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 [15, 21] 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 [28, 37] 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, $El : U \to \star$ which assigns to every name $a : U$ the type of its elements $El\ a : \star$—we call this the *extension* of the name a. Historically, universes have been already used by Type Theory to capture the predicative hierarchy of types, first introduced by Russell to prevent set-theoretic paradoxes:

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

If we want to avoid chains of $:$ we can represent this hierarchy as:

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

Here $\star$ plays the role of a superuniverse in which all universes can be embedded, while U_i is the type of (names of) types at level i. The operation El_i assigns to any name of a type at level i the type of its elements. In particular u_i is the name of the previous universe at level $i+1$ and hence $El_{i+1}\ u_i = U_i$.

The predicative hierarchy of universes is necessary to have types of types without running into paradoxes (e.g. by having a type of all types). 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.

Related work

The idea to use dependent types for generic programming isn't new: starting with the pioneering work by Pfeifer and Rueß [36] who used the LEGO system as a vehicle for generic programming, the authors of [16] actually introduced a universe containing codes for dependent types. The latter is based on the work by Dybjer and Setzer on Induction-Recursion [18, 19] which can also be understood as universe constructions for non-dependent and dependent types. The first two authors of the present notes presented a universe construction for the first order fragment of Haskell datatypes including nested datatypes in [9] which was motivated by the work on generic Haskell [14, 25].

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 [32] for a definition of the language with many applications, [11] for a short and more recent overview and [31] 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 [30].

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 [35]. 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 [24] (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 [29], 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 [35].

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

All the code contained with these notes is available to download from the Epigram website [12].

2 Programming with Dependent Types in Epigram

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 [26]. The design of the interactive program and proof development environment is heavily influenced by the ALF system [27].

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:

```
data (---------!  where (------------! ;  (--------------!
     ! Bool : * )        ! true : Bool )   ! false : Bool )

                                          (   n : Nat    !
data (---------!  where (-----------! ;  !-------------!
     ! Nat : * )        ! zero : Nat )   ! suc n : Nat )
```

We first give the *formation rules*, Bool and Nat are types ($\star$ is the type of types) without any assumptions. We then introduce the *constructors*: true and false are Booleans; zero is a natural number and suc n is a natural number, if n is one. These declarations correspond to the Haskell datatypes:

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

The natural deduction notation may appear quite verbose for simple definitions, but you don't have to manage the layout for yourself—the editor does it for you. On paper, we prefer LaTeX to ASCII-art, and we take the liberty of typesetting declarations without the bracket delimiters and semicolon separators.

$$\underline{\text{data}}\ \frac{}{\mathsf{Nat} : \star}\quad \underline{\text{where}}\ \frac{}{\mathsf{zero} : \mathsf{Nat}}\ \frac{n : \mathsf{Nat}}{\mathsf{suc}\, n : \mathsf{Nat}}$$

The formation rule may contain assumptions as well, declaring the arguments to type constructors, e.g. in the case of lists:

```
     (   A : *    !                               ( a : A ;  as : List A !
data !------------!  where (-------------! ;  !---------------------!
     ! List A : * )        ! nil : List A )   !  cons a as : List A  )
```

In LaTeX this becomes:

$$\underline{\text{data}}\ \frac{A\ :\ \star}{\mathsf{List}\ A\ :\ \star}\quad \underline{\text{where}}\ \frac{}{\mathsf{nil}\ :\ \mathsf{List}\ A}\quad \frac{a\ :\ A \quad as\ :\ \mathsf{List}\ A}{\mathsf{cons}\ a\ as\ :\ \mathsf{List}\ a}$$

Here the type-valued arguments to the constructors are left implicit. However, as we shall see, the use of implicit arguments becomes more subtle with 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:

```
------------------------------------------------------------------
    ( A : *  !                              ( a : A ;  as : List A !
data !--------!  where (----------! ;  !----------------------!
    ! List A !         ! nil _A : !    !     cons _A a as     !
    !  : *   )         !   List A )    !      : List A        )
------------------------------------------------------------------
```

Here, the underscore overrides the implicit quantifier to allow the user to specify its value explicitly. In LaTeX we denote this by subscripting the argument itself, as you can see in the typeset version of the above:

$$\underline{\text{data}}\ \frac{A\ :\ \star}{\mathsf{List}\ A\ :\ \star}\quad \underline{\text{where}}\ \frac{A\ :\ \star}{\mathsf{nil}_A\ :\ \mathsf{List}\ A}\quad \frac{A\ :\ \star\quad a\ :\ A\quad as\ :\ \mathsf{List}\ A}{\mathsf{cons}_A\ a\ as\ :\ \mathsf{List}\ a}$$

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.

Defining functions

We define functions using let and generate patterns interactively. The text of a program records its construction. E.g. we define boolean **not** by case analysis on its first argument:

$$\underline{\text{let}}\ \ \frac{b\ :\ \mathsf{Bool}}{\mathbf{not}\ b\ :\ \mathsf{Bool}} \qquad \begin{array}{l} \mathbf{not}\ b \ \Leftarrow\ \underline{\text{case}}\ b \\ \quad \mathbf{not}\ \mathsf{true} \ \Rightarrow\ \mathsf{false} \\ \quad \mathbf{not}\ \mathsf{false} \ \Rightarrow\ \mathsf{true} \end{array}$$

Let-declarations use the two-dimensional rule syntax to declare a program's type: in Epigram, all top-level identifiers must be typed explicitly, but this type information is then propagated into their definitions, leaving them relatively free of annotation. Epigram programs are not sequences of prioritised equations as is traditional [33]. Rather, they are treelike: on the left-hand side, the *machine* presents a 'programming problem', e.g., to compute **not** b; on the right-hand side, we explain how to attack the problem in one of two ways:

$\Leftarrow$ **'by'** refinement into subproblems, using an *eliminator* like $\underline{\text{case}}\,b$, above; the machine then generates a bunch of subproblems for us to solve;

$\Rightarrow$ **'return'** an answer directly, giving an expression of the appropriate return type, which may well be more specific than the type we started with, thanks to the analysis of the problem into cases.

In ASCII source, this tree structure is made explicit using $\{\ldots\}$ markings: here, we drop these in favour of indentation.

Epigram programs thus explain the strategy by which they compute values:

case analysis is one such strategy. By ensuring that the available strategies are total (e.g., <u>case</u> eliminators cover all constructors), we guarantee that all programs terminate.[1] To implement a recursive program, we can invoke a <u>rec</u> eliminator, capturing the strategy of structural recursion on the nominated argument: recursive calls have to be structurally smaller in this argument. As simple examples, consider addition and multiplication of Peano natural numbers:

$$\underline{\text{let}}\ \frac{x, y\ :\ \mathsf{Nat}}{\mathbf{plus}\ x\ y\ :\ \mathsf{Nat}} \quad \begin{array}{l} \mathbf{plus}\ x\ y\ \Leftarrow\ \underline{\text{rec}}\ x \\ \quad \mathbf{plus}\ x\ y\ \Leftarrow\ \underline{\text{case}}\ x \\ \quad\quad \mathbf{plus}\ \mathsf{zero}\ y\ \Rightarrow\ y \\ \quad\quad \mathbf{plus}\ (\mathsf{suc}\ x)\ y\ \Rightarrow\ \mathsf{suc}\ (\mathbf{plus}\ x\ y) \end{array}$$

$$\underline{\text{let}}\ \frac{m, n\ :\ \mathsf{Nat}}{\mathbf{times}\ m\ n\ :\ \mathsf{Nat}} \quad \begin{array}{l} \mathbf{times}\ m\ n\ \Leftarrow\ \underline{\text{rec}}\ m \\ \quad \mathbf{times}\ m\ n\ \Leftarrow\ \underline{\text{case}}\ m \\ \quad\quad \mathbf{times}\ \mathsf{zero}\ n\ \Rightarrow\ \mathsf{zero} \\ \quad\quad \mathbf{times}\ (\mathsf{suc}\ m)\ n\ \Rightarrow\ \mathbf{plus}\ n\ (\mathbf{times}\ m\ n) \end{array}$$

It should be noted that these programs produce terms in Epigram's underlying type theory which use only standard elimination constants to perform recursion or case analysis. Programmng with <u>rec</u> is much more flexible than 'primitive recursion': it is straightforward to implement programs with deeper structural recursion, like the Fibonacci function, or lexicographically combined structural recursions, like the Ackermann function.

Data-type families

So far we haven't used dependent types explicitly. Dependent types come in *families* [20], indexed by data. A standard example is the family of vectors *indexed by length*:

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

Here $\mathsf{Vec}\, n\, X$ is the type of vectors, 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:

$$\underline{\text{let}}\ \frac{ys : \mathsf{Vec}\, (\mathsf{suc}\, m)\ Y}{\mathbf{vhead}\, ys : Y} \quad \begin{array}{l} \mathbf{vhead}\, ys \Leftarrow \underline{\text{case}}\, ys \\ \quad \mathbf{vhead}\, (\mathsf{vcons}\, y\, ys) \Rightarrow y \end{array}$$

Note that Epigram does not ask for a vnil case—vnil's length, zero, does not unify with suc m, the length of ys, so the case cannot ever arise.

[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.

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 $\mathsf{Fin}\,n$ represents the finite set $0, 1, \ldots, n-1$, i.e. exactly the positions in a vector of length n.

$$\underline{\text{data}}\ \frac{n : \mathsf{Nat}}{\mathsf{Fin}\,n : \star}\quad \underline{\text{where}}\ \frac{}{\mathsf{fz} : \mathsf{Fin}\,(\mathsf{suc}\,n)}\ \ \frac{i : \mathsf{Fin}\,n}{\mathsf{fs}\,i : \mathsf{Fin}\,(\mathsf{suc}\,n)}$$

Here fz represents the 0 which is present in any non-empty finite set, and $\mathsf{fs}\,i : \mathsf{Fin}\,(\mathsf{suc}\,n)$ represents $i+1$, given that $i : \mathsf{Fin}\,n$. Note that $\mathsf{Fin}\,\mathsf{zero}$ is meant to be empty: case analysis on a hypothetical element of $\mathsf{Fin}\,\mathsf{zero}$ leaves the empty set of patterns. The following table enumerates the elements up to $\mathsf{Fin}\,4$:

Fin 0	Fin 1	Fin 2	Fin 3	Fin 4	$\cdots$
	fz_0	fz_1	fz_2	fz_3	$\cdots$
		$\mathsf{fs}_1\ \mathsf{fz}_0$	$\mathsf{fs}_2\ \mathsf{fz}_1$	$\mathsf{fs}_3\ \mathsf{fz}_2$	$\ddots$
			$\mathsf{fs}_2\ (\mathsf{fs}_1\ \mathsf{fz}_0)$	$\mathsf{fs}_3\ (\mathsf{fs}_2\ \mathsf{fz}_1)$	$\ddots$
				$\mathsf{fs}_3\ (\mathsf{fs}_2\ (\mathsf{fs}_1\ \mathsf{fz}_0))$	$\ddots$
					$\ddots$

As you can see, each non-empty column contains a copy of the previous column, embedded by fs, together with a 'new' fz at the start.

