair } i (\text{finr } n \ o \ j)) \Rightarrow \text{finr } (\text{times } m \ n) (\text{times } m \ o) (\text{pair } i \ j) \\ \quad \} \} \} \end{array}$$

The categorically inclined may notice that this is not an automatic consequence of having coproducts and coproducts, but usually established as a consequence of having exponentials. We leave it as an exercise to define exponentials.

Exercise 3. Define exponentials (i.e. function types) by implementing a function to represent the name of a function type:

$$\underline{\text{let}} \left(\frac{m, n : \text{Nat}}{\text{exp } m \ n : \text{Nat}} \right)$$

a constructor corresponding to lambda abstraction:

$$\underline{\text{let}} \left(\frac{m, n : \text{Nat} \quad f : \text{Fin } m \rightarrow \text{Fin } n}{\mathbf{flam} \ m \ n \ f : \text{Fin}(\mathbf{exp} \ m \ n)} \right)$$

Unlike in the previous cases we cannot implement a pattern matching principle due to the lack of extensionality in Epigram's type system. However, we can define an application operator:

$$\underline{\text{let}} \left(\frac{m, n : \text{Nat} \quad f : \text{Fin}(\mathbf{exp} \ m \ n) \quad i : \text{Fin } m}{\mathbf{fapp} \ m \ n \ f \ i : \text{Fin } n} \right)$$

4 Universes for Generic Programming

The previously introduced universe of finite types is extensional, any two functions which are extensionally equal are given the same code. E.g. using the example from [11] we can see that the functions $\lambda f : \text{Bool} \Rightarrow \text{Bool} \Rightarrow f$ and $\lambda f : \text{Bool} \Rightarrow \text{Bool} ; x : \text{Bool} \Rightarrow f(f(f\ x))$ are extensionally equal by encoding them using the combinators defined in the previous section and observing that they compute the same element in [Fin 256](#)²

While extensionality is a desirable feature, it is not always as easy to achieve as in the case of finite types. Hence, when moving to larger universes which allow us to represent infinite datatypes we shall use a different approach. Instead of identifying our type constructors within a given type of names, we inductively define the type of type names and the family of inhabitants.

Finite types, revisited

To illustrate this let us revisit the universe of finite types, we can inductively define the type names generated from 0, +, 1, ×:

$$\underline{\text{data}} \left(\frac{}{\text{Ufin} : \star} \right) \underline{\text{where}} \left(\frac{}{\text{'0'} : \text{Ufin}} \right) ; \left(\frac{a, b : \text{Ufin}}{\text{'plus'} \ a \ b : \text{Ufin}} \right) \\ \left(\frac{}{\text{'1'} : \text{Ufin}} \right) ; \left(\frac{a, b : \text{Ufin}}{\text{'times'} \ a \ b : \text{Ufin}} \right)$$

We could also have included function types, however, they will require special attention later when we introduce inductive types.

² We don't recommend trying this with the current implementation of Epigram.

We define the family of elements `Elfin` inductively:

$$\begin{aligned} \underline{\text{data}} \left(\frac{a : \text{Ufin}}{\text{Elfin } a : \star} \right) \text{ where} \\ \left(\frac{b, a \quad x : \text{Elfin } a}{\text{inl } x : \text{Elfin } ('plus' \ a \ b)} \right) ; \left(\frac{a, b \quad y : \text{Elfin } b}{\text{inr } y : \text{Elfin } ('plus' \ a \ b)} \right) \\ \left(\frac{}{\text{void} : \text{Elfin } '1'} \right) ; \left(\frac{x : \text{Elfin } a \quad y : \text{Elfin } b}{\text{pair } x \ y : \text{Elfin } ('times' \ a \ b)} \right) \end{aligned}$$

Indeed, we have seen inductively defined families already when we introduced `Vec` and `Fin`. We can reimplement the `dist` function for this universes without having to resort to views, the built in pattern matching will do the job:

$$\underline{\text{let}} \left(\frac{x : \text{Elfin } ('times' \ a \ ('plus' \ b \ c))}{\text{dist } x : \text{Elfin } ('plus' \ ('times' \ a \ b) \ ('times' \ a \ c))} \right)$$

$$\begin{aligned} \text{dist } x \Leftarrow \text{case } x \{ \\ \text{dist } (\text{pair } x \ y) \Leftarrow \text{case } y \{ \\ \text{dist } (\text{pair } x \ (\text{inl } y)) \Rightarrow \text{inl } (\text{pair } x \ y) \\ \text{dist } (\text{pair } x \ (\text{inr } z)) \Rightarrow \text{inr } (\text{pair } x \ z) \} \} \end{aligned}$$

4.1 Enumerating finite types

So far we haven't defined any generic operations — an operation which is *typical* for finite types is the possibility to enumerate all elements of a given type. We shall use binary trees instead of lists to represent the results of an enumeration so that the path in the tree correspond to the choices we have to make to identify the element. Since our types may be empty we require a special constructor to represent an empty tree:

$$\begin{aligned} \underline{\text{data}} \left(\frac{A : \star}{\text{BT } A : \star} \right) \text{ where} \\ \left(\frac{a : A}{\text{leaf } a : \text{BT } A} \right) ; \left(\frac{l, r : \text{BT } A}{\text{node } l \ r : \text{BT } A} \right) ; \left(\frac{}{\varepsilon : \text{BT } A} \right) \end{aligned}$$

Our generic enumeration function has the following type:

$$\underline{\text{let}} \left(\frac{a : \text{Ufin}}{\text{enum } a : \text{BT } (\text{Elfin } a)} \right)$$

To implement **enum** it is helpful to observe that **BT** is a monad, with

$$\begin{aligned} & \underline{\text{let}} \left(\frac{a : A}{\text{returnBT } a : \text{BT } A} \right) ; \text{returnBT } a \Rightarrow \text{leaf } a \\ & \underline{\text{let}} \left(\frac{t : \text{BT } A \quad f : A \rightarrow \text{BT } B}{\text{bindBT } t f : \text{BT } B} \right) \\ & \text{bindBT } t f \Leftarrow \underline{\text{rec}} t \{ \\ & \quad \text{bindBT } t f \Leftarrow \underline{\text{case}} t \{ \\ & \quad \quad \text{bindBT } (\text{leaf } a) f \Rightarrow f a \\ & \quad \quad \text{bindBT } (\text{node } l r) f \Rightarrow \text{node } (\text{bindBT } l f) (\text{bindBT } r f) \\ & \quad \quad \text{bindBT } \varepsilon f \Rightarrow \varepsilon \} \} \end{aligned}$$

Consequently, **BT** is also functorial:

$$\begin{aligned} & \underline{\text{let}} \left(\frac{f : A \rightarrow B \quad t : \text{BT } A}{\text{mapBT } f t : \text{BT } B} \right) \\ & \text{mapBT } f t \Rightarrow \text{bindBT } t (\lambda x \Rightarrow \text{returnBT } (f x)) \end{aligned}$$

We are now ready to implement **enum** by structural recursion over the type name:

$$\begin{aligned} \text{enum } a & \Leftarrow \underline{\text{rec}} a \{ \\ \text{enum } a & \Leftarrow \underline{\text{case}} a \{ \\ \text{enum } \text{'0'} & \Rightarrow \varepsilon \\ \text{enum } \text{'plus'} a b & \Rightarrow \text{node } (\text{mapBT } \text{inl } (\text{enum } a)) (\text{mapBT } \text{inr } (\text{enum } b)) \\ \text{enum } \text{'1'} & \Rightarrow \text{leaf void} \\ \text{enum } \text{'times'} a b & \\ & \Rightarrow \text{bindBT } (\text{enum } a) (\lambda x \Rightarrow \text{mapBT } (\lambda y \Rightarrow \text{pair } x y) (\text{enum } b)) \} \} \end{aligned}$$

Exercise 4. Implement the function **enum** for the extensional universe of finite types.

$$\underline{\text{let}} \left(\frac{n : \text{Nat}}{\text{enum } a : \text{BT } (\text{Fin } n)} \right)$$

Exercise 5. Add function types to **Ufin** and extend **Elfin**. Can you extend **enum** to cover function types?

Context-free types

By context-free types³ we mean types which can be constructed by combining the polynomial operators from the previous section (0, +, ×, 1) with an operator μ to

³ We used the term *regular tree types* previously [23]; however this may be confusing since there are types which correspond to regular expressions.

construct inductive types, or in categorical terms initial algebras. Examples are natural numbers $\mathbf{Nat} = \mu X.1+X$, lists $\mathbf{List} A = \mu X.1+A \times X$ and rose trees $\mathbf{RT} = \mu X.\mathbf{List} X = \mu X.\mu Y.1 + X \times Y$. We use the term context-free types because the types have the same structure as context-free grammars, identifying parameters with terminal symbols, recursive variables with non-terminal symbols, choice with $+$ and sequence with \times .

The first technical issue we need to adress is how to represent variables. We use a deBruijn style presentation of names, that is we ignore names and avoid the issue of α -conversion. The names of context free types becomes a family indexed by the number of free variables:

$$\underline{\text{data}} \left(\frac{n : \mathbf{Nat}}{\mathbf{Ucf} \ n : \star} \right)$$

as constructors we retain the polynomial operators which leave the number of free variables unchanged:

$$\left(\frac{}{\mathbf{0} : \mathbf{Ucf} \ n} \right) ; \left(\frac{a, b : \mathbf{Ucf} \ n}{\mathbf{plus} \ a \ b : \mathbf{Ucf} \ n} \right) ; \left(\frac{}{\mathbf{1} : \mathbf{Ucf} \ n} \right) ; \left(\frac{a, b : \mathbf{Ucf} \ n}{\mathbf{times} \ a \ b : \mathbf{Ucf} \ n} \right)$$

To represent variables we introduce two constructors: \mathbf{vl} which represents the last variable in a non-empty context, and $\mathbf{wk} \ a$ which means that the type name a is weakened, i.e. the last variable is not used and \mathbf{vl} now refers to the variable before the last:

$$\left(\frac{}{\mathbf{vl} : \mathbf{Ucf} \ (\text{suc } n)} \right) ; \left(\frac{a : \mathbf{Ucf} \ n}{\mathbf{wk} \ a : \mathbf{Ucf} \ (\text{suc } n)} \right)$$

An alternative is to use the previously defined family of finite types directly, i.e. only to introduce one constructor:

$$\left(\frac{x : \mathbf{Fin} \ n}{\mathbf{var} \ x : \mathbf{Ucf} \ n} \right)$$

but it is slightly more convenient to use \mathbf{wk} and \mathbf{vl} because this otherwise we have to define operations on \mathbf{Fin} and \mathbf{Ucf} instead of just for \mathbf{Ucf} .

Dual to weakening is an operator representing *local definitions*, which allows us to replace the last variable by a given type name:

$$\left(\frac{f : \mathbf{Ucf} \ (\text{suc } n) \quad a : \mathbf{Ucf} \ n}{\mathbf{let} \ f \ a : \mathbf{Ucf} \ n} \right)$$

Alternatively, we could have defined substitution by recursion over the structure of type names. In the presence of a binding operator, here μ , this is not completely trivial. Later, we will see that another advantage of local definitions is that it allows us to define operations by structural recursion whose termination would have to be justified otherwise.

Finally, we introduce the constructor for inductive types, which binds the last variable and hence decreases the number of free variables by one:

$$\left(\frac{f : \mathbf{Ucf}(\mathbf{suc} \ n)}{\mathbf{'mu'} \ f : \mathbf{Ucf} \ n} \right)$$

Our examples (natural numbers, lists and rose trees) can be translated into type names in **Ucf**:

$$\underline{\text{let}} \left(\frac{}{\mathbf{nat} : \mathbf{Ucf} \ n} \right); \mathbf{nat} \Rightarrow \mathbf{'mu'}(\mathbf{'plus'} \ \mathbf{'1'} \ \mathbf{vl})$$

$$\underline{\text{let}} \left(\frac{}{\mathbf{list} : \mathbf{Ucf}(\mathbf{suc} \ n)} \right); \mathbf{list} \Rightarrow \mathbf{'mu'}(\mathbf{'plus'} \ \mathbf{'1'}(\mathbf{'times'}(\mathbf{wk} \ \mathbf{vl}) \ \mathbf{vl}))$$

$$\underline{\text{let}} \left(\frac{}{\mathbf{rt} : \mathbf{Ucf} \ n} \right); \mathbf{rt} \Rightarrow \mathbf{'mu'} \ \mathbf{list}$$

While **nat** and **rt** are closed types and hence inhabit **Ucf** *n* for any *n* : **Nat**, **list** is parametrized by the last type variable and hence inhabits **Ucf**(**suc** *n*). We exploit this in the definition of rose trees where we construct rose trees as the initial algebra of **list**. Alternatively we can instantiate **list** to any type name using **let**, e.g. **let list nat** : **Ucf** *n* represents the type of lists of natural numbers.

4.2 Elements of context-free types

How are we going to define the family of elements for **Ucf**? We have to take care of the free type variables. A first attempt would be to say that we have to interpret any type variable by a type, leading to the following signature⁴:

$$\left(\frac{a : \mathbf{Ucf} \ n \quad Xs : \mathbf{Vec} \ n \ \star}{\mathbf{Elcf} \ a \ Xs : \star} \right)$$

This approach works fine for the polynomial operators, which are interpreted as before, and the variables which correspond to projections; however, we run in difficulties for μ . A reasonable attempt is to say:

$$\left(\frac{x : \mathbf{Elcf} \ f \ (\mathbf{vcons}(\mathbf{Elcf}(\mathbf{'mu'} \ f) \ Xs) \ Xs)}{\text{in } x : \mathbf{Elcf}(\mathbf{'mu'} \ f) \ Xs} \right)$$

However, this definition is not accepted by Epigram's schema checker, since it is not able to verify that the nested occurrence of **Elcf** is only used in a strictly positive fashion. This check is necessary to keep Epigram's type system sound by avoiding potentially non-terminating programs.

⁴ We exploiting here $\star : \star$, however, this use can be stratified, i.e. if $Xs : \star_i$ then $\mathbf{Elcf} \ a \ Xs : \star_{i+1}$.

However, if we restrict ourselves to interpreting only closed types, we can overcome this problem. We define the **Elcf** wrt to a *closing substitution* or telescope, which interprets any free type variable by a type name with fewer free variables. Hence we define the family of telescopes:

$$\underline{\text{data}} \left(\frac{n : \text{Nat}}{\text{Tel } n : \star} \right) \text{ where } \left(\frac{}{\text{tnil} : \text{Tel zero}} \right) ; \left(\frac{a : \text{Ucf } n \quad as : \text{Tel } n}{\text{tcons } a \text{ } as : \text{Tel } (\text{suc } n)} \right)$$

We can now define an interpretation for an open type together with a fitting telescope:

$$\underline{\text{data}} \left(\frac{a : \text{Ucf } n \quad as : \text{Tel } n}{\text{Elcf } a \text{ } as : \star} \right)$$

the constructors for the polynomial operators stay the same, only indexed with a telescope which is passed through:

$$\left(\frac{b, a \quad x : \text{Elcf } a \text{ } as}{\text{inl } x : \text{Elcf } (\text{'plus'} \ a \ b) \ as} \right) ; \left(\frac{a, b \quad y : \text{Elcf } b \ as}{\text{inr } y : \text{Elcf } (\text{'plus'} \ a \ b) \ as} \right)$$

$$\left(\frac{}{\text{void} : \text{Elcf } \text{'1'} \ as} \right) ; \left(\frac{x : \text{Elcf } a \quad y : \text{Elcf } b \ as}{\text{pair } x \ y : \text{Elcf } (\text{'times'} \ a \ b) \ as} \right)$$

The interpretation of the last variable is simply the interpretation of the first type name in the telescope:

$$\left(\frac{x : \text{Elcf } a \ as}{\text{top } x : \text{Elcf } \text{vl} (\text{tcons } b \ as)} \right)$$

While the interpretation of a weakened type is given by popping off the first item of the telescope:

$$\left(\frac{x : \text{Elcf } a \ as}{\text{pop } x : \text{Elcf } (\text{wk } a) (\text{tcons } b \ as)} \right)$$

A local definition is explained by pushing the right hand side of the definition onto the telescope stack:

$$\left(\frac{x : \text{Elcf } f (\text{cons } a \ as)}{\text{push } x : \text{Elcf } (\text{'let'} \ f \ a) \ as} \right)$$

We can finally reap the fruits of our syntactic approach by providing an interpretation of **mu** which doesn't require a nested use of **Elcf**:

$$\left(\frac{x : \text{Elcf } f (\text{tcons } as \ (\text{'mu'} \ f))}{\text{in } x : \text{Elcf } (\text{'mu'} \ f) \ as} \right)$$

We can now derive constructors for our encoded types and provide a derived case analysis using views. We show this in the case of **nat** and leave the other examples as an exercise.

We derive the constructors representing 0 and successor:

$$\begin{aligned} & \underline{\text{let}} \left(\frac{}{\text{'zero'} : \text{Elcf nat } as} \right) ; \text{'zero'} \Rightarrow \text{in (inl void)} \\ & \underline{\text{let}} \left(\frac{n : \text{Elcf nat } as}{\text{'suc'} n : \text{Elcf nat } as} \right) ; \text{'suc'} n \Rightarrow \text{in (inr (top n))} \end{aligned}$$

Our view is that all elements of **Elcf nat** are constructed by one of the constructors, this is expressed by the family **NatView**:

$$\begin{aligned} & \underline{\text{data}} \left(\frac{n : \text{Elcf nat } as}{\text{NatView } n : \star} \right) \text{ where} \\ & \left(\frac{}{\text{isZ} : \text{NatView 'zero'}} \right) ; \left(\frac{n : \text{Elcf nat } as}{\text{isS } n : \text{NatView ('suc' n)}} \right) \end{aligned}$$

We show that **NatView** is exhaustive:

$$\begin{aligned} & \underline{\text{let}} \left(\frac{n : \text{Elcf nat } as}{\text{natview } n : \text{NatView } n} \right) \\ & \text{natview } n \Leftarrow \text{case } n \{ \\ & \quad \text{natview (in } x) \Leftarrow \text{case } x \{ \\ & \quad \quad \text{natview (in (inl } x)) \Leftarrow \text{case } x \{ \\ & \quad \quad \quad \text{natview (in (inl void))} \Rightarrow \text{isZ} \} \\ & \quad \quad \text{natview (in (inr } y)) \Leftarrow \text{case } y \{ \\ & \quad \quad \quad \text{natview (in (inr (top } n')))} \Rightarrow \text{isS } n' \} \} \} \end{aligned}$$

We can now use the derived pattern matching principle to implement functions over the encoded natural numbers:

$$\begin{aligned} & \underline{\text{let}} \left(\frac{m, n : \text{Elcf nat } as}{\text{'add'} m n : \text{Elcf nat } as} \right) \\ & \text{'add'} m n \Leftarrow \text{rec } m \{ \\ & \quad \text{'add'} m n \Leftarrow \text{view natview } m \{ \\ & \quad \quad \text{'add'} (\text{in (inl void)}) n \Rightarrow n \\ & \quad \quad \text{'add'} (\text{in (inr (top } m'))} n \Rightarrow \text{in (inr (top ('add' m' n)))} \} \} \end{aligned}$$

Unfortunately, epigram always normalizes terms which appear in patterns, expanding **'zero'** and **'suc'**, which makes the pattern not very readable.

Note that we don't need to derive a new recursion principle since structural recursion over the encoded natural numbers is the same as structural recursion over the natural numbers.

A natural question is whether we should have to distinguish between *encoded natural numbers* and *natural numbers* at all. The answer is clearly **no**, since they

are isomorphic anyway. To exploit this fact and avoid unnecessary duplication of definitions we need to built in a reflection mechanism into Epigram which allows us to access the names of the top level universe as data.

Exercise 6. Derive constructors and pattern matching principles (i.e. views) for the other two examples (lists and rose trees).

The *typical* generic operation on context-free types is generic equality. We can implement generic equality by structural recursion over the elements in `Elcf`:

```

let ( (x, y : Elcf a as)
      geq x y : Bool )

geq x y ← rec x {
  geq x y ← case x {
    geq (inl a) y ← case y {
      geq (inl a) (inl b) ⇒ geq x y
      geq (inl a) (inr b) ⇒ false }
    geq (inr a) y ← case y {
      geq (inr a) (inl b) ⇒ false
      geq (inr a) (inr b) ⇒ geq x y }
    geq void y ← case y {
      geq void void ⇒ true }
    geq (pair xa xb) y ← case y {
      geq (pair xa xb) (pair ya yb) ⇒ and (geq xa ya) (geq xb yb) }
    geq (top x) y ← case y {
      geq (top x) (top y) ⇒ geq x y }
    geq (pop x) y ← case y {
      geq (pop x) (pop y) ⇒ geq x y }
    geq (push x) y ← case y {
      geq (push x) (push y) ⇒ geq x y }
    geq (in x) y ← case y {
      geq (in x) (in y) ⇒ geq x y } } }

```

The algorithm is completely data-driven — indeed we never inspect the type. However, using the type information, the choice of the first argument limits the possible cases for the 2nd. This is what dependently typed pattern matching buys us, that it records the consequences of choices we have already made.

Unlike other approaches to generic equality we don't have to assume that the equality of type parameters is decidable. This is due to the fact that we only derive generic operations for closed types here.

Exercise 7. Instead of just returning a boolean we can actually show that we can decide equality of elements of `Elcf`. We say that a type is **decided**, if it can be established whether it is empty or inhabited. This is reflected by the following definition:

$$\text{let } \left(\frac{A : \star}{\text{Not } A : \star} \right) ; \text{Not } A \Rightarrow A \rightarrow \text{Zero}$$

$$\underline{\text{data}} \left(\frac{A : \star}{\text{Dec } A : \star} \right) \underline{\text{where}} \left(\frac{a : A}{\text{yes } a : \text{Dec } A} \right) ; \left(\frac{f : \text{Not } A}{\text{no } f : \text{Dec } A} \right)$$

Here **zero** is Epigram's built in empty type, establishing a function of type **Not A**, i.e. $A \rightarrow \text{Zero}$ establishes that A is uninhabited.

To show that equality for context-free types is decidable we have to implement:

$$\underline{\text{let}} \left(\frac{x, y : \text{Elcf } a \text{ as}}{\text{geqdec } x \ y : \text{Dec } (x = y)} \right)$$

geqdec is a non-trivial refinement of **geq** using Epigram's type system to show that the implementation of the program delivers what its name promises.