We implement the function **proj** which safely accesses the ith element of a vector by structural recursion over the vector. We analyse the index given as an element of $\mathsf{Fin}\,n$ and since both constructors of $\mathsf{Fin}\,n$ produce elements in $\mathsf{Fin}\,(\mathsf{suc}\,m)$ the subsequent analysis of the vector needs only a vcons case.

$$\underline{\text{let}}\ \ \frac{xs : \mathsf{Vec}\,n\,X \quad i : \mathsf{Fin}\,n}{\mathbf{proj}\,xs\,i : X} \quad \begin{array}{l} \mathbf{proj}\,xs\,i \Leftarrow \underline{\text{rec}}\,xs \\ \ \ \mathbf{proj}\,xs\,i \Leftarrow \underline{\text{case}}\,i \\ \ \ \ \ \mathbf{proj}\,xs\,\mathsf{fz} \Leftarrow \underline{\text{case}}\,xs \\ \ \ \ \ \ \ \mathbf{proj}\,(\mathsf{vcons}\,x\,xs)\,\mathsf{fz} \Rightarrow x \\ \ \ \ \ \mathbf{proj}\,xs\,(\mathsf{fs}\,i) \Leftarrow \underline{\text{case}}\,xs \end{array}$$

$$\mathbf{proj}\ (\mathsf{vcons}\ x\ xs)\ (\mathsf{fs}\ i)\ \Rightarrow\ \mathbf{proj}\ xs\ i$$

Let's look more closely at what just happened. Here's **proj** again, but with the numeric indices shown as subscripts:

```
$\underline{\mathrm{let}}\ \dfrac{xs\ :\ \mathsf{Vec}\ n\ X \quad i\ :\ \mathsf{Fin}\ n}{\mathbf{proj}_n\ xs\ i\ :\ X}$

$\mathbf{proj}_n\ xs\ i\ \Leftarrow\ \underline{\mathrm{rec}}\ xs$
 $\mathbf{proj}_n\ xs\ i\ \Leftarrow\ \underline{\mathsf{case}}\ i$
  $\mathbf{proj}_{(\mathsf{suc}\ n)}\ xs\ (\mathsf{fz}_n)\ \Leftarrow\ \underline{\mathsf{case}}\ xs$
   $\mathbf{proj}_{(\mathsf{suc}\ n)}\ (\mathsf{vcons}_n\ x\ xs)\ (\mathsf{fz}_n)\ \Rightarrow\ x$
  $\mathbf{proj}_{(\mathsf{suc}\ n)}\ xs\ (\mathsf{fs}_n\ i)\ \Leftarrow\ \underline{\mathsf{case}}\ xs$
   $\mathbf{proj}_{(\mathsf{suc}\ n)}\ (\mathsf{vcons}_n\ x\ xs)\ (\mathsf{fs}_n\ i)\ \Rightarrow\ \mathbf{proj}_n\ xs\ i$
```

When we analyse i, we get patterns for i, but we learn more about n at the same time. Case analysis specialises the whole programming problem, propagating the consequences of inspecting one value for others related to it by type dependency. The extra requirements imposed on the construction of dependently typed data become extra guarantees when we take it apart.

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:

$$\underline{\text{let}}\ \frac{xys\ :\ \mathsf{Vec}\ n\ (\mathsf{Vec}\ m\ X)}{\textbf{transpose}\ xys\ :\ \mathsf{Vec}\ m\ (\mathsf{Vec}\ n\ X)}$$

Predefined types in Epigram

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 constructor for equality is $\mathsf{refl} : a = a$ in the special case that both sides of the equation compute to the same value. For example,

$$\mathsf{refl}\ :\ \textbf{plus}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero}))\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero})) = \mathsf{suc}\ (\mathsf{suc}\ (\mathsf{suc}\ (\mathsf{suc}\ \mathsf{zero})))$$

Epigram has dependent function types $\forall a : A \Rightarrow B$, where firstly $A : \star$ and secondly $B : \star$ under the assumption $a : A$. We retain the conventional $A \rightarrow B$ notation for function types where the latter assumption is not 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 abstractions 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 declare functions, e.g. the type of **vhead** can be written explicitly as $\forall\_m; \_Y \Rightarrow \mathsf{Vec}\,(\mathsf{suc}\,m)\ Y \rightarrow Y$.

Common datatypes in this paper

There are a few additional standard type constructors which we shall use in this paper: we define a type for disjoint union corresponding to `Either` in Haskell:

$$\underline{\text{data}}\ \frac{A, B\ :\ \star}{\mathsf{Plus}\ A\ B\ :\ \star}\quad \underline{\text{where}}\ \frac{a\ :\ A}{\mathsf{Inl}\ a\ :\ \mathsf{Plus}\ A\ B}\ \ \frac{b\ :\ B}{\mathsf{Inr}\ b\ :\ \mathsf{Plus}\ A\ B}$$

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

$$\underline{\text{data}}\ \frac{A, B\ :\ \star}{\mathsf{Times}\ A\ B}\quad \underline{\text{where}}\ \frac{a\ :\ A\quad b\ :\ B}{\mathsf{Pair}\ a\ b\ :\ \mathsf{Times}\ A\ B}$$

We also introduce the Σ-type, giving us *dependent* tupling: [2]

$$\underline{\text{data}}\ \frac{A\ :\ \star\quad B\ :\ A \rightarrow \star}{\mathsf{Sigma}\ A\ B\ :\ \star}\quad \underline{\text{where}}\ \frac{a\ :\ A\quad b\ :\ B\ a}{\mathsf{Tup}\ a\ b\ :\ \mathsf{Sigma}\ A\ B}$$

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

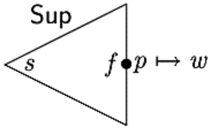
$$\underline{\text{let}}\ \frac{p : \mathsf{Sigma}\,A\,B}{\mathbf{fst}\,p\ :\ A} \qquad\qquad \underline{\text{let}}\ \frac{p : \mathsf{Sigma}\,A\,B}{\mathbf{snd}\,p\ :\ B\,(\mathbf{fst}\,p)}$$

W-types

Later in the paper, we shall also need a general-purpose inductive datatype, abstracting once and for all over *well-founded* tree-like data. Tree-like data is built from nodes. Each node carries some sort of data—its *shape*—usually, a tag indicating what sort of node it is, plus some appropriate labelling. In any case, the node shape determines what *positions* there might be for subtrees. This characterisation of well-founded data in terms of shapes and positions is presented via the W-type:

$$\underline{\text{data}}\ \frac{S : \star \quad P : S \to \star}{\mathsf{W}\,S\,P : \star} \quad \underline{\text{where}}\ \frac{s : S \quad f : P\,s \to \mathsf{W}\,S\,P}{\mathsf{Sup}\,s\,f\ :\ \mathsf{W}\,S\,P}$$

The constructor packs up a choice s of shape, together with a function f assigning subtrees to the positions appropriate to that shape. It is traditional to call the constructor Sup for 'supremum', as a node is the least thing bigger than its subtrees. We often illustrate this pattern—choice of shape, function from positions—as a *triangle* diagram. We write the shape s in the apex, and we think of the base as the corresponding set $P\ s$ of positions. The function f, part of the node hence inside the triangle, attaches subtrees w to positions p.



For example, the natural numbers have *two* node shapes, suc with one subtree and zero without. Correspondingly, we can use Bool for the shapes; the positions

[2] Some authors call this a 'dependent product', as it's the dependent version of Times. Other call it a 'dependent sum', as it's the infinitary analogue of Plus, and say 'dependent product' for functions, as these are the infinitary analogue of tuples. To avoid confusion, we prefer to talk of 'dependent function types' and 'dependent tuple types'.

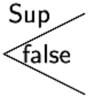
are given by the following type family which captures 'being true':

$$\underline{\text{data}}\ \frac{b\ :\ \mathsf{Bool}}{\mathsf{So}\ b\ :\ \star}\quad \underline{\text{where}}\ \frac{}{\mathsf{oh}\ :\ \mathsf{So\ true}}$$

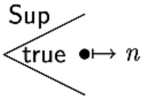
$$\underline{\text{let}}\ \frac{p\ :\ \mathsf{So\ false}}{\mathbf{notSo}\ p\ :\ X}\quad \mathbf{notSo}\ p\ \Leftarrow\ \underline{\text{case}}\ p$$

We may now *define* the natural numbers as a W-type:

$$\underline{\text{let}}\ \frac{}{\mathbf{wNat}\ :\ \star}$$
$$\mathbf{wNat}\ \Rightarrow\ \mathsf{W\ Bool\ So}$$

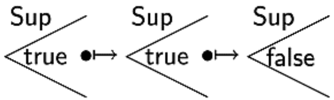


$$\underline{\text{let}}\ \frac{}{\mathbf{wZero}\ :\ \mathbf{wNat}}$$
$$\mathbf{wZero}\ \Rightarrow\ \mathsf{Sup\ false}\ \mathbf{notSo}$$



$$\underline{\text{let}}\ \frac{n\ :\ \mathbf{wNat}}{\mathbf{wSuc}\ n\ :\ \mathbf{wNat}}$$
$$\mathbf{wSuc}\ n\ \Rightarrow\ \mathsf{Sup\ true}\ (\lambda p \Rightarrow n)$$

The pictures show us the components we can plug together to make numbers. ‘Two’ looks like this:



Lots of our favourite inductive datatypes fit this pattern. Another key example is the type of *finitely branching trees*, W Nat Fin, where the shape of each node is its arity.

Exercise 3. Construct the W-type corresponding to Haskell’s

```
data BTree l n = Leaf l | Node (BTree l n) n (BTree l n)
```

Later, we shall exploit the W-type analysis of data in terms of shapes and positions to characterise *containers*, more generally.

Views in Epigram

Once we have the idea of programming by stepwise refinement of problems, it becomes interesting to ask ‘What refinements can we have? Are we restricted to case and rec?’. The *eliminators* which we use to refine programming problems are first-class Epigram values, so it is entirely possible to implement your own.