4.3 Strictly positive types

Context-free types capture most of the types which are useful in daily functional programming. However, in some situations we want to use trees which are infinitely branching. E.g. we may want to define a system of ordinal notations, which extends natural numbers by the possibility to form the limit, i.e. the least upper bound, of an infinite sequence of ordinals.

$$\underline{\text{data}} \left(\frac{}{\text{Ord} : \star} \right) \underline{\text{where}} \left(\frac{}{\text{oz} : \text{Ord}} \right) ; \left(\frac{a : \text{Ord}}{\text{os } a : \text{Ord}} \right) ; \left(\frac{f : \text{Nat} \rightarrow \text{Ord}}{\text{olim } f : \text{Ord}} \right)$$

We can embed the natural numbers into the ordinals:

$$\underline{\text{let}} \left(\frac{n : \text{Nat}}{\text{o2n } n : \star} \right) ; \text{n2o } n \Leftarrow \underline{\text{rec}} \ n \{ \\ \text{n2o } n \Leftarrow \underline{\text{case}} \ n \{ \\ \text{n2o } \text{zero} \Rightarrow \text{oz} \\ \text{n2o } (\text{suc } n) \Rightarrow \text{os } (\text{n2o } n) \} \}$$

and using this embedding we define the first infinite ordinal (ω) as the limit of the sequence of all natural numbers:

$$\underline{\text{let}} \left(\frac{}{\text{omega} : \text{Ord}} \right) ; \text{omega} \Rightarrow \text{olim } \text{n2o}$$

We can also do arithmetic on ordinals, using structural recursion we define addition⁵ of ordinals:

$$\begin{aligned} & \underline{\text{let}} \left(\frac{a, b : \text{Ord}}{\text{oplus } a \ b : \text{Ord}} \right) \\ & \text{oplus } a \ b \Leftarrow \underline{\text{rec}} \ b \{ \\ & \quad \text{oplus } a \ b \Leftarrow \underline{\text{case}} \ b \{ \\ & \quad \quad \text{oplus } a \ \text{oz} \Rightarrow a \\ & \quad \quad \text{oplus } a \ (\text{os } b) \Rightarrow \text{os} (\text{oplus } a \ b) \\ & \quad \quad \text{oplus } a \ (\text{olim } f) \Rightarrow \text{olim} (\lambda n \Rightarrow \text{oplus } a \ (f \ n)) \} \} \end{aligned}$$

Categorically, **Ord** is an initial algebra $\mu X.1 + X + \text{Nat} \rightarrow X$, it is an instance of a *strictly positive type*. Strictly positive types may use function types, like in $\text{Nat} \rightarrow X$ but we do not allow type variables to appear on the left-hand side of the arrow. I.e. $\mu X.X \rightarrow \text{Bool}$ is not strictly positive because X appears negatively, but neither is $\mu X.(X \rightarrow \text{Bool}) \rightarrow \text{Bool}$ because X appears positively but not strictly positive.

We introduce the universe of strictly positive types by amending the universe of context-free types. That is we define

$$\begin{aligned} & \underline{\text{data}} \left(\frac{n : \text{Nat}}{\text{Usp } n : \star} \right) \\ & \underline{\text{data}} \left(\frac{a : \text{Usp } n \quad as : \text{Tel } n}{\text{Elspl } a \ as : \star} \right) \end{aligned}$$

with all the same constructors as **Ucf** and **Elcf** and additionally a constructor for constant exponentiation:

$$\left(\frac{A : \star \quad b : \text{Usp } n}{\text{'arr'} \ A \ b : \text{Usp } n} \right)$$

and a corresponding constructor for **Elspl**:

$$\left(\frac{f : A \rightarrow \text{Elspl } b \ as}{\text{fun } f : \text{Elspl} (\text{'arr'} \ A \ b) \ as} \right)$$

It is now easy to represent ordinals in this universe:

$$\underline{\text{let}} \left(\frac{}{\text{ord} : \text{Usp } n} \right); \text{ord} \Rightarrow \text{'mu'} (\text{'plus'} \ \text{'1'} (\text{'plus'} \ \text{vl} (\text{'arr'} \ \text{Nat} \ \text{vl})))$$

⁵ To reflect the standard definition of ordinal addition we **have** to recur on the 2nd argument. Ordinal addition is not commutative, $\omega + 1$ denotes the successor of ω , while $1 + \omega$ is order-isomorphic to ω .

We have been cheating a bit, because the constructor ‘**arr**’ refers to a type. Thus $\mathbf{Usp} \ n : \star_{i+1}$ if $A : \star_i$ in ‘**arr**’ $A \ b$. An alternative would be to insist that the codomain of ‘**arr**’ is a closed strictly positive type, but this causes problems when introducing **fun** because of a negative occurrence of **Elsp**.

Exercise 8. Derive the constructors for **ord** : $\mathbf{Usp} \ n$ and a view which allows pattern matching over **ord**. Use this to define ordinal addition for the encoded ordinals.

4.4 Generic map

We don’t know any useful generic operation which apply to closed strictly positive types, but there is an important one for open ones: generic map. While we have given only an interpretation for closed types we are able to express generic map by introducing maps between telescopes.

We introduce a family representing maps between telescopes, which correspond to a sequence of maps between the components of the telescopes:

$$\underline{\text{data}} \left(\frac{as, bs : \mathbf{Tel} \ n}{\mathbf{Map} \ as \ bs : \star} \right)$$

and generic map simply lifts maps on telescope to a function between the element of a type instantiated with the telescopes:

$$\underline{\text{let}} \left(\frac{fs : \mathbf{Map} \ as \ bs \quad x : \mathbf{Elsp} \ a \ as}{\mathbf{gmap} \ fs \ x : \mathbf{Elsp} \ a \ bs} \right)$$

What are the constructors for **Map**? There are two obvious ones, which correspond to the idea that $\mathbf{Map} \ as \ bs$ is simply a sequences of maps between the components of as and bs :

$$\left(\frac{}{\mathbf{mnil} : \mathbf{Map} \ \mathbf{tnil} \ \mathbf{tnil}} \right) ; \left(\frac{f : \mathbf{Elsp} \ a \ as \rightarrow \mathbf{Elsp} \ b \ bs \quad fs : \mathbf{Map} \ as \ bs}{\mathbf{mcons} \ f \ fs : \mathbf{Map} \ (\mathbf{tcons} \ a \ as) \ (\mathbf{tcons} \ a \ as)} \right)$$

However, it is useful to introduce a 3rd constructor, which extends a given sequence of maps by the identity function:

$$\left(\frac{fs : \mathbf{Map} \ as \ bs}{\mathbf{mext} \ fs : \mathbf{Map} \ (\mathbf{tcons} \ a \ as) \ (\mathbf{tcons} \ a \ bs)} \right)$$

Note that **mext** fs isn’t just **mcons** $(\lambda x \Rightarrow X) \ fs$ as the type contexts change. It would be possible to define **mext** mutually with **gmap** but it is much easier to introduce an additional constructor which also keeps the program structural recursive.

The definition of **gmap** is now rather straightforward by structural recursion on the argument:

```

gmap fs x  $\Leftarrow$  rec x {
  gmap fs x  $\Leftarrow$  case x {
    gmap fs (inl x)  $\Rightarrow$  inl (gmap fs x)
    gmap fs (inr y)  $\Rightarrow$  inr (gmap fs y)
    gmap fs void  $\Rightarrow$  void
    gmap fs (pair x y)  $\Rightarrow$  pair (gmap fs x) (gmap fs y)
    gmap fs (fun f)  $\Rightarrow$  fun ( $\lambda x \Rightarrow$  gmap fs (f x))
    gmap fs (top x)  $\Leftarrow$  case fs {
      gmap (mcons f fs) (top x)  $\Rightarrow$  top (f x)
      gmap (mext fs) (top x)  $\Rightarrow$  top (gmap fs x) }
    gmap fs (pop x)  $\Leftarrow$  case fs {
      gmap (mcons f fs) (pop x)  $\Rightarrow$  pop (gmap fs x)
      gmap (mext fs) (pop x)  $\Rightarrow$  pop (gmap fs x) }
    gmap (push x)  $\Rightarrow$  push (gmap (mext fs) x)
    gmap (in x)  $\Rightarrow$  in (gmap (mext fs) x) } }

```

As before in the case of **geq** the program is data-driven, i.e. we never have to inspect the type. However, the type-discipline helps us to find the right definition, which in many cases is the only possible one.

Exercise 9. Instantiate **gmap** for **list**:

$$\underline{\text{let}} \left(\frac{}{\mathbf{list} : \mathbf{Usp}(\mathbf{suc} \ n)} \right) ; \mathbf{list} \Rightarrow \text{'mu' ('plus' '1' ('times' (wk vl) vl))}$$

to obtain

$$\underline{\text{let}} \left(\frac{f : \mathbf{Elsp} \ a \ as \rightarrow \mathbf{Elsp} \ b \ as \quad xs : \mathbf{Elsp} \ \mathbf{list} \ (\mathbf{tcons} \ a \ as)}{\mathbf{map} \ f \ xs : \mathbf{Elsp} \ \mathbf{list} \ (\mathbf{tcons} \ b \ as)} \right)$$

4.5 Relating universes

In the previous section we have incrementally defined three universes, each one extending the previous one together with a *typical* generic operation:

	universe of	inhabited by	generic operation
Ufin	finite types	Booleans (bool)	Enumeration (enum)
Ucf	context-free types	Rose trees (rt)	Equality (geq)
Usp	strictly positive types	Ordinals (ord)	Map (gmap)

We could have factored out the common parts of the definitions and established that every universe can be embedded into the next one but for paedagogical reasons we chose the incremental style of presentation. We note that the generic operations are typical for a given universe because they do not extend

to the next level, i.e. enumeration doesn't work for context-free types because they contain types with an infinite number of elements; equality doesn't work for strictly positive types because equality here is in general undecidable (e.g. for ordinals).

Are there any important universes we have left out? Between **Ufin** and **Ucf** we can find the universe of regular types, i.e. types which are represented as regular expressions, where the datatype of lists plays the role of Kleene's star. Another possibility is to also allow coinductive context-free types like streams by including codes for terminal coalgebras, e.g. **Stream** $X = \nu X. A \times X$. However, this doesn't fit very well with our way to define **El** inductively.

What about the universe of positive types extending the strictly positive types? It is unclear how to understand a type like $\mu X. (X \rightarrow \mathbf{Bool}) \rightarrow \mathbf{Bool}$ intuitively and there seem to be only very limited applications of positive inductive types. However, for **gmap** it is sensible to allow parameters in non-strict positive positions without closing under μ .

4.6 Universes and Representation Types

There is a very strong connection between the notion of universe in Type Theory, and the more recent notion of *representation type* which has emerged from work on type analysis [12] to become a popular basis for generic programming in Haskell [15, 14, 26]. The two notions are both standard ways to give a data representation to a collection of things, in this case, types:

- Martin-Löf's universes (U, El) collect types as the *image of a function* $El : U \rightarrow \star$. Elements of U may thus be treated as proxies for the types to which they map.
- Representation types characterise a collection of types as a *predicate*, $Rep : \star \rightarrow \star$. An element of $Rep T$ is a *proof* that T is in the collection, and it is also a piece of data from which one may compute.