This flexibility is central to the design of Epigram [32], and it gives rise to a novel and useful programming technique inspired by Wadler's notion of 'views' [39].

We can specify a new way to analyse data, just by indexing a datatype family with it. Consider pairs of Boolean values, for example: regardless of whether they are true or false, it is surely the case that either they coincide, or the second is the negation of the first. We can express this idea by defining a datatype family—the *view relation*—with one constructor for each case of our desired analysis:

$$\underline{\text{data}}\ \frac{b, a\ :\ \mathsf{Bool}}{\mathsf{EqOrNot}\ b\ a\ :\ \star}\quad \underline{\text{where}}\ \frac{}{\mathsf{same}\ :\ \mathsf{EqOrNot}\ b\ b}\quad \frac{}{\mathsf{diff}\ :\ \mathsf{EqOrNot}\ b\ (\mathbf{not}\ b)}$$

If we had an element p of $\mathsf{EqOrNot}\,b\,a$, then case analysis for p as same or diff tells us *ipso facto* whether a is b or $\mathbf{not}\ b$. Let us make sure that we can always have such a p by writing a *covering function* to show that the view relation always holds.

$$\underline{\text{let}}\ \frac{}{\mathbf{eqOrNot}\ b\ a\ :\ \mathsf{EqOrNot}\ b\ a}\quad \begin{array}{l} \mathbf{eqOrNot}\ b\ a\ \Leftarrow\ \underline{\text{case}}\ b \\ \quad \mathbf{eqOrNot}\ \mathsf{true}\ a\ \Leftarrow\ \underline{\text{case}}\ a \\ \quad\quad \mathbf{eqOrNot}\ \mathsf{true}\ \mathsf{true}\ \Rightarrow\ \mathsf{same} \end{array}$$

$$
\begin{aligned}
&\mathbf{eqOrNot}\ \mathsf{true}\ \mathsf{false} \Rightarrow \mathsf{diff}\\
&\mathbf{eqOrNot}\ \mathsf{false}\ a \Leftarrow \underline{\mathrm{case}}\ a\\
&\quad \mathbf{eqOrNot}\ \mathsf{false}\ \mathsf{true} \Rightarrow \mathsf{diff}\\
&\quad \mathbf{eqOrNot}\ \mathsf{false}\ \mathsf{false} \Rightarrow \mathsf{same}
\end{aligned}
$$

How do we use this information in practice? Epigram has syntactic support for case analysis derived in this style: if p is a proof that the view relation holds, $\underline{\mathrm{view}}\ p$ is the eliminator which delivers the corresponding analysis of its indices. For example, we may now write

$$
\underline{\mathrm{let}}\ \frac{x, y\ :\ \mathsf{Bool}}{\mathbf{xor}\ x\ y\ :\ \mathsf{Bool}} \qquad
\begin{aligned}
&\mathbf{xor}\ x\ y \Leftarrow \underline{\mathrm{view}}\,(\mathbf{eqOrNot}\ x\ y)\\
&\quad \mathbf{xor}\ x\ x \Rightarrow \mathsf{false}\\
&\quad \mathbf{xor}\ x\,(\mathbf{not}\ x) \Rightarrow \mathsf{true}
\end{aligned}
$$

There is no need to be alarmed at the appearance of repeated pattern variables and even *defined* functions on the left-hand side. Operationally, this program computes an element of EqOrNot $x\,y$, then forks control accordingly as it is same or diff. What you see on the left comes from the specialisation of y which accompanies that constructor analysis.

Views are important tools for testing data on which types depend. Our EqOrNot construction may be more complex than the ordinary Boolean 'equivalence' test, but it is also more revealing. The view actually shows the typechecker what the equality test learns.

To see this in action, consider implementing an equality test for **wNat**. At each node, we shall need to compare shapes, and if they coincide, check equality at each position. How do we know that the position sets must be identical whenever the shapes coincide? Our view makes the connection.

$$\underline{\text{let}}\ \frac{x, y\ :\ \mathbf{wNat}}{\mathbf{wNatEq}\ x\ y\ :\ \mathsf{Bool}}$$

$$\begin{array}{l}
\mathbf{wNatEq}\ x\ y\ \Leftarrow\ \underline{\text{rec}}\ x \\
\quad \mathbf{wNatEq}\ x\ y\ \Leftarrow\ \underline{\text{case}}\ x \\
\quad\quad \mathbf{wNatEq}\ (\mathsf{Sup}\ b\ f)\ y\ \Leftarrow\ \underline{\text{case}}\ y \\
\quad\quad\quad \mathbf{wNatEq}\ (\mathsf{Sup}\ b\ f)\ (\mathsf{Sup}\ a\ g)\ \Leftarrow\ \underline{\text{view}}\ (\mathbf{eqOrNot}\ b\ a) \\
\quad\quad\quad\quad \mathbf{wNatEq}\ (\mathsf{Sup}\ b\ f)\ (\mathsf{Sup}\ b\ g)\ \Leftarrow\ \underline{\text{case}}\ b \\
\quad\quad\quad\quad\quad \mathbf{wNatEq}\ (\mathsf{Sup}\ \mathsf{true}\ f)\ (\mathsf{Sup}\ \mathsf{true}\ g)\ \Rightarrow\ \mathbf{wNatEq}\ (f\ \mathsf{oh})\ (g\ \mathsf{oh}) \\
\quad\quad\quad\quad\quad \mathbf{wNatEq}\ (\mathsf{Sup}\ \mathsf{false}\ f)\ (\mathsf{Sup}\ \mathsf{false}\ g)\ \Rightarrow\ \mathsf{true} \\
\quad\quad\quad\quad \mathbf{wNatEq}\ (\mathsf{Sup}\ b\ f)\ (\mathsf{Sup}\ (\mathbf{not}\ b)\ g))\ \Rightarrow\ \mathsf{false}
\end{array}$$

3 The Universe of Finite Types

We have already implicitly introduced our 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 name is given by the family Fin given in the previous section, which assigns to any n : 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 : 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 analyse 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 : Nat leading to the following signatures:

$$\underline{\text{let}}\ \frac{m, n : \mathsf{Nat} \quad i : \mathsf{Fin}\ m}{\mathbf{finl}\ m\ n\ i : \mathsf{Fin}\,(\mathbf{plus}\ m\ n)} \qquad \underline{\text{let}}\ \frac{m, n : \mathsf{Nat} \quad j : \mathsf{Fin}\ n}{\mathbf{finr}\ m\ n\ j : \mathsf{Fin}\,(\mathbf{plus}\ m\ n)}$$

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

```
$\mathbf{finl}\ m\ n\ i \Leftarrow \underline{\text{rec}}\ m$
 $\mathbf{finl}\ m\ n\ i \Leftarrow \underline{\text{case}}\ i$
  $\mathbf{finl}\,(\mathsf{suc}\ m)\ n\ \mathsf{fz} \Rightarrow \mathsf{fz}$
  $\mathbf{finl}\,(\mathsf{suc}\ m)\ n\ (\mathsf{fs}\ i) \Rightarrow \mathsf{fs}\,(\mathbf{finl}\ m\ n\ i)$
```

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

```
$\mathbf{finr}\ m\ n\ j \Leftarrow \underline{\text{rec}}\ m$
 $\mathbf{finr}\ m\ n\ j \Leftarrow \underline{\text{case}}\ m$
  $\mathbf{finr}\ \mathsf{zero}\ n\ j \Rightarrow j$
  $\mathbf{finr}\,(\mathsf{suc}\ m)\ n\ j \Rightarrow \mathsf{fs}\,(\mathbf{finr}\ m\ n\ j)$
```

We analyse the type name $m : \mathsf{Nat}$ to apply m successor operations fs to lift $\mathsf{Fin}\ n$ into $\mathsf{Fin}\,(\mathbf{plus}\ m\ n)$. 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 $\mathsf{Fin}\,(\mathbf{plus}\,m\;n)$? One way to answer this question is to provide an eliminator in form of a case-function:

$$\underline{\text{let}}\;\; \frac{s\,:\,\mathsf{Fin}\,(\mathbf{plus}\,m\;n) \quad l\,:\,\mathsf{Fin}\,m \to X \quad r\,:\,\mathsf{Fin}\,n \to X}{\mathbf{fcase}_{m\,n}\,s\;l\;r\,:\,X}$$

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:

$$\underline{\text{data}}\;\; \frac{i\,:\,\mathsf{Fin}\,(\mathbf{plus}\,m\;n)}{\mathsf{FinPlus}\,m\;n\;i}\;\; \underline{\text{where}}$$

$$\frac{i\,:\,\mathsf{Fin}\,m}{\mathsf{isfinl}\,i\,:\,\mathsf{FinPlus}\,m\;n\;(\mathbf{finl}\;m\;n\;i)} \quad \frac{j\,:\,\mathsf{Fin}\,n}{\mathsf{isfinr}\,j\,:\,\mathsf{FinPlus}\,m\;n\;(\mathbf{finr}\;m\;n\;j)}$$

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

$$\underline{\text{let}}\;\; \frac{}{\mathbf{finPlus}\,m\;n\;i\,:\,\mathsf{FinPlus}\,m\;n\;i}$$

$$
\begin{array}{l}
\mathbf{finPlus}\, m\ n\ i \ \Leftarrow\ \underline{\mathsf{rec}}\, m \\
\quad \mathbf{finPlus}\, m\ n\ i \ \Leftarrow\ \underline{\mathsf{case}}\, m \\
\quad\quad \mathbf{finPlus}\,\mathsf{zero}\ n\ i \ \Rightarrow\ \mathsf{isfinr}\ i \\
\quad\quad \mathbf{finPlus}\,(\mathsf{suc}\, m)\ n\ i \ \Leftarrow\ \underline{\mathsf{case}}\ i \\
\quad\quad\quad \mathbf{finPlus}\,(\mathsf{suc}\, m)\ n\ \mathsf{fz} \ \Rightarrow\ \mathsf{isfinl}\ \mathsf{fz} \\
\quad\quad\quad \mathbf{finPlus}\,(\mathsf{suc}\, m)\ n\ (\mathsf{fs}\, i) \ \Leftarrow\ \underline{\mathsf{view}}\,\mathbf{finPlus}\, m\ n\ i \\
\quad\quad\quad\quad \mathbf{finPlus}\,(\mathsf{suc}\, m)\ n\ (\mathsf{fs}\,(\mathbf{finl}\ m\ n\ i)) \ \Rightarrow\ \mathsf{isfinl}\,(\mathsf{fs}\, i) \\
\quad\quad\quad\quad \mathbf{finPlus}\,(\mathsf{suc}\, m)\ n\ (\mathsf{fs}\,(\mathbf{finr}\ m\ n\ j)) \ \Rightarrow\ \mathsf{isfinr}\ j
\end{array}
$$