The former approach is not possible in Haskell, because $U \rightarrow \star$ is not expressible when U is a type rather than a 'kind'. However, the latter has become possible, thanks to the recent extension of (ghc) Haskell with a type-indexed variant of inductive families [13], the so-called 'Generalized Algebraic Data Types' [25]. For example, one might define a universe of regular expression types as follows

```
data Rep a where
  Char    :: Rep Char
  Unit    :: Rep ()
  Pair    :: Rep a -> Rep b -> Rep (a,b)
  Either  :: Rep a -> Rep b -> Rep (Either a b)
  List    :: Rep a -> Rep [a]
```

and then write a function generic with respect to this universe by pattern-matching on **Rep**, always making sure to keep the the type representative to the left of the data which indirectly depends on it:

```

string :: Rep a -> a -> String
string Char      c      = [c]
string Unit      ()     = ""
string (Pair a b) (x, y) = string a x ++ string b y
string (Either a b) (Left x) = string x
string (Either a b) (Right y) = string y
string (List a)   xs     = xs >>= string a

```

Of course, in Epigram, these two kinds of collection are readily interchangeable. Given U and El , we may readily construct the predicate for ‘being in the image of El ’:

$$\mathbf{Rep} X \Rightarrow \mathbf{Sigma} U (\lambda u \Rightarrow El u = X)$$

In the other direction, given some Rep , we may ‘name’ a type as the dependent pair of the type itself and its representation: we interpret such a pair by projecting out the type!

$$\begin{aligned} \mathbf{U} &\Rightarrow \mathbf{Sigma} \star (\lambda X \Rightarrow Rep X) \\ \mathbf{El} &\Rightarrow \mathbf{fst} \end{aligned}$$

One can simulate this to some extent in Haskell by means of *existential* types

```
data U = forall a. U (Rep a)
```

and thus provide a means to compute one type from another—some sort of auxiliary data structure, perhaps—by writing a function `aux :: U -> U`. Unfortunately, `U` is not a `Sigma` type but a System F ‘weak’ existential and it prevents direct access to the type it packages. This capacity for secrecy is rather useful in other circumstances, but it is a pain here. The nearest we can get to El is a rank-2 accessor which grants *temporary* access to the witness to facilitate a computation whose type does not depend on it.

```
for :: U -> (forall a. Rep a -> b) -> b
```

However, there is no way to express operations whose types explicitly invoke `aux`. That’s not the fault of representation types as opposed to universes (although the latter are a little neater for such tasks), it’s just the gag we are forced to wear when we program in Haskell.

5 Containers

In section 3 we started with a semantic definition of the universe of finite type, while in the previous section we introduced universes syntactically, i.e. using inductive definitions. In the present section we will exploit our work on container types to give a semantic interpretation of the universe of context-free types which also works for strictly positive types. It is good to have both views of the universes available, we have seen that the inductive approach is very practical

to define generic and non-generic operations on data. However, the semantic approach we introduce here often provides an alternative approach to defining generic functions semantically. We will demonstrate this in more detail in the next section using the example of derivatives of datatypes. Another advantage of the semantic view is that it allows us to interpret open datatypes directly as operations on types, e.g. we can apply **list** to types which don't have a name in our universe.

5.1 Unary containers

Before embarking on the more general concept of n -ary containers, which as we will see can model exactly the universe of strictly positive types, we have look at unary containers, which model operators on types, i.e. inhabitants of $\star \Rightarrow \star$. An unary container is given by a type of shapes $S : \star$ and a family of positions $P : S \rightarrow \star$. I.e. we define:

$$\underline{\text{data}} \left(\frac{}{\text{UCont} : \star} \right) \text{ where } \left(\frac{S : \star \quad P : S \rightarrow \star}{\text{ucont } S \mid vP : \text{UCont}} \right)$$

The extension of container is given by a choice of shape and an assignment of payload (i.e. elements of the type parameter) to positions:

$$\underline{\text{data}} \left(\frac{C : \text{UCont} \quad X : \star}{\text{UExt } C \ X : \star} \right) \text{ where } \left(\frac{s : S \quad f : P \ s \rightarrow X}{\text{uext } s \ f : \text{UExt } (\text{ucont } S \ P) \ X} \right)$$

An example of an unary container is the representation of **List**:

$$\underline{\text{let}} \left(\frac{}{\text{cList} : \text{UCont}} \right) ; \text{cList} \Rightarrow \text{ucont Nat Fin}$$

The shape of a list is its length, i.e. a natural number, and a list with shape $n : \text{Nat}$ has **Fin** n positions.

Exercise 10. Implement **nil** and **cons** for **cList**:

$$\underline{\text{let}} \left(\frac{}{\text{cnil} : \text{UExt cList } X} \right)$$

$$\underline{\text{let}} \left(\frac{x : X \quad xs : \text{UExt cList } X}{\text{ccons } x \ xs : \text{UExt cList } X} \right)$$

Each container gives rise to a functor. We can implement **map** for unary containers by applying the function to be mapped directly to the payload:

$$\underline{\text{let}} \left(\frac{C : \text{UCont} \quad f : X \rightarrow Y \quad x : \text{UExt } C \ X}{\text{ucmap } C \ f \ x : \text{UExt } C \ Y} \right)$$

$$\text{ucmap } C \ f \ x \Leftarrow \text{case } C \{$$

$$\text{ucmap } (\text{ucont } S \ P) \ f \ x \Leftarrow \text{case } x \{$$

$$\text{ucmap } (\text{ucont } S \ P) \ f \ (\text{uext } s \ g) \Rightarrow \text{uext } s \ (\lambda x \Rightarrow f (g \ x)) \} \}$$

A morphism between functors is a natural transformation, i.e. *reverse* is a natural transformation from list to list. We can explicitly represent morphisms between containers, given unary containers $\mathbf{ucont} S P$ and $\mathbf{ucont} T Q$ a morphism is a function on shapes $f : S \Rightarrow T$ and a family of functions on positions, which assigns to every position in the target a position in the source, i.e. $u : \forall s : S \Rightarrow Q (f s) \Rightarrow P s$. The contravariance of the function on positions may be surprising, however, it can be intuitively understood by the fact that we can always say where a piece of payload comes from but not where it goes to, since it may be copied or disappear. Hence we define:

$$\underline{\text{data}} \left(\frac{C, D : \mathbf{UCont}}{\mathbf{UMor} C D : \star} \right) \text{ where } \left(\frac{P : S \rightarrow \star \quad Q \rightarrow \star}{f : S \rightarrow T \quad u : \forall s \Rightarrow Q (f s) \rightarrow P s} \right) \\ \underline{\text{umor}} f u : \mathbf{UMor} (\mathbf{ucont} S P) (\mathbf{ucont} T Q)$$

To every formal morphism between containers we assign a family of maps:

$$\underline{\text{let}} \left(\frac{fu : \mathbf{UMor} C D \quad x : \mathbf{UExt} C X}{\mathbf{UMapp} fu x : \mathbf{UExt} D X} \right) \\ \mathbf{UMapp} fu x \Leftarrow \text{case } fu \{ \\ \mathbf{UMapp} (\underline{\text{umor}} f u) x \Leftarrow \text{case } x \{ \\ \mathbf{UMapp} (\underline{\text{umor}} f u) (\underline{\text{uext}} s g) \Rightarrow \underline{\text{uext}} (f s) (\lambda y \Rightarrow g (u s y)) \} \}$$

It is not hard to show that these families of maps are always natural transformations in the categorical sense, wrt. to the interpretation of unary containers as functors given above. Indeed, it turns out that all natural transformations between functors arising from containers can be given as container morphisms, see theorem 3.4. in [2].

Exercise 11. Give representations of *head* and *reverse* as morphisms between unary containers, i.e.

$$\underline{\text{let}} \left(\frac{}{\mathbf{cHead} : \mathbf{UMor} \mathbf{cList} \mathbf{cI}} \right) \\ \underline{\text{let}} \left(\frac{}{\mathbf{cRev} : \mathbf{UMor} \mathbf{cList} \mathbf{cList}} \right)$$

where \mathbf{cI} is the unary container representing the identity functor:

$$\underline{\text{let}} \left(\frac{}{\mathbf{cI} : \mathbf{UCont}} \right) ; \mathbf{cI} \Rightarrow \mathbf{ucont} \mathbf{One} (\lambda x \Rightarrow \mathbf{One})$$

Exercise 12. While the interpretation of morphisms is full, i.e. every natural transformation comes from a container morphism, the same is not true for containers as representations of functors. Can you find a functor which is not representable as a unary container?

5.2 n -ary containers

To reflect our previous universe definitions we move straight-away to n -ary containers:

$$\underline{\text{data}} \left(\frac{n : \text{Nat}}{\text{Cont } n : \star} \right) \underline{\text{where}} \left(\frac{S : \star \quad P : \text{Fin } n \rightarrow S \rightarrow \star}{\text{cont } S P : \text{Cont } n} \right)$$

It's extension is given by an operator on a sequence of types, generalizing the sketch above to the n -ary case:

$$\underline{\text{let}} \left(\frac{C : \text{Cont } n \quad Xs : \text{Fin } n \rightarrow \star}{\text{Ext } C Xs : \star} \right) \underline{\text{where}} \left(\frac{P : \text{Fin } n \rightarrow S \rightarrow \star \quad Xs : \text{Fin } n \rightarrow \star}{S : S \quad f : \forall i : \text{Fin } n \Rightarrow P i s \rightarrow Xs i} \right) \\ \frac{\text{ext } s f : \text{Ext } (\text{cont } S P) Xs}{\text{ext } s f : \text{Ext } (\text{cont } S P) Xs}$$

Exercise 13. Show that n -ary containers give rise to n -ary functors, i.e. implement:

$$\underline{\text{let}} \left(\frac{C : \text{Cont } n \quad Xs, Ys : \text{Fin } n \rightarrow \star}{fs : \forall i : \text{Fin } n \Rightarrow Xs i \rightarrow Ys i \quad c : \text{Ext } C Xs} \right) \\ \frac{\text{map } C fs x : \text{Ext } C Ys}{\text{map } C fs x : \text{Ext } C Ys}$$

5.3 Coproducts and products

A constant operator is represented by a container which has no positions, e.g. the following containers represent the empty and the unit type:

$$\underline{\text{let}} \left(\frac{}{\text{CZero} : \text{Cont } n} \right) ; \text{CZero} \Rightarrow \text{cont Zero } (\lambda i ; s \Rightarrow \text{Zero})$$

$$\underline{\text{let}} \left(\frac{}{\text{COne} : \text{Cont } n} \right) ; \text{COne} \Rightarrow \text{cont One } (\lambda i ; s \Rightarrow \text{Zero})$$

Given two containers $C = \text{cont } S P, D = \text{cont } T Q$ we construct their coproduct or sum. On the shapes this is just the type-theoretic coproduct $\text{Plus } S T$ as defined earlier. What is a position in $\text{Plus } S T$? If our shape is of the form $\text{inl } s$ then it is given by $P s$, on the other hand if it is of the form $\text{inr } t$ then it is given by $Q t$. Abstracting shapes and positions, we arrive at:

$$\underline{\text{data}} \left(\frac{P : A \rightarrow \star \quad Q : B \rightarrow \star \quad ab : \text{Plus } A B}{\text{PPlus } P Q ab : \star} \right) \underline{\text{where}} \\ \left(\frac{p : P a}{\text{pinl } p : \text{PPlus } P Q (\text{Inl } a)} \right) ; \left(\frac{q : Q b}{\text{pinr } q : \text{PPlus } P Q (\text{Inr } b)} \right)$$