We can now use <u>view</u> to do pattern matching over Fin (**plus** $m\ n$), e.g. to implement **fcase**:

$$
\begin{array}{l}
\mathbf{fcase}_{m\,n}\ s\ l\ r \ \Leftarrow\ \underline{\mathsf{view}}\,\mathbf{finPlus}\, m\ n\ s \\
\quad \mathbf{fcase}_{m\,n}\,(\mathbf{finl}\ m\ n\ i)\ l\ r \ \Rightarrow\ l\ i \\
\quad \mathbf{fcase}_{m\,n}\,(\mathbf{finr}\ m\ n\ j)\ l\ r \ \Rightarrow\ r\ j
\end{array}
$$

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:

$$\underline{\text{let}}\ \frac{i\ :\ \mathsf{Fin}\,m \quad j\ :\ \mathsf{Fin}\,n}{\mathbf{fpair}\,m\ n\ i\ j\ :\ \mathsf{Fin}\,(\mathbf{times}\,m\ n)}$$

The intuitive idea is to arrange the elements of $\mathsf{Fin}\,(\mathbf{times}\,m\ n)$ as a rectangle and assign to **pair** $i\,j$ the jth column in the ith row. This is realised by the following primitive recursive function which uses the previously defined constructors for coproducts, since our products are merely iterated coproducts:

```
$\mathbf{fpair}\,m\ n\ i\ j \ \Leftarrow\ \underline{\text{rec}}\ i$
 $\mathbf{fpair}\,m\ n\ i\ j \ \Leftarrow\ \underline{\text{case}}\ i$
  $\mathbf{fpair}\,(\mathsf{suc}\,m)\ n\ \mathsf{fz}\ j \ \Rightarrow\ \mathbf{finl}\,n\,(\mathbf{times}\,m\ n)\ j$
  $\mathbf{fpair}\,(\mathsf{suc}\,m)\ n\ (\mathsf{fs}\,i)\ j \ \Rightarrow\ \mathbf{finr}\,n\,(\mathbf{times}\,m\ n)\,(\mathbf{fpair}\,m\ n\ i\ j)$
```

Indeed the **pair** $m\,n\,i\,j$ 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:

$$\underline{\text{data}}\ \frac{i\ :\ \mathsf{Fin}\,(\mathbf{times}\,m\ n)}{\mathsf{FinTimes}\,m\ n\ i\ :\ \star}$$

$$\underline{\text{where}}\ \frac{i : \mathsf{Fin}\,m \quad j : \mathsf{Fin}\,n}{\mathsf{isfpair}\,i\,j : \mathsf{FinTimes}\,m\,n\,(\mathbf{fpair}\,m\,n\,i\,j)}$$

As before we show that this view is exhaustive:

$$\underline{\text{let}}\ \frac{}{\mathbf{finTimes}\,m\,n\,i : \mathsf{FinTimes}\,m\,n\,i}$$

```
$\mathbf{finTimes}\,m\,n\,i \Leftarrow \underline{\text{rec}}\,m$
 $\mathbf{finTimes}\,m\,n\,i \Leftarrow \underline{\text{case}}\,m$
  $\mathbf{finTimes}\,\mathsf{zero}\,n\,i \Leftarrow \underline{\text{case}}\,i$
  $\mathbf{finTimes}\,(\mathsf{suc}\,m)\,n\,i \Leftarrow \underline{\text{view}}\,\mathbf{finPlus}\,n\,(\mathbf{times}\,m\,n)\,i$
   $\mathbf{finTimes}\,(\mathsf{suc}\,m)\,n\,(\mathbf{finl}\,n\,(\mathbf{times}\,m\,n)\,i) \Rightarrow \mathsf{isfpair}\,\mathsf{fz}\,i$
   $\mathbf{finTimes}\,(\mathsf{suc}\,m)\,n\,(\mathbf{finr}\,n\,(\mathbf{times}\,m\,n)\,j) \Leftarrow \underline{\text{view}}\,\mathbf{finTimes}\,m\,n\,j$
    $\mathbf{finTimes}\,(\mathsf{suc}\,m)\,n\,(\mathbf{finr}\,n\,(\mathbf{times}\,m\,n)\,(\mathbf{fpair}\,m\,n\,i\,j))$
          $\Rightarrow \mathsf{isfpair}\,(\mathsf{fs}\,i)\,j$
```

Note that we are using the previously defined **FinPlus** view to analyse the iterated coproducts. We can use both derived pattern matching principles to

show that products distribute over coproducts

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

```
dist $m\ n\ o\ x$ ⇐ view finTimes $m$ (plus $n\ o$) $x$
 dist $m\ n\ o$ (fpair $m$ (plus $n\ o$) $i\ j$) ⇐ view finPlus $n\ o\ j$
  dist $m\ n\ o$ (fpair $m$ (plus $n\ o$) $i$ (finl $n\ o\ j$))
           ⇒ finl (times $m\ n$) (times $m\ o$) (fpair $m\ n\ i\ j$)
  dist $m\ n\ o$ (fpair $m$ (plus $n\ o$) $i$ (finr $n\ o\ j$))
           ⇒ finr (times $m\ n$) (times $m\ o$) (fpair $m\ o\ i\ j$)
```

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

Exercise 4. Define exponentials (i.e. function types) by implementing a function to represent the name of a function type:

$$\underline{\text{let}}\ \frac{m, n\ :\ \mathsf{Nat}}{\mathbf{exp}\,m\ n\ :\ \mathsf{Nat}}$$

and a constructor corresponding to lambda abstraction:

$$\underline{\text{let}}\ \frac{f\ :\ \mathsf{Fin}\ m\ \rightarrow\ \mathsf{Fin}\ n}{\mathbf{flam}\ m\ n\ f\ :\ \mathsf{Fin}\ (\mathbf{exp}\ m\ n)}$$

Unlike in the previous cases we cannot implement a pattern matching principle due to the lack of extensionality in Epigram's type system.[3]

However, we can define an application operator:

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

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 [13] we can see that the functions $\lambda f : \mathsf{Bool} \rightarrow \mathsf{Bool} \Rightarrow f$ and $\lambda f : \mathsf{Bool} \rightarrow \mathsf{Bool}; x : \mathsf{Bool} \Rightarrow f\ (f\ (f\ x))$ are extensionally equal by encoding them

[3] We cannot show that two functions are equal if they are pointwise equal. As a consequence we cannot show for example that there are exactly 4 functions of type

$\mathsf{Bool} \to \mathsf{Bool}$ which would be necessary if we want to establish a case analysis principle for finite types. Our ongoing work on *Observational Type Theory* [10] will address this issue.

using the combinators defined in the previous section and observing that they compute the same element in $\mathsf{Fin}\,256$.[4]

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 $0, +, 1, \times$:

$$\underline{\text{data}}\ \frac{}{\mathsf{Ufin} : \star}\ \underline{\text{where}}\ \frac{}{\text{`}0\text{'} : \mathsf{Ufin}}\quad \frac{a, b : \mathsf{Ufin}}{\text{`plus'}\ a\ b : \mathsf{Ufin}}$$

$$\frac{}{\text{`}1\text{'} : \mathsf{Ufin}}\quad \frac{a, b : \mathsf{Ufin}}{\text{`times'}\ a\ b : \mathsf{Ufin}}$$

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

We define the family of elements Elfin inductively:

$$\underline{\text{data}}\ \frac{a\ :\ \mathsf{Ufin}}{\mathsf{Elfin}\ a\ :\ \star}$$

$$\underline{\text{where}}\ \frac{b, a\ :\ \mathsf{Ufin}\quad x\ :\ \mathsf{Elfin}\ a}{\mathsf{inl}\ x\ :\ \mathsf{Elfin}\ (\text{`}\mathsf{plus}\text{'}\ a\ b)}\quad \frac{a, b\ :\ \mathsf{Ufin}\quad y\ :\ \mathsf{Elfin}\ b}{\mathsf{inr}\ y\ :\ \mathsf{Elfin}\ (\text{`}\mathsf{plus}\text{'}\ a\ b)}$$

$$\frac{}{\mathsf{void}\ :\ \mathsf{Elfin}\ \text{`}1\text{'}}\quad \frac{x\ :\ \mathsf{Elfin}\ a\quad y\ :\ \mathsf{Elfin}\ b}{\mathsf{pair}\ x\ y\ :\ \mathsf{Elfin}\ (\text{`}\mathsf{times}\text{'}\ a\ b)}$$

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:

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

```
dist x ⇐ case x
  dist (pair x y) ⇐ case y
    dist (pair x (inl y)) ⇒ inl (pair x y)
    dist (pair x (inr z)) ⇒ inr (pair x z)
```

[4] We don't recommend trying this with the current implementation of Epigram.

4.1 Enumerating finite types

So far we haven't defined any proper generic operations, i.e. an operation which works on all types of a universe by inspecting the name. A generic 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:

$$\underline{\text{data}}\ \frac{A\ :\ \star}{\mathsf{ET}\ A\ :\ \star}$$

$$\underline{\text{where}}\ \frac{a\ :\ A}{\mathsf{V}\ a\ :\ \mathsf{ET}\ A}\quad \frac{l, r\ :\ \mathsf{ET}\ A}{\mathsf{C}\ l\ r\ :\ \mathsf{ET}\ A}\quad \frac{}{\mathsf{E}\ :\ \mathsf{ET}\ A}$$

Our generic enumeration function has the following type:

$$\underline{\mathrm{let}}\ \frac{a\ :\ \mathsf{Ufin}}{\mathbf{enum}\ a\ :\ \mathsf{ET}\ (\mathsf{Elfin}\ a)}$$

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

$$\underline{\mathrm{let}}\ \frac{a\ :\ A}{\mathbf{returnET}\ a\ :\ \mathsf{ET}\ A} \quad \mathbf{returnET}\ a \Rightarrow \mathsf{V}\ a$$

$$\underline{\mathrm{let}}\ \frac{t\ :\ \mathsf{ET}\ A \quad f\ :\ A \to \mathsf{ET}\ B}{\mathbf{bindET}\ t\ f\ :\ \mathsf{ET}\ B}$$

$$\begin{array}{l}
\mathbf{bindET}\ t\ f \Leftarrow \underline{\mathrm{rec}}\ t \\
\quad \mathbf{bindET}\ t\ f \Leftarrow \underline{\mathrm{case}}\ t \\
\quad\quad \mathbf{bindET}\ (\mathsf{V}\ a)\ f \Rightarrow f\ a \\
\quad\quad \mathbf{bindET}\ (\mathsf{C}\ l\ r)\ f \Rightarrow \mathsf{C}\ (\mathbf{bindET}\ l\ f)\ (\mathbf{bindET}\ r\ f) \\
\quad\quad \mathbf{bindET}\ \mathsf{E}\ f \Rightarrow \mathsf{E}
\end{array}$$