Putting everything together we define the containers as:

$$\begin{aligned} & \underline{\text{let}} \left(\frac{C, D : \text{Cont } n}{\mathbf{CPlus } C D : \text{Cont } n} \right) \\ & \mathbf{CPlus } C D \Leftarrow \text{case } C \{ \\ & \quad \mathbf{CPlus } (\text{cont } S P) D \Leftarrow \text{case } D \{ \\ & \quad \quad \mathbf{CPlus } (\text{cont } S P) (\text{cont } T Q) \\ & \quad \quad \Rightarrow \text{cont } (\mathbf{Plus } S T) (\lambda i \Rightarrow \mathbf{PPlus } (P i) (Q i)) \} \} \end{aligned}$$

Let's turn our attention to products: on shapes again this is just the type-theoretic product **Times**. Given two containers $C = \text{cont } S P, D = \text{cont } T Q$ as above what are the positions in a product shape $\text{pair } s t : \mathbf{Times } S T$? There are two possibilities: either the position is in the left component, then it is given by $P s$ or it is in the right component then it is given by $Q t$. Abstracting shapes and positions again we define abstractly:

$$\begin{aligned} & \underline{\text{data}} \left(\frac{P : A \rightarrow \star \quad Q : B \rightarrow \star \quad ab : \mathbf{Times } A B}{\mathbf{PTimes } P Q ab : \star} \right) \text{ where} \\ & \left(\frac{p : P a}{\mathbf{pleft } p : \mathbf{PTimes } P Q (\mathbf{Pair } a b)} \right) ; \left(\frac{q : Q b}{\mathbf{pright } q : \mathbf{PTimes } P Q (\mathbf{Pair } a b)} \right) \end{aligned}$$

and we define the product of containers as:

$$\begin{aligned} & \underline{\text{let}} \left(\frac{C, D : \text{Cont } n}{\mathbf{CTimes } C D : \text{Cont } n} \right) \\ & \mathbf{CTimes } C D \Leftarrow \text{case } C \{ \\ & \quad \mathbf{CTimes } (\text{cont } S P) D \Leftarrow \text{case } D \{ \\ & \quad \quad \mathbf{CTimes } (\text{cont } S P) (\text{cont } T Q) \\ & \quad \quad \Rightarrow \text{cont } (\mathbf{Times } S T) (\lambda i \Rightarrow \mathbf{PTimes } (P i) (Q i)) \} \} \end{aligned}$$

Exercise 14. Define an operation on containers which interprets constant exponentiation as described in section 4.3, i.e. define

$$\underline{\text{let}} \left(\frac{A : \star \quad C : \text{Cont } n}{\mathbf{Carr } A C : \text{Cont } n} \right)$$

5.4 Structural operations

If we want to faithfully interpret the universe of context-free or strictly positive types, we also have to find counterparts for the structural operation **vl** (last variable), **wk** (weakening) and **'let'** (local definition).

The interpretation of **vl** is straightforward: There is only one shape and in the family of positions $P : \mathbf{Fin } (\text{suc } n)$ there is only one position at index **fz**:

$$\underline{\text{let}} \left(\frac{}{\mathbf{cvl} : \text{Cont } n} \right) ; \mathbf{cvl} \Rightarrow \text{cont } \mathbf{One} (\lambda i; s \Rightarrow i = \mathbf{fz})$$

Weakening isn't much harder: the shape stays the same but the position indices get shifted by one assigning no positions to index **fz**. We define first an auxilliary operator on positions:

$$\underline{\text{let}} \left(\frac{P : \text{Fin } n \rightarrow S \rightarrow \star \quad i\text{Fin}(\text{suc } n) \quad s : S}{\text{Pwk } P i s : \star} \right)$$

$$\begin{aligned} \text{Pwk } P i s &\leftarrow \underline{\text{case}} i \{ \\ \text{Pwk } P \text{fz } s &\Rightarrow \text{Zero} \\ \text{Pwk } P (\text{fs } i) s &\Rightarrow P i s \} \end{aligned}$$

and use this to define:

$$\underline{\text{let}} \left(\frac{C : \text{Cont } n}{\text{cwk } C : \text{Cont}(\text{suc } n)} \right); \quad \text{cwk } C \leftarrow \underline{\text{case}} C \{ \\ \text{cwk}(\text{cont } S P) \Rightarrow \text{cont } S (\text{Pwk } P) \}$$

The case of local definition is more interesting. We assume as given two containers: $C = \text{cont } S P : \text{Cont}(\text{suc } n)$, $D = \text{cont } T Q : \text{Cont } n$. We can split $P : (\text{Fin}(\text{suc } n) \rightarrow \star)$ into $P z$ (the position where we substitute) and $P' : (\text{Fin } n) \Rightarrow \star$ defined as $P' i \Rightarrow P(\text{fs } i)$. The shape of the new container is given by a shape of the first container $s : S$ and a function which assigns to any position which is substituted a shape of the 2nd container $g : (P z) \rightarrow T$. As before we abstract from the specific position types and define abstractly:

$$\underline{\text{data}} \left(\frac{S, T : \star \quad P : S \rightarrow \star}{\text{Slet } S T P : \star} \right) \quad \underline{\text{where}} \left(\frac{s : S \quad f : P s \rightarrow T}{\text{slet } s f : \text{Slet } S T P} \right)$$

What are the positions in the new container? The result of the substitution is an n -ary container, a position at index i is either a position in the first container, i.e. $P' i$ or a position in the substituted position $p \in P \text{fz}$ together with a position in the 2nd container, i.e. $Q(gp)$. Hence we define a general operator for positions in **Slet**:

$$\underline{\text{data}} \left(\frac{S, T : \star \quad P, P' : S \rightarrow \star \quad Q : T \rightarrow \star \quad x : \text{Slet } S T P}{\text{Plet } S T P P' Q x : \star} \right)$$

$$\underline{\text{where}} \left(\frac{P' : S \rightarrow \star \quad q : Q(f P)}{\text{ppos } p : \text{Plet } S T P P' Q(\text{slet } s f)} \right)$$

$$\left(\frac{P : S \rightarrow \star \quad Q : T \rightarrow \star \quad f : P s \rightarrow T \quad p : P s \quad q : Q(f p)}{\text{qpos } p q : \text{Plet } S T P P' Q(\text{slet } s f)} \right)$$

Putting the components together we are able to define the substitution operator:

$$\begin{aligned} & \underline{\text{let}} \left(\frac{C : \text{Cont}(\text{suc } n) \quad D : \text{Cont } n}{\text{Clet } C D : \text{Cont } n} \right) \\ & \text{Clet } C D \leftarrow \underline{\text{case}} C \{ \\ & \quad \text{Clet } (\text{cont } S P) D \leftarrow \underline{\text{case}} D \{ \\ & \quad \quad \text{Clet } (\text{cont } S P) (\text{cont } T Q) \\ & \quad \quad \Rightarrow \text{cont } (\text{Slet } S T (P \text{ fz})) (\lambda i \Rightarrow \text{Plet } S T (P \text{ fz}) (P (\text{fs } i)) (P i)) \} \} \end{aligned}$$

5.5 Inductive types (μ)

To interpret the **mu** constructor we assume as given an $n + 1$ -ary container $C = \text{cont } S P : \text{Cont}(\text{suc } n)$, our goal is to find a container which represents the initial algebra wrt. to the first index. As before in the case of local definition let's define $P' : \text{Fin } n \rightarrow S \rightarrow \star$ as $P' i \Rightarrow P (\text{fs } i)$. The shape of this fixpoint container is given by trees whose nodes are labelled by $s : S$ and whose subtrees are indexed by $P \text{ fz}$: Clearly, to be able to construct a tree at all we need at least one shape such that there are no positions, i.e. $P \text{ fz } s$ is empty. Otherwise there are no leaves and the corresponding tree type is empty.

It turns out that the construction of these trees from S and $P \text{ fz}$ is a well known type constructor in Type Theory, called a W -type:

$$\underline{\text{data}} \left(\frac{S : \star \quad P : S \rightarrow \star}{\text{W } S P : \star} \right) \quad \underline{\text{where}} \left(\frac{s : S \quad f : P s \rightarrow \text{W } S P}{\text{sup } s f : \text{W } S P} \right)$$

Given a shape in form of a tree, the positions at index $i : \text{Fin } n$ correspond to path leading to a position in $P' i$ somewhere in the tree. We can define the types of paths in a tree in general:

$$\begin{aligned} & \underline{\text{data}} \left(\frac{S : \star \quad P, Q : S \rightarrow \star \quad x : \text{W } S P}{\text{P } W S P Q x : \star} \right) \\ & \underline{\text{where}} \left(\frac{q : Q s}{\text{here } q : \text{P } W S P Q (\text{sup } s f)} \right) ; \left(\frac{p : P s \quad r : \text{P } W S P Q (f p)}{\text{later } p r : \text{P } W S P Q (\text{sup } s f)} \right) \end{aligned}$$

The idea is that a path either exits at the top level node **here** at a position in $Q s$ or continues via one of the positions in $P s$. Putting shapes and paths together we arrive at the following definition:

$$\begin{aligned} & \underline{\text{let}} \left(\frac{C : \text{Cont}(\text{suc } n)}{\text{Cmu } C : \text{Cont } n} \right) \\ & \text{Cmu } C \leftarrow \underline{\text{case}} C \{ \\ & \quad \text{Cmu } (\text{cont } S P) \Rightarrow \text{cont } (\text{W } S P) (\lambda i \Rightarrow \text{P } W S (P \text{ fz}) (P (\text{fs } i))) \} \end{aligned}$$

5.6 Interpreting universes

Since we have constructed semantic counterparts to every syntactic constructor in **Ucf** we can interpret any type name by a container with the corresponding arity:

$$\begin{aligned} & \underline{\text{let}} \left(\frac{a : \mathbf{Ucf} \ n}{\mathbf{evalC} \ a : \mathbf{Cont} \ n} \right) \\ & \mathbf{evalC} \ a \Leftarrow \underline{\text{rec}} \ a \{ \\ & \quad \mathbf{evalC} \ a \Leftarrow \underline{\text{case}} \ a \{ \\ & \quad \quad \mathbf{evalC} \ \text{vl} \Rightarrow \mathbf{cvl} \\ & \quad \quad \mathbf{evalC} \ (\text{wk} \ a) \Rightarrow \mathbf{cwk} \ (\mathbf{evalC} \ a) \\ & \quad \quad \mathbf{evalC} \ '0' \Rightarrow \mathbf{CZero} \\ & \quad \quad \mathbf{evalC} \ ('plus' \ a \ b) \Rightarrow \mathbf{CPlus} \ (\mathbf{evalC} \ a) \ (\mathbf{evalC} \ b) \\ & \quad \quad \mathbf{evalC} \ '1' \Rightarrow \mathbf{COne} \\ & \quad \quad \mathbf{evalC} \ ('times' \ a \ b) \Rightarrow \mathbf{CTimes} \ (\mathbf{evalC} \ a) \ (\mathbf{evalC} \ b) \\ & \quad \quad \mathbf{evalC} \ ('let' \ f \ a) \Rightarrow \mathbf{Clet} \ (\mathbf{evalC} \ f) \ (\mathbf{evalC} \ a) \\ & \quad \quad \mathbf{evalC} \ ('mu' \ f) \Rightarrow \mathbf{Cmu} \ (\mathbf{evalC} \ f) \} \} \end{aligned}$$

Combining **evalC** with **Ext** we can assign to any name in **Ucf** an operator on types:

$$\underline{\text{let}} \left(\frac{a : \mathbf{Ucf} \ n \quad Xs : \mathbf{Fin} \ n \rightarrow \star}{\mathbf{eval} \ a \ Xs : \star} \right) ; \mathbf{eval} \ a \ Xs \Rightarrow \mathbf{Ext} \ (\mathbf{evalC} \ a) \ Xs$$

The advantage is that we can apply our operators to any types, not just those which have name. Using the solution to exercise 13 we also obtain a generic map function.