Consequently, ET is also functorial:

$$\underline{\mathrm{let}}\ \frac{f\ :\ A \to B \quad t\ :\ \mathsf{ET}\ A}{\mathbf{mapET}\ f\ t\ :\ \mathsf{ET}\ B}$$

$$\textbf{mapET}\; f\; t \;\Rightarrow\; \textbf{bindET}\; t\; (\lambda x \Rightarrow \textbf{returnET}\; (f\; x))$$

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

```
enum a ⇐ rec a
 enum a ⇐ case a
  enum '0' ⇒ E
  enum ('plus' a b) ⇒ C (mapET inl (enum a)) (mapET inr (enum b))
  enum '1' ⇒ V void
  enum ('times' a b)
     ⇒ bindET (enum a) (λx ⇒ mapET (λy ⇒ pair x y) (enum b))
```

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

Context-free types

By context-free types[5] we mean types which can be constructed by combining the polynomial operators from the previous section $(0, +, \times, 1)$ with an operator μ to construct inductive types, or in categorical terms initial algebras. We have already seen some examples of context-free types, for instance Nat can be expressed as $\mathsf{Nat} = \mu X.1+X$ and List which can be encoded: $\mathsf{List}\, A = \mu X.1+A\times X$.

Other examples we will use include binary trees with data at the nodes which can be given by the expression $\mathsf{Tree}\,A = \mu X.1 + (X \times A \times X)$. Finally rose trees which are given by the code $\mathsf{RT}\,A = \mu X.\mathsf{List}\,(A \times X) = \mu X.\mu Y.1 + (A \times X) \times Y$. 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 address is how to represent variables. We use a deBruijn style representation of variables, this seems to be essential since we give are going to represent types an inductive family, using names would cause a considerable overhead and also would mean that we have to deal with issues like alpha conversion. Moreover, we are free to implement a function which translates a name carrying type into our internal deBruijn representation. This choice is a variation on the approach taken by McBride[29] when he first gave an inductive characterisation of these types. The names of context free types becomes a family indexed by the number of free variables:

$$\underline{\text{data}}\ \frac{n\ :\ \mathsf{Nat}}{\mathsf{Ucf}\ n\ :\ \star}\ \underline{\text{where}}$$

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

$$\frac{}{\text{‘0’} : \mathsf{Ucf}\ n} \quad \frac{a, b : \mathsf{Ucf}\ n}{\text{‘plus’}\ a\ b : \mathsf{Ucf}\ n}$$

$$\frac{}{\text{‘1’} : \mathsf{Ucf}\ n} \quad \frac{a, b : \mathsf{Ucf}\ n}{\text{‘times’}\ a\ b : \mathsf{Ucf}\ n}$$

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

$$\frac{}{\mathsf{vl} : \mathsf{Ucf}\ (\mathsf{suc}\ n)} \quad \frac{a : \mathsf{Ucf}\ n}{\mathsf{wk}\ a : \mathsf{Ucf}\ (\mathsf{suc}\ n)}$$

[5] We previously used the term ‘regular *tree* types’ [35] in a vain attempt to avoid confusion with regular *expressions*.

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

$$\frac{x : \mathsf{Fin}\ n}{\mathsf{var}\ x : \mathsf{Ucf}\ n}$$

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

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

$$\frac{f : \mathsf{Ucf}\,(\mathsf{suc}\,n) \quad a : \mathsf{Ucf}\,n}{\mathsf{def}\,f\,a : \mathsf{Ucf}\,n}$$

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:

$$\frac{f : \mathsf{Ucf}\,(\mathsf{suc}\,n)}{\text{`}\mathsf{mu}\text{'}\,f : \mathsf{Ucf}\,n}$$

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

$$\underline{\text{let}}\ \frac{}{\mathbf{nat}\ :\ \mathsf{Ucf}\ n}\quad \mathbf{nat}\ \Rightarrow\ \text{`}\mathsf{mu}\text{'}\ (\text{`}\mathsf{plus}\text{'}\ \text{`}\mathsf{1}\text{'}\ \mathsf{vl})$$

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

$$\underline{\text{let}}\ \frac{}{\mathbf{tree}\ :\ \mathsf{Ucf}\ (\mathsf{suc}\ n)}$$

$$\mathbf{tree}\ \Rightarrow\ \text{`}\mathsf{mu}\text{'}\ (\text{`}\mathsf{plus}\text{'}\ \text{`}\mathsf{1}\text{'}\ (\text{`}\mathsf{times}\text{'}\ \mathsf{vl}\ (\text{`}\mathsf{times}\text{'}\ (\mathsf{wk}\ \mathsf{vl})\ \mathsf{vl})))$$

$$\underline{\text{let}}\ \frac{}{\mathbf{rt}\ :\ \mathsf{Ucf}\ (\mathsf{suc}\ n)}\quad \mathbf{rt}\ \Rightarrow\ \text{`}\mathsf{mu}\text{'}\ (\mathsf{def}\ \mathbf{list}\ (\text{`}\mathsf{times}\text{'}\ (\mathsf{wk}\ \mathsf{vl})\ \mathsf{vl}))$$

While **nat** and **rt** are closed types and hence inhabit $\mathsf{Ucf}\ n$ for any $n : \mathsf{Nat}$, **list** is parametrized by the last type variable and hence inhabits $\mathsf{Ucf}\ (\mathsf{suc}\ n)$. 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. $\mathsf{let}\ \mathbf{list}\ \mathbf{nat}\ :\ \mathsf{Ucf}\ n$ 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 signature[6] :

$$\frac{a \,:\, \mathsf{Ucf}\; n \quad Xs \,:\, \mathsf{Vec}\; n \star}{\mathsf{Elcf}\; a\; Xs \,:\, \star}$$

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. Let's see why: A reasonable attempt is to say:

$$\frac{x \,:\, \mathsf{Elcf}\; f\; (\mathsf{vcons}\; (\mathsf{Elcf}\; (\text{`}\mathsf{mu}\text{'}\; f)\; Xs)\; Xs)}{\mathsf{in}\; x \,:\, \mathsf{Elcf}\; (\text{`}\mathsf{mu}\text{'}\; f)\; Xs}$$

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.

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:

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

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

$$\underline{\text{data}}\quad \frac{a : \mathsf{Ucf}\ n \quad as : \mathsf{Tel}\ n}{\mathsf{Elcf}\ a\ as : \star}$$

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

$$\frac{b, a : \mathsf{Ucf} \quad x : \mathsf{Elcf}\ a\ as}{\mathsf{inl}\ x : \mathsf{Elcf}\ (\text{`}\mathsf{plus}\text{'}\ a\ b)\ as} \qquad \frac{a, b : \mathsf{Ucf} \quad y : \mathsf{Elcf}\ b\ as}{\mathsf{inr}\ y : \mathsf{Elcf}\ (\text{`}\mathsf{plus}\text{'}\ a\ b)\ as}$$

$$\frac{}{\mathsf{void} : \mathsf{Elcf}\ \text{`}1\text{'}\ as} \qquad \frac{x : \mathsf{Elcf}\ a \quad y : \mathsf{Elcf}\ b\ as}{\mathsf{pair}\ x\ y : \mathsf{Elcf}\ (\text{`}\mathsf{times}\text{'}\ a\ b)\ as}$$

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

$$\frac{x : \mathsf{Elcf}\ a\ as}{\mathsf{top}\ x : \mathsf{Elcf}\ \mathsf{vl}\ (\mathsf{tcons}\ a\ as)}$$

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

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

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

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

$$\frac{x\ :\ \mathsf{Elcf}\ f\ (\mathsf{tcons}\ a\ as)}{\mathsf{push}\ x\ :\ \mathsf{Elcf}\ (\mathsf{def}\ f\ a)\ as}$$

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:

$$\frac{x\ :\ \mathsf{Elcf}\ f\ (\mathsf{tcons}\ as\ (\text{‘}\mathsf{mu}\text{’}\ f))}{\mathsf{in}\ x\ :\ \mathsf{Elcf}\ (\text{‘}\mathsf{mu}\text{’}\ f)\ as}$$

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:

$$\underline{\mathrm{let}}\ \frac{}{\text{‘}\mathbf{zero}\text{’} : \mathsf{Elcf}\,\mathbf{nat}\,as} \quad \text{‘}\mathbf{zero}\text{’} \Rightarrow \mathsf{in}\,(\mathsf{inl}\,\mathsf{void})$$

$$\underline{\mathrm{let}}\ \frac{n : \mathsf{Elcf}\,\mathbf{nat}\,as}{\text{‘}\mathbf{suc}\text{’}\,n : \mathsf{Elcf}\,\mathbf{nat}\,as} \quad \text{‘}\mathbf{suc}\text{’}\,n \Rightarrow \mathsf{in}\,(\mathsf{inr}\,(\mathsf{top}\,n))$$

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

$$\underline{\mathrm{data}}\ \frac{n : \mathsf{Elcf}\,\mathbf{nat}\,as}{\mathsf{NatView}\,n : \star}$$

$$\underline{\mathrm{where}}\ \frac{}{\mathsf{isZ} : \mathsf{NatView}\,\text{‘}\mathbf{zero}\text{’}} \quad \frac{n : \mathsf{Elcf}\,\mathbf{nat}\,as}{\mathsf{isS}\,n : \mathsf{NatView}\,(\text{‘}\mathbf{suc}\text{’}\,n)}$$

We show that NatView is exhaustive:

$$\underline{\mathrm{let}}\ \frac{n : \mathsf{Elcf}\,\mathbf{nat}\,as}{\mathbf{natView}\,n : \mathsf{NatView}\,n}$$

$$\begin{array}{l}
\mathbf{natView}\,n \Leftarrow \underline{\mathsf{case}}\,n \\
\quad \mathbf{natView}\,(\mathsf{in}\,x) \Leftarrow \underline{\mathsf{case}}\,x \\
\qquad \mathbf{natView}\,(\mathsf{in}\,(\mathsf{inl}\,x)) \Leftarrow \underline{\mathsf{case}}\,x
\end{array}$$

$$\begin{array}{l} \textbf{natView}\,(\mathsf{in}\,(\mathsf{inl}\,\mathsf{void})) \;\Rightarrow\; \mathsf{isZ} \\ \textbf{natView}\,(\mathsf{in}\,(\mathsf{inr}\,y)) \;\Leftarrow\; \underline{\mathsf{case}}\,y \\ \quad \textbf{natView}\,(\mathsf{in}\,(\mathsf{inr}\,(\mathsf{top}\,n'))) \;\Rightarrow\; \mathsf{isS}\,n' \end{array}$$