Sofar we have only interpreted the type names, i.e. the inhabitants of **Ucf** n , what about the elements, i.e. the inhabitants of **Elcf** aas ? Using **Ext** we can define a semantic version of **Elcf**:

$$\underline{\text{data}} \left(\frac{n : \mathbf{Nat}}{\mathbf{CTel} \ n : \star} \right) \underline{\text{where}} \left(\frac{}{\mathbf{ctnil} : \mathbf{Tel} \ \text{zero}} \right) ; \left(\frac{a : \mathbf{Cont} \ n \quad as : \mathbf{Tel} \ n}{\mathbf{ctcons} \ a \ as : \mathbf{Tel} \ (\text{suc} \ n)} \right)$$

$$\underline{\text{let}} \left(\frac{Cs : \mathbf{Tel} \ n \quad i : \mathbf{Fin} \ n}{\mathbf{TelEl} \ Cs \ i : \star} \right)$$

$$\begin{aligned} & \mathbf{TelEl} \ Cs \ i \Leftarrow \underline{\text{rec}} \ Cs \{ \\ & \quad \mathbf{TelEl} \ Cs \ i \Leftarrow \underline{\text{case}} \ i \{ \\ & \quad \quad \mathbf{TelEl} \ Cs \ \text{fz} \Leftarrow \underline{\text{case}} \ Cs \{ \\ & \quad \quad \quad \mathbf{TelEl} \ (\text{ctcons} \ C \ Cs) \ \text{fz} \Rightarrow \mathbf{Ext} \ C \ (\mathbf{TelEl} \ Cs) \} \\ & \quad \quad \mathbf{TelEl} \ Cs \ (\text{fs} \ i) \Leftarrow \underline{\text{case}} \ Cs \{ \\ & \quad \quad \quad \mathbf{TelEl} \ (\text{ctcons} \ C \ Cs) \ (\text{fs} \ i) \Rightarrow \mathbf{TelEl} \ Cs \ i \} \} \} \end{aligned}$$

$$\underline{\text{let}} \left(\frac{C : \mathbf{Cont} \ n \quad Cs : \mathbf{Tel} \ n}{\mathbf{CEl} \ C \ Cs : \star} \right) ; \mathbf{CEl} \ C \ Cs \Rightarrow \mathbf{Ext} \ C \ (\mathbf{TelEl} \ Cs)$$

Exercise 15. Implement semantic counterparts of the constructor for **Elcf** giving rise to an interpretation of **Elcf** by **CEl**. Indeed, this interpretation is exhaustive and disjoint.

5.7 Small containers

We have given a translation of the context free types as containers, but as exercise 14 shows, these capture more than just the context free types, in fact it corresponds to the strictly positive universe. As a result we cannot derive a semantic version of generic equality which is *typical* of the smaller universe.

We can, however, define a notion of container which captures precisely the context free types and give a semantic version **geq** for these containers which we christen ‘small containers’.

A container is small if there is a decidable equality on its shapes and if the positions at a given shape are finite, so:

$$\begin{array}{l}
 \underline{\text{let}} \left(\frac{A : \star}{\mathbf{DecEq} A : \star} \right) \\
 \mathbf{DecEq} A \Rightarrow \forall a, a' \Rightarrow \mathbf{Dec} (a = a') \\
 \\
 \underline{\text{data}} \left(\frac{n : \mathbf{Nat}}{\mathbf{SCont} n : \star} \right) \\
 \text{where} \left(\frac{S : \star \quad eqS : \mathbf{DecEq} S \quad P : \mathbf{Fin} n \rightarrow S \rightarrow \mathbf{Nat}}{\mathbf{scont} S eqS P : \mathbf{SCont} n} \right) \\
 \\
 \underline{\text{data}} \left(\frac{C : \mathbf{SCont} \quad Xs : \mathbf{Fin} n \rightarrow \star}{\mathbf{SExt} C Xs : \star} \right) \\
 \text{where} \left(\frac{s : S \quad f : \forall i : \mathbf{Fin} n \Rightarrow \mathbf{Fin} (P i s) \rightarrow Xs i}{\mathbf{sext} s f : \mathbf{SCont} S eq P} \right)
 \end{array}$$

We can redefine the variable case, disjoint union, products and the fix point operator for these containers, for instance:

$$\begin{array}{l}
 \underline{\text{let}} \left(\frac{C, D : \mathbf{SCont} n}{\mathbf{SCTimes} C D : \mathbf{Cont} n} \right) \\
 \mathbf{SCTimes} C D \Leftarrow \underline{\text{case}} C \{ \\
 \quad \mathbf{SCTimes} (\mathbf{scont} S eqS P) D \Leftarrow \underline{\text{case}} D \{ \\
 \quad \quad \mathbf{SCTimes} (\mathbf{scont} S eqS P) (\mathbf{scont} T eqT Q) \\
 \quad \quad \Rightarrow \mathbf{scont} (\mathbf{Times} S T) \\
 \quad \quad \quad (\mathbf{TimesEq} eqS eqT) \\
 \quad \quad \quad (\lambda i; s \Rightarrow \mathbf{plus} (P i s) (Q i s)) \} \}
 \end{array}$$

Where **TimesEq** is a proof that cartesian product preserves decidable equality by comparing pointwise:

$$\underline{\text{let}} \left(\frac{\text{eqS} : \mathbf{DecEq} \ S \quad \text{eqT} : \mathbf{DecEq} \ T}{\mathbf{TimesEq} \ \text{eqS} \ \text{eqT} : \mathbf{DecEq} \ (\mathbf{Times} \ S \ T)} \right)$$

Our generic equality for small containers is then a proof that **SExt** preserves equality:

$$\underline{\text{let}} \left(\frac{C : \mathbf{SCont} \ n \quad Xs : \mathbf{Fin} \ n \rightarrow \star \quad eqs : \forall i : \mathbf{Fin} \ n \Rightarrow \mathbf{DecEq} \ (Xs \ i)}{\mathbf{SContEq} \ C \ Xs \ eqs : \mathbf{DecEq} \ (\mathbf{SExt} \ C \ Xs)} \right)$$

Exercise 16. Complete the construction of **SCTimes** and develop operators constructing disjoint union, local definition, fixed points, and variables for small containers. Finally construct the definition of **SContEq**.

To work with Epigram's built in equality you will need to use the fact that application preserves equality:

$$\underline{\text{let}} \left(\frac{f, g : S \rightarrow T \quad a, b : S \quad p : f = g \quad q : a = b}{\mathbf{applEq} \ p \ q : f \ a = g \ b} \right)$$

$\mathbf{applEq} \ p \ q \Leftarrow \underline{\text{case}} \ p \ \{$
 $\quad \mathbf{applEq} \ \text{refl} \ q \Leftarrow \underline{\text{case}} \ q \ \{$
 $\quad \quad \mathbf{applEq} \ \text{refl} \ \text{refl} \Rightarrow \text{refl} \ \} \ \}$

And that constructors are disjoint, so for example $\mathbf{Inl} \ a = \mathbf{Inr} \ b$ is a provably empty type:

$$\underline{\text{let}} \left(\frac{a : A \quad b : B \quad p : (\mathbf{Inl} \ a : \mathbf{Plus} \ A \ B) = (\mathbf{Inr} \ b : \mathbf{Plus} \ A \ B)}{\mathbf{InlneqInr} \ p : X} \right)$$

$\mathbf{InlneqInr} \ p \Leftarrow \underline{\text{case}} \ p$

6 Derivatives

In [16] Huët introduced the *zipper* as a datatype to represent a position within a tree. The basic idea is that at every step on the path to the current position we remember the context left over of our path. E.g. in the example of unlabelled binary trees:

$$\underline{\text{data}} \left(\frac{A : \star}{\mathbf{BT} \ A : \star} \right) \text{ where } \left(\frac{a : A}{\mathbf{leaf} \ a : \mathbf{BT} \ A} \right) ; \left(\frac{l, r : \mathbf{BT} \ A}{\mathbf{node} \ l \ r : \mathbf{BT} \ A} \right)$$

the corresponding zipper is:

$$\underline{\text{data}} \left(\frac{}{\mathbf{Zipper} : \star} \right)$$

$$\underline{\text{where}} \left(\frac{l : \mathbf{Zipper} \quad r : \mathbf{BT} \ A}{\mathbf{left} \ l \ r : \mathbf{Zipper}} \right) ; \left(\frac{l : \mathbf{BT} \ A \quad r : \mathbf{Zipper}}{\mathbf{right} \ l \ r : \mathbf{Zipper}} \right) ; \left(\frac{}{\mathbf{here} : \mathbf{Zipper}} \right)$$

One of the operations on a zipper is to plug in a binary tree which fills the position with a subtree, i.e. we define:

$$\underline{\text{let}} \left(\frac{z : \text{Zipper} \quad t : \text{BT}}{\text{plug } z \ t : \text{BT}} \right); \text{plug } z \ t \Leftarrow \underline{\text{rec}} \ z \ \{ \\ \text{plug } z \ t \Leftarrow \underline{\text{case}} \ z \ \{ \\ \text{plug } (\text{left } l \ r) \ t \Rightarrow \text{node } (\text{plug } l \ t) \ r \\ \text{plug } (\text{right } l \ r) \ t \Rightarrow \text{node } l \ (\text{plug } r \ t) \\ \text{plug } \text{here } t \Rightarrow t \} \}$$

Clearly, the zipper is a generic construction, which should certainly work on any context-free type. When trying to express the general scheme of a zipper, Conor McBride realized that a zipper is always a sequence of basic steps which arise as the formal derivative of the functor defining the datatype. I.e. if our datatype is $\mu X.FX$, e.g. $\mu X.1+X \times X$ in the example of binary trees, then the corresponding zipper is $\text{List}(\partial F(\mu X.FX))$. In the binary tree example $FX = 1 + X \times X$ and $\partial F X = 2 \times X$. Indeed Zipper is isomorphic to $\text{List}(2 \times \text{BT})$.

6.1 Derivatives of context-free types

We will here concentrate on the notion of the partial derivative of an n -ary operator on types, which corresponds to the type of *one hole contexts* of the given type. This is an alternative explanation of the formal laws of derivatives and we shall define an operator on context-free types following this intuition:

$$\underline{\text{let}} \left(\frac{a : \text{Ucf } n \quad i : \text{Fin } n}{\text{partial } a \ i : \text{Ucf } n} \right)$$

We define this operation by structural recursion on a , let's consider the polynomial cases: what is the derivative, i.e. the type of one hole contexts of 'plus' $a \ b$? We either have a hole in an element of a or a hole in an element of b , hence:

$$\text{partial } (\text{'plus'} \ a \ b) \ i \Rightarrow \text{'plus'} \ (\text{partial } a \ i) \ (\text{partial } b \ i)$$

Maybe slightly more interesting, what is the type of one-hole contexts of 'times' $a \ b$? A hole in a pair is either a hole in the left component, leaving the right intact or symmetrically, a hole on the right, leaving the left intact. Hence we arrive at

$$\text{partial } (\text{'times'} \ a \ b) \ i \Rightarrow \text{'plus'} \ (\text{'times'} \ (\text{partial } a \ i) \ b) \ (\text{'times'} \ a \ (\text{partial } b \ i))$$

which indeed corresponds to the formal derivative of a product, albeit we arrived at it using a completely different explanation. Unsurprisingly, the derivative of a constant is '0', since there are no holes to plug:

$$\begin{aligned} \text{partial } \text{'0'} \ i &\Rightarrow \text{'0'} \\ \text{partial } \text{'1'} \ i &\Rightarrow \text{'1'} \end{aligned}$$

Structural operations like variables and weakening are usually ignored in Calculus, an omission we will have to fill here to be able to implement **partial**

for those cases. In both cases we have to inspect i : for vl we have exactly one choice if $i = \text{fz}$ and none otherwise, hence we have:

$$\begin{aligned} \text{partial vl fz} &\Rightarrow \text{'1'} \\ \text{partial vl (fs } i) &\Rightarrow \text{'0'} \end{aligned}$$

In the case of $\text{wk } a$ the situation is reversed, there is no choice if $i = \text{fz}$ and otherwise we recur structurally:

$$\begin{aligned} \text{partial (wk } a) \text{ fz} &\Rightarrow \text{'0'} \\ \text{partial (wk } a) \text{ (fs } i) &\Rightarrow \text{wk (partial } a \text{ } i) \end{aligned}$$

The case of local definitions $\text{'let' } f \ a$ corresponds to the chain rule in Calculus. A hole in an element of $\text{'let' } f \ a$ is either a hole in f but not in the variable which is substituted, or it is a hole in f for the variable we are substituting together with a hole in a . More formally we have:

$$\begin{aligned} \text{partial ('let' } f \ a) \ i \\ \Rightarrow \text{'plus' ('let' (partial } f \ \text{(fs } i)) \ a) \ \text{'times' ('let' (partial } f \ \text{fz}) \ a) \ \text{(partial } a \ i)} \end{aligned}$$

Note that the condition *but not in the variable which is substituted* corresponds to shifting the variable index by one.

The case for initial algebras $\text{'mu' } f$ has no counterpart in calculus. However, it can be derived using the chain rule above: we know that $\text{'mu' } f$ is isomorphic to $\text{'let' } f \ (\text{'mu' } f)$. Now using the chain rule we arrive at

$$\begin{aligned} \text{'plus' ('let' (partial } f \ \text{(fs } i)) \ (\text{'mu' } f)) \\ \text{'times' ('let' (partial } f \ \text{fz}) \ (\text{'mu' } f)) \ \text{(partial ('mu' } f) \ i)} \end{aligned}$$

This expression is recursive in $\text{partial ('mu' } f) \ i$ hence we obtain the formal derivative by taking the initial algebra of it:

$$\begin{aligned} \text{partial ('mu' } f) \ i &\Rightarrow \text{'mu' ('plus' (wk ('let' (partial } f \ \text{(fs } i)) \ (\text{'mu' } f))) \\ &\quad \text{'times' (wk ('let' (partial } f \ \text{fz}) \ (\text{'mu' } f)) \ \text{vl))} \end{aligned}$$

A closer analysis shows that the use of initial algebras here is justified by the fact that we are only interested in holes which appear at some finite depths.

As an example consider the derivative of lists partial list fz , after applying some simplification we obtain $\text{'mu' 'plus' (wklist) ('times' vl (wkvl))}$ or reexpressed in a more standard notation $\mu X. (\text{list } A) + A \times X$, which can be easily seen to correspond to lists with a hole for A .

We summarize the definition of **partial**:

```

partial  $a\ i \Leftarrow \text{rec } a \{$ 
partial  $a\ i \Leftarrow \text{case } a \{$ 
  partial  $\text{vl } i \Leftarrow \text{case } i \{$ 
    partial  $\text{vl } \text{fz} \Rightarrow \text{'1'}$ 
    partial  $\text{vl } (\text{fs } i) \Rightarrow \text{'0'}$ 
  partial  $(\text{wk } a)\ i \Leftarrow \text{case } i \{$ 
    partial  $(\text{wk } a)\ \text{fz} \Rightarrow \text{'0'}$ 
    partial  $(\text{wk } a)\ (\text{fs } i) \Rightarrow \text{wk } (\text{partial } a\ i)$ 
  partial  $\text{'0' } i \Rightarrow \text{'0'}$ 
  partial  $(\text{'plus' } a\ b)\ i \Rightarrow \text{'plus' } (\text{partial } a\ i)\ (\text{partial } b\ i)$ 
  partial  $\text{'1' } i \Rightarrow \text{'0'}$ 
  partial  $(\text{'times' } a\ b)\ i \Rightarrow$ 
     $\text{'plus' } (\text{'times' } (\text{partial } a\ i)\ b)\ (\text{'times' } a\ (\text{partial } b\ i))$ 
  partial  $(\text{'let' } f\ a)\ i \Rightarrow$ 
     $\text{'plus' } (\text{'let' } (\text{partial } f\ (\text{fs } i))\ a)\ (\text{'times' } (\text{'let' } (\text{partial } f\ \text{fz})\ a)\ (\text{partial } a\ i))$ 
  partial  $(\text{'mu' } f)\ i \Rightarrow \text{'mu' } (\text{'plus' } (\text{wk } (\text{'let' } (\text{partial } f\ (\text{fs } i))\ (\text{'mu' } f))))$ 
     $(\text{'times' } (\text{wk } (\text{'let' } (\text{partial } f\ \text{fz})\ (\text{'mu' } f))))\ \text{vl}) \}$ 

```

Exercise 17. Calculate (by hand) the derivative of labelled rose trees, i.e. ...

6.2 Generic plugging

To convince ourselves that the definition of derivatives as one hole contexts given above is correct we derive ⁶ a generic version of the generic plugging operation:

$$\underline{\text{let}} \left(\frac{a\ i\ x : \text{Elcf } (\text{partial } a\ i)\ as\ y : \text{Elcf } (\text{var } i)\ as}{\text{gplug } a\ i\ x\ y : \text{Elcf } a\ as} \right)$$

We construct **gplug** by recursion over x , however, unlike in the previous examples, which were completely data driven we have to analyze the type directly, i.e. we have to invoke `case a`. However, the operational behaviour of the plug operation doesn't actually depend on the type.

⁶ We were unable to convince Epigram to check all of the definition below due to a space leak in the current implementation. We are confident that this will be fixed in the next release of Epigram.

6.3 Derivatives of containers

Previously, we have defined derivatives by induction over the syntax of types. Using containers we can give a more direct, semantic definition. The basic idea can be related to derivatives of polynomials, i.e. the derivative of $f x = x^n$ is $f' x = n \times x^{n-1}$. As a first step we need to find a type-theoretic counterpart to the predecessor of a type by removing one element of the type. We define:

$$\underline{\text{data}} \left(\frac{A : \star \quad a : A}{\text{Minus } A \ a : \star} \right) \quad \underline{\text{where}} \left(\frac{a' : A \quad na : (a = a') \rightarrow \text{Zero}}{\text{minus } a \ na : \text{Minus } A \ a'} \right)$$

We can embed $\text{Minus } A \ a$ back into A :

$$\underline{\text{let}} \left(\frac{m : \text{Minus } A \ a}{\text{emb } m : A} \right) ; \quad \text{emb } m \Leftarrow \underline{\text{case}} \ m \{ \\ \text{emb } (\text{minus } a' \ na) \Rightarrow a' \}$$

We can analyze A in terms of $\text{Minus } A \ a$ by defining a view. An element of A is either a or it is in the range of emb :

$$\underline{\text{data}} \left(\frac{a; a' : A}{\text{MinusV } a \ a'} \right) \\ \underline{\text{where}} \left(\frac{}{\text{same } a : \text{MinusV } a \ a} \right) ; \left(\frac{m : \text{Minus } A \ a}{\text{other } m : \text{MinusV } a \ (\text{emb } m)} \right)$$

This view is exhaustive, if the A has a decidable equality:

$$\underline{\text{let}} \left(\frac{a, a' : A \quad eq : \text{Dec } (a = a')}{\text{minusV}' a \ a' \ eq : \text{MinusV } a \ a'} \right) \\ \text{minusV}' a \ a' \ eq \Leftarrow \underline{\text{case}} \ eq \{ \\ \text{minusV}' a \ a' \ (\text{yes refl}) \Rightarrow \text{same } a \\ \text{minusV}' a \ a' \ (\text{no } f) \Rightarrow \text{other } (\text{minus } a' \ f) \} \\ \underline{\text{let}} \left(\frac{eqA : \text{DecEq } A \quad a, a' : A}{\text{minusV } eqA \ a \ a' : \text{MinusV } a \ a'} \right) \\ \text{minusV } eqA \ a \ a' \Rightarrow \text{minusV}' a \ a' \ (eqA \ a \ a')$$

We are now ready to construct the derivative of containers and implement a variant **plug** for containers. To simplify the presentation we first restrict our attention to unary containers

Given a unary container $\text{ucont } S \ P$ its derivative is given by shapes which are the original shapes together with a chosen position, i.e. $\text{Sigma } S \ P$. The new type of positions is obtained by subtracting this chosen element from P hence we define:

$$\underline{\text{let}} \left(\frac{P : S \rightarrow \star \quad sp : \text{Sigma } S \ P}{\text{derivP } P \ sp : \star} \right) \\ \text{derivP } P \ sp \Leftarrow \underline{\text{case}} \ sp \{ \\ \text{derivP } P \ (\text{tup } s \ p) \Rightarrow \text{Minus } (P \ a) \ b \}$$

and hence the derivative of a unary container is given by:

$$\begin{aligned} & \text{let } \left(\frac{C : \mathbf{UCont}}{\mathbf{derivC} C : \mathbf{UCont}} \right) \\ & \mathbf{derivC} C \Leftarrow \text{case } C \{ \\ & \quad \mathbf{derivC} (\mathbf{ucont} S P) \Rightarrow \mathbf{ucont} (\mathbf{Sigma} S P) (\mathbf{deriv} P) \} \end{aligned}$$

While the definition above works for any unary container, we need decidability of equality on positions to define the generic plugging operation. Intuitively, we have to be able to differentiate between position to identify the location of a hole. Hence, only containers with a decidable equality are differentiable. We define a predicate on containers:

$$\text{data } \left(\frac{C : \mathbf{UCont}}{\mathbf{DecUCont} C : \star} \right) \text{ where } \left(\frac{\text{decP} : \forall s : S \Rightarrow \mathbf{DecEq} (P s)}{\mathbf{decUCont} \text{decP} : \mathbf{DecUCont} (\mathbf{ucont} S P)} \right)$$