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

$$\underline{\mathsf{let}}\;\; \frac{m, n \;:\; \mathsf{Elcf}\,\textbf{nat}\,as}{\text{‘}\textbf{add}\text{’}\,m\,n \;:\; \mathsf{Elcf}\,\textbf{nat}\,as}$$

$$\begin{array}{l} \text{‘}\textbf{add}\text{’}\,m\,n \;\Leftarrow\; \underline{\mathsf{rec}}\,m \\ \quad \text{‘}\textbf{add}\text{’}\,m\,n \;\Leftarrow\; \underline{\mathsf{view}}\,\textbf{natView}\,m \\ \quad\quad \text{‘}\textbf{add}\text{’}\,(\mathsf{in}\,(\mathsf{inl}\,\mathsf{void}))\,n \;\Rightarrow\; n \\ \quad\quad \text{‘}\textbf{add}\text{’}\,(\mathsf{in}\,(\mathsf{inr}\,(\mathsf{top}\,m')))\,n \;\Rightarrow\; \mathsf{in}\,(\mathsf{inr}\,(\mathsf{top}\,(\text{‘}\textbf{add}\text{’}\,m'\,n))) \end{array}$$

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 build in a reflection mechanism into Epigram which allows us to access the names of the top level universe as data.

Exercise 6. Define a pattern matching principle for lists, that is first define 'constructors'

$$\underline{\text{let}}\quad \frac{}{\text{‘}\mathbf{nil}\text{’} : \mathsf{Elcf}\ \mathbf{list}\ X}$$

$$\underline{\text{let}}\quad \frac{x : \mathsf{Elcf}\ a\ as \quad xs : \mathsf{Elcf}\ \mathbf{list}\ (\mathsf{tcons}\ a\ as)}{\text{‘}\mathbf{cons}\text{’}\ x\ xs : \mathsf{Elcf}\ \mathbf{list}\ (\mathsf{tcons}\ a\ as)}$$

and then create an appropriate view, following the nat example. Consider how to do this for 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:

$$\underline{\text{let}}\ \frac{x, y\ :\ \mathsf{Elcf}\ a\ as}{\mathbf{geq}\ x\ y\ :\ \mathsf{Bool}}$$

```
geq x y ⇐ rec x
 geq x y ⇐ case x
  geq (inl xa) y ⇐ case y
   geq (inl xa) (inl ya) ⇒ geq xa ya
   geq (inl xa) (inr yb) ⇒ false
  geq (inr xb) y ⇐ case y
   geq (inr xb) (inl ya) ⇒ false
   geq (inr a) (inr yb) ⇒ geq xb yb
  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.

Note that 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:

$$\underline{\text{let}}\ \ \frac{A\ :\ \star}{\mathbf{Not}\ A\ :\ \star}\quad \mathbf{Not}\ A\ \Rightarrow\ A \rightarrow \mathsf{Zero}$$

$$\underline{\text{data}}\ \ \frac{A\ :\ \star}{\mathsf{Dec}\ A\ :\ \star}\ \ \underline{\text{where}}\ \ \frac{a\ :\ A}{\mathsf{yes}\ a\ :\ \mathsf{Dec}\ A}\quad \frac{f\ :\ \mathsf{Not}\ A}{\mathsf{no}\ f\ :\ \mathsf{Dec}\ A}$$

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

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

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

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.

$$\underline{\text{data}}\ \frac{}{\mathsf{Ord}\ :\ \star}\ \underline{\text{where}}\ \frac{}{\mathsf{oz}\ :\ \mathsf{Ord}}\quad \frac{a\ :\ \mathsf{Ord}}{\mathsf{os}\, a\ :\ \mathsf{Ord}}\quad \frac{f\ :\ \mathsf{Nat} \to \mathsf{Ord}}{\mathsf{olim}\, f\ :\ \mathsf{Ord}}$$

We can embed the natural numbers into the ordinals:

$$\underline{\text{let}}\ \frac{n\ :\ \mathsf{Nat}}{\mathbf{o2n}\, n\ :\ \star}\quad \begin{array}{l} \mathbf{n2o}\, n \Leftarrow \underline{\text{rec}}\, n \\ \quad \mathbf{n2o}\, n \Leftarrow \underline{\text{case}}\, n \\ \quad \mathbf{n2o}\, \mathsf{zero} \Rightarrow \mathsf{oz} \\ \quad \mathbf{n2o}\, (\mathsf{suc}\, n) \Rightarrow \mathsf{os}\, (\mathbf{n2o}\, n) \end{array}$$

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

$$\underline{\text{let}}\ \frac{}{\mathbf{omega}\ :\ \mathsf{Ord}}\quad \mathbf{omega} \Rightarrow \mathsf{olim}\, \mathbf{n2o}$$

We can also do arithmetic on ordinals, using structural recursion we define addition[7] of ordinals:

$$\underline{\text{let}}\ \frac{a, b\ :\ \mathsf{Ord}}{\mathbf{oplus}\ a\ b\ :\ \mathsf{Ord}}$$

$$\begin{array}{l}
\mathbf{oplus}\ a\ b \ \Leftarrow\ \underline{\text{rec}}\ b \\
\quad \mathbf{oplus}\ a\ b \ \Leftarrow\ \underline{\text{case}}\ b \\
\qquad \mathbf{oplus}\ a\ \mathsf{oz} \ \Rightarrow\ a \\
\qquad \mathbf{oplus}\ a\ (\mathsf{os}\ b) \ \Rightarrow\ \mathsf{os}\ (\mathbf{oplus}\ a\ b) \\
\qquad \mathbf{oplus}\ a\ (\mathsf{olim}\ f) \ \Rightarrow\ \mathsf{olim}\ (\lambda n\ \Rightarrow\ \mathbf{oplus}\ a\ (f\ n))
\end{array}$$

Categorically, Ord is an initial algebra $\mu X.1 + X + \mathsf{Nat} \to X$, it is an instance of a *strictly positive type*. Strictly positive types may use function types, like

[7] 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.

in $\mathsf{Nat} \to X$ but we do not allow type variables to appear on the left-hand side of the arrow. I.e. $\mu X.X \to \mathsf{Bool}$ is not strictly positive because X appears negatively, but neither is $\mu X.(X \to \mathsf{Bool}) \to \mathsf{Bool}$ 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

$$\underline{\text{data}}\quad \frac{n : \mathsf{Nat}}{\mathsf{Usp}\, n : \star}$$

$$\underline{\text{data}}\quad \frac{a : \mathsf{Usp}\, n \quad as : \mathsf{Tel}\, n}{\mathsf{Elsp}\, a\, as : \star}$$

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

$$\frac{A : \star \quad b : \mathsf{Usp}\, n}{\text{`arr'}\, A\, b : \mathsf{Usp}\, n}$$

and a corresponding constructor for Elsp:

$$\frac{f : A \to \mathsf{Elsp}\, b\, as}{\mathsf{fun}\, f : \mathsf{Elsp}\, (\text{`arr'}\, A\, b)\, as}$$

It is now easy to represent ordinals in this universe:

$$\underline{\text{let}}\quad \frac{}{\mathbf{ord} : \mathsf{Usp}\, n} \qquad \mathbf{ord} \Rightarrow \text{`mu'}\, (\text{`plus'}\, \text{`1'}\, (\text{`plus'}\, \mathsf{vl}\, (\text{`arr'}\, \mathsf{Nat}\, \mathsf{vl})))$$

We have been cheating a bit, because the constructor 'arr' refers to a type.

Thus $\mathsf{Usp}\; n : \star_{i+1}$ 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** : $\mathsf{Usp}\; n$ 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 many useful generic operations 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:

$$\underline{\text{data}}\ \ \frac{as, bs\ :\ \mathsf{Tel}\; n}{\mathsf{Map}\; as\; bs\ :\ \star}$$

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

$$\underline{\text{let}}\ \frac{\mathsf{fs} : \mathsf{Map}\ as\ bs \quad x : \mathsf{Elsp}\ a\ as}{\mathbf{gmap}\ fs\ x : \mathsf{Elsp}\ a\ bs}$$

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

$$\frac{}{\mathsf{mnil} : \mathsf{Map}\ \mathsf{tnil}\ \mathsf{tnil}} \quad \frac{f : \mathsf{Elsp}\ a\ as \to \mathsf{Elsp}\ b\ bs \quad fs : \mathsf{Map}\ as\ bs}{\mathsf{mcons}\ f\ fs : \mathsf{Map}\ (\mathsf{tcons}\ a\ as)\ (\mathsf{tcons}\ b\ bs)}$$

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

$$\frac{fs : \mathsf{Map}\ as\ bs}{\mathsf{mext}\ fs : \mathsf{Map}\ (\mathsf{tcons}\ a\ as)\ (\mathsf{tcons}\ a\ bs)}$$

Note that mext *fs* isn't just mcons $(\lambda\ x \Rightarrow x)$ *fs* because instead of the identity we need a function of the type Elsp *a as* $\to$ Elsp *a bs*. 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:

```
$\mathbf{gmap}\, fs\; x \;\Leftarrow\; \underline{\mathrm{rec}}\; x$
  $\mathbf{gmap}\, fs\; x \;\Leftarrow\; \underline{\mathrm{case}}\; x$
    $\mathbf{gmap}\, fs\; (\mathsf{inl}\; x) \;\Rightarrow\; \mathsf{inl}\; (\mathbf{gmap}\, fs\; x)$
    $\mathbf{gmap}\, fs\; (\mathsf{inr}\; y) \;\Rightarrow\; \mathsf{inr}\; (\mathbf{gmap}\, fs\; y)$
    $\mathbf{gmap}\, fs\; \mathsf{void} \;\Rightarrow\; \mathsf{void}$
    $\mathbf{gmap}\, fs\; (\mathsf{pair}\; x\; y) \;\Rightarrow\; \mathsf{pair}\; (\mathbf{gmap}\, fs\; x)\; (\mathbf{gmap}\, fs\; y)$
    $\mathbf{gmap}\, fs\; (\mathsf{fun}\; f) \;\Rightarrow\; \mathsf{fun}\; (\lambda x \;\Rightarrow\; \mathbf{gmap}\, fs\; (f\; x))$
    $\mathbf{gmap}\, fs\; (\mathsf{top}\; x) \;\Leftarrow\; \underline{\mathrm{case}}\; fs$
      $\mathbf{gmap}\; (\mathsf{mcons}\; f\; fs)\; (\mathsf{top}\; x) \;\Rightarrow\; \mathsf{top}\; (f\; x)$
      $\mathbf{gmap}\; (\mathsf{mext}\; fs)\; (\mathsf{top}\; x) \;\Rightarrow\; \mathsf{top}\; (\mathbf{gmap}\, fs\; x)$
    $\mathbf{gmap}\, fs\; (\mathsf{pop}\; x) \;\Leftarrow\; \underline{\mathrm{case}}\; fs$
      $\mathbf{gmap}\; (\mathsf{mcons}\; f\; fs)\; (\mathsf{pop}\; x) \;\Rightarrow\; \mathsf{pop}\; (\mathbf{gmap}\, fs\; x)$
      $\mathbf{gmap}\; (\mathsf{mext}\; fs)\; (\mathsf{pop}\; x) \;\Rightarrow\; \mathsf{pop}\; (\mathbf{gmap}\, fs\; x)$
    $\mathbf{gmap}\, fs\; (\mathsf{push}\; x) \;\Rightarrow\; \mathsf{push}\; (\mathbf{gmap}\; (\mathsf{mext}\; fs)\; x)$
    $\mathbf{gmap}\, fs\; (\mathsf{in}\; x) \;\Rightarrow\; \mathsf{in}\; (\mathbf{gmap}\; (\mathsf{mext}\; fs)\; x)$
```

The cases for the proper data constructors (inl, inr, void, pair, fun and in) are standard and just push **gmap** under the constructor. The other cases deal with the environment: top and pop have to analyse whether he environment has been constructed using mcons or mext while in the line for push we can reap the fruits by using mext instead having to use a non-structural recursive call to **gmap**.

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**:

$$\underline{\text{let}}\ \frac{}{\textbf{list}\ :\ \mathsf{Usp}\,(\mathsf{suc}\ n)}\quad \textbf{list}\ \Rightarrow\ \text{‘}\mathsf{mu}\text{’}\,(\text{‘}\mathsf{plus}\text{’}\,\text{‘}\mathsf{1}\text{’}\,(\text{‘}\mathsf{times}\text{’}\,(\mathsf{wk}\ \mathsf{vl})\ \mathsf{vl}))$$

to obtain

$$\underline{\text{let}}\ \frac{f\ :\ \mathsf{Elsp}\ a\ as \rightarrow \mathsf{Elsp}\ b\ as \quad xs\ :\ \mathsf{Elsp}\ \textbf{list}\,(\mathsf{tcons}\ a\ as)}{\textbf{map}\ f\ xs\ :\ \mathsf{Elsp}\ \textbf{list}\,(\mathsf{tcons}\ b\ as)}$$

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 pedagogical 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. $\mathsf{Stream}\,X = \nu X.A \times X$. 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 $\mu X.(X \rightarrow \mathsf{Bool}) \rightarrow \mathsf{Bool}$ [8] 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.

[8] Note that $(X \to \mathsf{Bool}) \to \mathsf{Bool}$ is covariant, unlike $X \to \mathsf{Bool} \times \mathsf{Bool}$ which is contravariant.

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 [17] to become a popular basis for generic programming in Haskell [23, 22, 40]. 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* $El : U \to \star$. 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*, $\mathsf{Rep} : \star \to \star$. An element of $\mathsf{Rep}\ T$ 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 $U \to \star$ 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 [20], the so-called 'Generalized Algebraic Data Types' [38]. 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 -> Reb 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':

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

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!

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

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. For example, aux might compute the notion of 'one-hole context' appropriate to its argument, in an attempt to support generic structure editing. Unfortunately, U is not a Sigma type but a System F 'weak' existential: direct access to the type it packages is not possible. There is no way to express operations whose types explicitly invoke these functions, e.g., plugging a value into a one-hole context to recover an element of the original type.

System F's capacity for secrecy is rather useful in other circumstances, but it is a problem 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
```

This problem is not the fault of representation types as opposed to universes (although the latter are a little neater for such tasks): it's just a shortfall in the expressivity of Haskell.

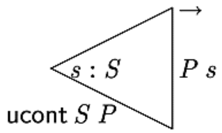
5 Containers

In section 3 we started with a semantic definition of the universe of finite types, 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 a look at unary containers, which model type constructors with one parameter, i.e. inhabitants of $\star \to \star$, where the parameter represents a type of *payload* elements to be stored within some structure. A unary container is given by a type of shapes $S : \star$ and a family of positions $P : S \to \star$. I.e. we define:

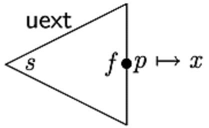
$$\underline{\text{data}} \quad \frac{}{\mathsf{UCont} : \star} \qquad \underline{\text{where}} \quad \frac{S : \star \quad P : S \to \star}{\mathsf{ucont}\, S\, P \,:\, \mathsf{UCont}}$$



We illustrate containers with *triangle diagrams*, intended to resemble a node in a tree with its root at the left. As with W-types, we indicate a choice of shape in the apex, and the base then represents a *set* of points, dependent on the shape; the arrow at the top of the base indicates that the points in this set are positions for payload.

The *extension* of a container is the parametric datatype which it describes: its values consist of a choice of shape and an assignment of payload to positions, represented functionally.

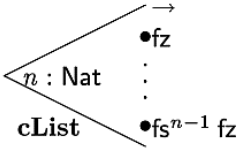
$$\underline{\text{data}} \quad \frac{C : \mathsf{UCont} \quad X : \star}{\mathsf{UExt}\, C\, X \,:\, \star} \qquad \underline{\text{where}} \quad \frac{s : S \quad f : P\, s \to X}{\mathsf{uext}\, s\, f \,:\, \mathsf{UExt}\, (\mathsf{ucont}\, S\, P)\, X}$$



We can also illustrate inhabitants of such a type by a diagram, labelling the base with the function f which takes each position p to some payload value x.

An example of a unary container is this representation of List:

$$\underline{\text{let}}\ \frac{}{\mathbf{cList} : \mathsf{UCont}} \quad \mathbf{cList} \Rightarrow \mathsf{ucont}\ \mathsf{Nat}\ \mathsf{Fin}$$



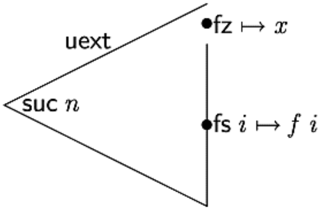
The shape of a list is its length, i.e. a natural number, and a list with shape $n : \mathsf{Nat}$ has $\mathsf{Fin}\ n$ positions. We can re-implement the constructors for lists to target the container representation

$$\underline{\text{let}}\ \frac{i : \mathsf{Fin\ zero}}{\mathbf{noFin}\ i : X} \quad \mathbf{noFin}\ i \Leftarrow \underline{\mathsf{case}}\ i$$

$$\underline{\text{let}}\ \frac{}{\mathbf{cnil} : \mathsf{UExt}\ \mathbf{cList}\ X} \quad \mathbf{cnil} \Rightarrow \mathsf{uext\ zero}\ \mathbf{noFin}$$

$$\underline{\text{let}}\ \frac{x : X \quad f : \mathsf{Fin}\, n \to X \quad i : \mathsf{Fin}\,(\mathsf{suc}\, n)}{\mathbf{caseFin}\, x\, f\, i : X}$$

$$\begin{array}{l}
\mathbf{caseFin}\, x\, f\, i \ \Leftarrow\ \underline{\text{case}}\ i \\
\quad \mathbf{caseFin}\, x\, f\, \mathsf{fz} \ \Rightarrow\ x \\
\quad \mathbf{caseFin}\, x\, f\, (\mathsf{fs}\, i) \ \Rightarrow\ f\ i
\end{array}$$



$$\underline{\text{let}}\ \frac{x : X \quad xs : \mathsf{UExt}\, \mathbf{cList}\, X}{\mathbf{ccons}\, x\, xs : \mathsf{UExt}\, \mathbf{cList}\, X}$$

$$\begin{array}{l}
\mathbf{ccons}\, x\, xs \ \Leftarrow\ \underline{\text{case}}\ xs \\
\quad \mathbf{ccons}\, x\, (\mathsf{uext}\, n\, f) \ \Rightarrow\ \mathsf{uext}\,(\mathsf{suc}\, n)\,(\mathbf{caseFin}\, x\, f)
\end{array}$$

Exercise 10. We can also give a container representation for binary trees, here shapes are given by trees containing data, positions by paths through such a tree:

$$\underline{\text{data}}\ \frac{}{\mathsf{cTreeS} : \star}\quad \underline{\text{where}}\ \frac{}{\mathsf{sleaf} : \mathsf{cTreeS}}\ \ \frac{l, r : \mathsf{cTreeS}}{\mathsf{snode}\, l\, r : \mathsf{cTreeS}}$$

$$\underline{\text{data}}\ \frac{s : \mathsf{cTreeS}}{\mathsf{cTreeP}\, s : \mathsf{Type}}\quad \underline{\text{where}}\ \frac{}{\mathsf{phere} : \mathsf{cTreeP}\,(\mathsf{node}\, l\, r)}$$

$$\frac{q \;:\; \mathsf{cTreeP}\; l}{\mathsf{pleft}\; q \;:\; \mathsf{cTreeP}\; (\mathsf{node}\; l\; r)} \quad \frac{p \;:\; \mathsf{cTreeP}\; r}{\mathsf{pright}\; p \;:\; \mathsf{cTreeP}\; (\mathsf{node}\; l\; r)}$$

$$\underline{\text{let}}\;\; \frac{}{\mathbf{cTree} \;:\; \mathsf{UCont}} \qquad \mathbf{cTree} \;\Rightarrow\; \mathsf{uext}\; \mathsf{cTreeS}\; \mathsf{cTreeP}$$

Implement leaf and node for **cTree**:

$$\underline{\text{let}}\;\; \frac{}{\mathbf{cleaf} \;:\; \mathsf{UExt}\; \mathbf{cTree}\; X}$$

$$\underline{\text{let}}\;\; \frac{r \;:\; \mathsf{UExt}\; \mathbf{cTree}\; X \quad x \;:\; X \quad r \;:\; \mathsf{UExt}\; \mathbf{cTree}\; X}{\mathbf{cnode}\; l\; x\; r \;:\; \mathsf{UExt}\; \mathbf{cTree}\; X}$$