Given a decidable unary container we can define the function **uplug** which given an element of the extension of the derivative of a container $x : \mathbf{UExt}(\mathbf{derivC} C) X$ and an element $y : X$ we can plug the hole with y thus obtaining an element of $\mathbf{UExt} C X$:

$$\begin{aligned} & \text{let } \left(\frac{\begin{array}{l} eq : \mathbf{DecEq} A \quad a : A \\ f : \mathbf{Minus} A a \rightarrow X \\ x : X \quad a' : A \end{array}}{\mathbf{mplug} eq a f x a' : X} \right) \\ & \mathbf{mplug} eq a f x a' \Leftarrow \text{view } \mathbf{minusV} eq a a' \{ \\ & \quad \mathbf{mplug} eq a f x a \Rightarrow x \\ & \quad \mathbf{mplug} eq a f x (\mathbf{emb} m) \Rightarrow f m \} \\ & \text{let } \left(\frac{\begin{array}{l} C : \mathbf{UCont} \quad d : \mathbf{DecUCont} C \\ x : \mathbf{UExt} (\mathbf{derivC} C) X \quad y : X \end{array}}{\mathbf{uplug} C d x y : \mathbf{UExt} C X} \right) \\ & \mathbf{uplug} C d x y \Leftarrow \text{case } C \{ \\ & \quad \mathbf{uplug} (\mathbf{ucont} S P) d x y \Leftarrow \text{case } d \{ \\ & \quad \quad \mathbf{uplug} (\mathbf{ucont} S P) (\mathbf{decUCont} \text{decP}) x y \Leftarrow \text{case } x \{ \\ & \quad \quad \quad \mathbf{uplug} (\mathbf{ucont} S P) (\mathbf{decUCont} \text{decP}) (\mathbf{uext} sp f) y \Leftarrow \text{case } sp \{ \\ & \quad \quad \quad \quad \mathbf{uplug} (\mathbf{ucont} S P) (\mathbf{decUCont} \text{decP}) (\mathbf{uext} (\mathbf{tup} s p) f) y \\ & \quad \quad \quad \quad \Rightarrow \mathbf{uext} a (\mathbf{mplug} (\text{decP } a) b f y) \} \} \} \} \} \end{aligned}$$

Exercise 19. Extend the derivative operator for containers to n -ary containers, i.e. define

$$\text{let } \left(\frac{C : \mathbf{Cont} n \quad i : \mathbf{Fin} n}{\mathbf{partialC} C i : \mathbf{Cont} n} \right)$$

To extend the plug operator we have to define decidability for an n -ary container. We also need to exploit that equality for finite types is decidable.

7 Conclusions and further work

Using dependent types we were able to define different universes and generic operations on them. We have studied two fundamentally different approaches: a semantic approach, first using finite types and the container types and a syntactic approach where the elements are defined inductively. Further work needs to be done to relate the two more precisely, they are only two views of the same collection of types. We have already observed that there is a trade-off between the size of the universe, i.e. the collection of types definable within it, and the number of generic operations. The previous section suggests that between the context-free types and the strictly positive types: the differentiable types, i.e. the types with a decidable equality on positions. Previous work in a more categorical framework [7] shows already that the types which are obtained by closing context-free types under a coinductive type former (ν) are still differentiable.

The size of a universe is not the only parameter we can vary, the universes we have considered here are still very coarse. E.g. while we have a more refined type system on the meta-level, dependent types, this is not reflected in our universes. We have no names for the family of finite types, the vectors or the family of elements of a universe itself. Recent, yet unpublished work, shows that it is possible to extend both the syntactic and the semantic approach to capture families of types, see [24, 8]. Another direction to pursue is to allow types where the positions are result of a quotient, like bags or multisets. We have already investigated this direction from a categorical point of view [6]; a typetheoretic approach requires a Type Theory which allows quotient types. Here our current work on *Observational Type Theory* [9] fits in very well.

Apart from the more theoretical questions regarding universes of datatypes there are more pragmatic issues. We don't want to work with isomorphic copies of our datatypes, but we want to be able to access the top-level types themselves. We are working on a new implementation of Epigram which will provide a quotation mechanism which makes the top-level universe accessible for the programmer. We also hope to be able to find a good pragmatic answer to vary the level of genericity, i.e. to be able to define generic operations for the appropriate universe without repeating definitions.

References

1. Michael Abbott. *Categories of Containers*. PhD thesis, University of Leicester, 2003.
2. Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In *Proceedings of Foundations of Software Science and Computation Structures*, 2003.

3. Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Representing nested inductive types using W-types. In *Automata, Languages and Programming, 31st International Colloquium (ICALP)*, pages 59 – 71, 2004.
4. Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.
5. Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of containers. In *Typed Lambda Calculi and Applications, TLCA*, 2003.
6. Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Constructing polymorphic programs with quotient types. In *7th International Conference on Mathematics of Program Construction (MPC 2004)*, 2004.
7. Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. ∂ for data. *Fundamentae Informatica*, 65(1,2):1 – 28, March 2005. Special Issue on Typed Lambda Calculi and Applications 2003.
8. Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. Manuscript, available online, February 2006.
9. Thorsten Altenkirch and Conor McBride. Towards observational type theory. Manuscript, available online, February 2006.
10. Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.
11. Thorsten Altenkirch and Tarmo Uustalu. Normalization by evaluation for λ^{-2} . In *Functional and Logic Programming*, number 2998 in LNCS, pages 260 – 275, 2004.
12. Karl Crary, Stephanie Weirich, , and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.
13. Peter Dybjer. Inductive Sets and Families in Martin-Löf’s Type Theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. CUP, 1991.
14. Ralf Hinze and Andres Löb. “scrap your boilerplate” revolutions. In Tarmo Uustalu, editor, *Mathematics of Program Construction, 2006*, volume 4014 of LNCS, pages 180–208. Springer-Verlag, 2006.
15. Ralf Hinze, Andres Löb, and Bruno C. D. S. Oliveira. ”scrap your boilerplate” reloaded. In Masami Hagiya and Philip Wadler, editors, *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 13–29. Springer, 2006.
16. Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
17. Zhaohui Luo and Robert Pollack. LEGO Proof Development System: User’s Manual. Technical Report ECS-LFCS-92-211, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.
18. Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, LNCS 806. Springer-Verlag, 1994. Selected papers from the Int. Workshop TYPES ’93, Nijmegen, May 1993.
19. Conor McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. Available at <http://www.cs.nott.ac.uk/~ctm/diff.pdf>, 2001.
20. Conor McBride. Epigram, 2004. <http://www.dur.ac.uk/CARG/epigram>.
21. Conor McBride. Epigram: Practical programming with dependent types. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming 2004*, Lecture Notes in Computer Science. Springer-Verlag, 2005+. Revised lecture notes from the International Summer School in Tartu, Estonia.
22. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.

23. Peter Morris, Thorsten Altenkirch, and Conor McBride. Exploring the regular tree types. In Christine Paulin-Mohring Jean-Christophe Filliatre and Benjamin Werner, editors, *Types for Proofs and Programs (TYPES 2004)*, Lecture Notes in Computer Science, 2006.
24. Thorsten Altenkirch Peter Morris and Neil Ghani. Constructing strictly positive families. Submitted for publication, August 2006.
25. Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy type inference for higher-rank types and impredicativity. In *Proceedings of the International Conference on Functional Programming (ICFP) 2006*, September 2006.
26. Stephanie Weirich. RepLib: A library for derivable type classes. In Andres Löh, editor, *Proceedings of the ACM Haskell Workshop, 2006*, 2006. to appear.

A The universe of finite types

```

-----!
data (-----!
      ! Ufin : * )

      ( a, b : Ufin !
where (-----! ; !-----! ; (-----! ; !-----!
      ! '0' : ! ! 'plus' a b ! ! '1' : ! ! 'times' a b !
      ! Ufin ) ! : Ufin ) ! Ufin ) ! : Ufin )
-----!

      ( a : Ufin !
data !-----!
      ! Elfin a : * )

      ( b ; a ; x : Elfin a !
where !-----!
      ! inl x : Elfin ('plus' a b) )

      ( a ; b ; x : Elfin b !
!-----!
      ! inr x : Elfin ('plus' a b) )

      ( x : Elfin a ; y : Elfin b !
(-----! ; !-----!
      ! void : Elfin '1' ) ! pair x y : Elfin ('times' a b) )
-----!

```

B The universe of context-free types

```

-----!
      ( n : Nat !
data !-----!
```

```

! Ucf n : * )

where (-----!
! v1 : Ucf (suc n) )

( a : Ucf n !
!-----!
! wk a : Ucf (suc n) )

(-----!
! '0' : Ucf n )

( a, b : Ucf n !
!-----!
! 'plus' a b : Ucf n )

( f : Ucf (suc n) ! ( f : !
! a, b : ! ! Ucf (suc !
! Ucf n ! ! a : Ucf n ! ! % n) !
(-----! ; !-----! ; !-----! ; !-----!
! '1' : ! ! 'times' a b ! ! 'let' f a ! ! mu f : !
! Ucf n ) ! : Ucf n ) ! : Ucf n ) ! Ucf n )

-----
( n : Nat ! ( a : Ucf n ; as : Tel n !
data !-----! where (-----! ; !-----!
! Tel n : * ) ! tnil : Tel zero ) ! tcons a as : Tel (suc n) )

-----
( a : Ucf n ; as : Tel n !
data !-----!
! Elcf a as : * )

( b ; a ; x : Elcf a as !
where !-----!
! inl x : Elcf ('plus' a b) as )

( a ; b ; x : Elcf b as !
!-----!
! inr x : Elcf ('plus' a b) as )

(-----!
! void : Elcf '1' as )

( x : Elcf a as ; y : Elcf b as !
!-----!
! pair x y : Elcf ('times' a b) as )

```

```

(      x : Elcf a as      !
!-----!
! top x : Elcf vl (tcons a as) )

(      x : Elcf a as      !
!-----!
! pop x : Elcf (wk a) (tcons b as) )

(  x : Elcf f (tcons a as)  !    ( x : Elcf f (tcons (mu f) as) !
!-----! ; !-----!
! push x : Elcf ('let' f a) as )    !    in x : Elcf (mu f) as    )

```

C The universe of strictly positive types

```

(  n : Nat  !
data !-----!
! Usp n : * )

where (-----!
! vl : Usp (suc n) )

(  a : Usp n  !
!-----!
! wk a : Usp (suc n) )

(-----!
! '0' : Usp n )

(  a, b : Usp n  !
!-----!
! 'plus' a b : Usp n )

(  A : * ;  b : Usp n  !
!-----!
! 'arr' A b : Usp n )

(  f : Usp (suc n) !    (  f :      !
! a, b :      !    !    Usp (suc !
!   Usp n  !    !    !% n) !
!-----! ; !-----! ; !-----! ; !-----!
! '1' :      !    ! 'times' a b !    ! 'let' f a      !    ! mu f :      !
! Usp n )    !    : Usp n    )    !    : Usp n    )    !    Usp n    )

```

```

-----
      ( n : Nat !
data !-----! where (-----! ; !-----!
      ! Tel n : * )      ! tnil : Tel zero )      ! tcons a as : Tel (suc n) )
-----

      ( a : Usp n ; as : Tel n !
data !-----!
      !      Elsp a as : *      )

      ( b ; a ; x : Elsp a as !
where !-----!
      ! inl x : Elsp ('plus' a b) as )

      ( a ; b ; x : Elsp b as !
!-----!
      ! inr x : Elsp ('plus' a b) as )

      (-----!
      ! void : Elsp '1' as )

      ( x : Elsp a as ; y : Elsp b as !
!-----!
      ! pair x y : Elsp ('times' a b) as )

      ( f : A -> Elsp b as !
!-----!
      ! fun f : Elsp ('arr' A b) as )

      ( x : Elsp a as !
!-----!
      ! top x : Elsp vl (tcons a as) )

      ( x : Elsp a as !
!-----!
      ! pop x : Elsp (wk a) (tcons b as) )

      ( x : Elsp f (tcons a as) ! ( x : Elsp f (tcons (mu f) as) !
!-----! ; !-----!
      ! push x : Elsp ('let' f a) as ) ! in x : Elsp (mu f) as )
-----

```