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:

$$\underline{\text{let}}\;\; \frac{C \;:\; \mathsf{UCont} \quad f \;:\; X \to Y \quad c \;:\; \mathsf{UExt}\; C\; X}{\mathbf{ucmap}\; C\; f\; c \;:\; \mathsf{UExt}\; C\; Y}$$

$$\begin{array}{l} \mathbf{ucmap}\; C\; f\; c \;\Leftarrow\; \underline{\text{case}}\; c \\ \;\; \mathbf{ucmap}\; (\mathsf{ucont}\; S\; P)\; f\; (\mathsf{uext}\; s\; g) \;\Rightarrow\; \mathsf{uext}\; s\; (\lambda x \Rightarrow f\; (g\; x)) \end{array}$$

A morphism between functors is a natural transformation, e.g. *reverse* is a natural transformation from list to list. We can explicitly represent morphisms between containers: given unary containers $\mathsf{ucont}\, S\, P$ and $\mathsf{ucont}\, T\, Q$, a morphism is a function on shapes $f : S \to T$ and a family of functions on positions, which assigns to every position in the target a position in the source, i.e.

$$u : \forall s{:}S \Rightarrow Q\,(f\ s) \to P\ s$$

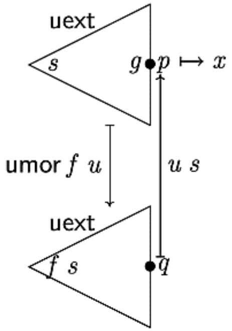
The contravariance of the function on positions may be surprising, however, it can be intuitively understood by the fact that we can always say where a piece of payload comes from but not where it goes to, since it may be copied or disappear. Hence we define:

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

To every formal morphism between containers we assign a family of maps, parametric in the payload type:

$$\underline{\text{let}}\ \frac{m : \mathsf{UMor}\ C\ D \quad c : \mathsf{UExt}\ C\ X}{\mathbf{UMapp}\ m\ c : \mathsf{UExt}\ D\ X}$$

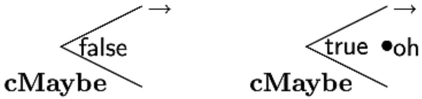
$$\begin{array}{l}
\mathbf{UMapp}\ m\ c \Leftarrow \underline{\text{case}}\ m \\
\quad \mathbf{UMapp}\ (\mathsf{umor}\ f\ u)\ c \Leftarrow \underline{\text{case}}\ c \\
\quad\quad \mathbf{UMapp}\ (\mathsf{umor}\ f\ u)\ (\mathsf{uext}\ s\ g) \Rightarrow \\
\quad\quad\quad \mathsf{uext}\ (f\ s)\ (\lambda q \Rightarrow g\ (u\ s\ q))
\end{array}$$



As an example we can define **cHead** for the container representation of lists, since we require totality we will define a morphism between **cList** and **cMaybe**, which relates to Haskell's `Maybe` type:

$$\underline{\text{let}}\ \frac{}{\mathbf{cMaybe} : \mathsf{UCont}} \quad \mathbf{cMaybe} \Rightarrow \mathsf{ucont\ Bool\ So}$$

There are two possible layouts for **cMaybe** containers:



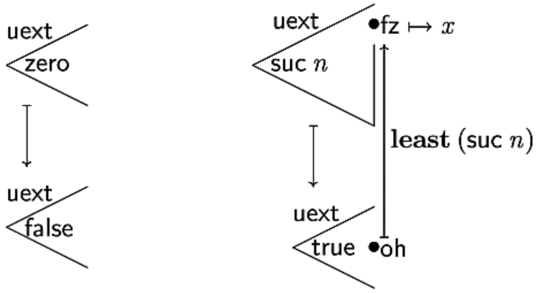
There are then two cases to consider when defining our morphism. For the zero shape of **cList**, we choose the false shape of **cMaybe**, leaving no positions to fill. For any other input shape, we choose true, leaving one position to fill: we fetch its payload from input position fz—the head.

$$\underline{\text{let}}\ \frac{n\ :\ \mathsf{Nat}}{\mathbf{isSuc}\ :\ \mathsf{Bool}} \qquad \begin{array}{l} \mathbf{isSuc}\,n\ \Leftarrow\ \underline{\mathsf{case}}\,n \\ \quad \mathbf{isSuc}\,\mathsf{zero}\ \Rightarrow\ \mathsf{false} \\ \quad \mathbf{isSuc}\,(\mathsf{suc}\,n)\ \Rightarrow\ \mathsf{true} \end{array}$$

$$\underline{\text{let}}\ \frac{n\ :\ \mathsf{Nat}\quad q\ :\ \mathsf{So}\,(\mathbf{isSuc}\,n)}{\mathbf{least}\,n\ q\ :\ \mathsf{Fin}\,n} \qquad \begin{array}{l} \mathbf{least}\,n\ q\ \Leftarrow\ \underline{\mathsf{case}}\,n \\ \quad \mathbf{least}\,\mathsf{zero}\ q\ \Leftarrow\ \underline{\mathsf{case}}\,q \\ \quad \mathbf{least}\,(\mathsf{suc}\,n)\ q\ \Rightarrow\ \mathsf{fz} \end{array}$$

$$\underline{\text{let}}\ \frac{}{\mathbf{cHead}\ :\ \mathsf{UMor}\,\mathbf{cList}\,\mathbf{cMaybe}} \qquad \mathbf{cHead}\ \Rightarrow\ \mathsf{umor}\,\mathbf{isSuc}\,\mathbf{least}$$

We illustrate these two cases as follows:



It is not hard to show that these families of maps are always natural transformations in the categorical sense, with respect to UExt's interpretation of unary containers as functors. 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 the representation of *reverse* as morphism between unary containers, i.e.

$$\underline{\text{let}}\quad \frac{}{\mathbf{cRev}\ :\ \mathsf{UMor}\,\mathbf{cList}\,\mathbf{cList}}$$

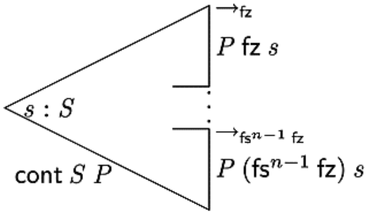
Exercise 12. While the interpretation of morphisms is full, i.e. every natural transformation comes from a container morphism, the same is not true for containers as representations of functors. Can you find a functor which is not representable as a unary container?

5.2 n-ary containers

We are now going to interpret strictly positive types Usp as containers by implementing operations on containers which correspond to constructors of Usp. We reap the fruits by defining a simple evaluation function which **evalC** which interprets Usps as containers. First of all we have to generalize our previous definition to n-ary containers to reflect the presence of variables in Usp:

$$\underline{\text{data}}\quad \frac{n : \mathsf{Nat}}{\mathsf{Cont}\ n : \star}$$

$$\underline{\text{where}}\quad \frac{S : \star \quad P : \mathsf{Fin}\ n \to S \to \star}{\mathsf{cont}\ S\ P : \mathsf{Cont}\ n}$$



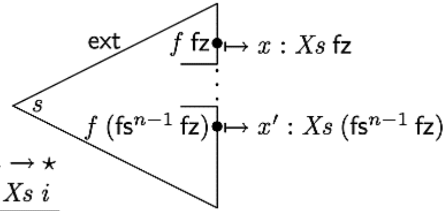
$\rightarrow_{\mathsf{fz}}$
$P\ \mathsf{fz}\ s$
$s : S$
$\rightarrow_{\mathsf{fs}^{n-1}\ \mathsf{fz}}$
$P\ (\mathsf{fs}^{n-1}\ \mathsf{fz})\ s$
$\mathsf{cont}\ S\ P$

It is important to understand that we use only one shape but n sets of positions. E.g. consider the two-parameter container of leaf and node labelled trees, the shape of a tree is given by ignoring the data, but the positions for leaf-data and tree-data are different. Accordingly, in our diagrams, we may segment the base of the triangle to separate the surfaces where each sort of payload attaches and we index the arrows accordingly.

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

$$\underline{\text{data}}\quad \frac{C : \mathsf{Cont}\, n \quad Xs : \mathsf{Fin}\, n \to \star}{\mathsf{Ext}\; C\; Xs : \star}$$

$$\underline{\text{where}}\quad \frac{\begin{array}{l} P : \mathsf{Fin}\, n \to S \to \star \quad Xs : \mathsf{Fin}\, n \to \star \\ s : S \quad f : \forall i{:}\mathsf{Fin}\, n \Rightarrow P\; i\; s \to Xs\; i \end{array}}{\mathsf{ext}\; s\; f : \mathsf{Ext}\; (\mathsf{cont}\; S\; P)\; Xs}$$



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

$$C : \mathsf{Cont}\, n \quad Xs, Ys : \mathsf{Fin}\, n \to \star$$

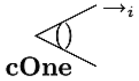
$$\underline{\text{let}}\ \frac{fs : \forall i{:}\mathsf{Fin}\ n \Rightarrow Xs\ i \rightarrow Ys\ i \quad x : \mathsf{Ext}\ C\ Xs}{\mathbf{map}\ C\ fs\ x : \mathsf{Ext}\ C\ Ys}$$

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:

$$\underline{\text{let}}\ \frac{}{\mathbf{cZero} : \mathsf{Cont}\ n} \quad \mathbf{cZero} \Rightarrow \mathsf{cont}\ \mathsf{Zero}\ (\lambda i; s \Rightarrow \mathsf{Zero})$$

$$\underline{\text{let}}\ \frac{}{\mathbf{cOne} : \mathsf{Cont}\ n} \quad \mathbf{cOne} \Rightarrow \mathsf{cont}\ \mathsf{One}\ (\lambda i; s \Rightarrow \mathsf{Zero})$$



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

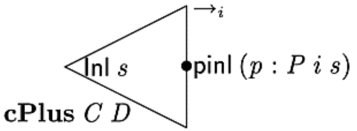
$$\underline{\text{data}}\ \frac{P : A \to \star \quad Q : B \to \star \quad ab : \mathsf{Plus}\ A\ B}{\mathsf{PPlus}\ P\ Q\ ab : \star}$$

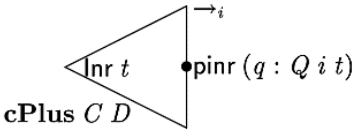
$$\underline{\text{where}}\ \frac{p : P\ a}{\mathsf{pinl}\ p : \mathsf{PPlus}\ P\ Q\ (\mathsf{lnl}\ a)} \quad \frac{q : Q\ b}{\mathsf{pinr}\ q : \mathsf{PPlus}\ P\ Q\ (\mathsf{lnr}\ b)}$$

Putting everything together we define the containers as follows, with the two typical layouts shown in the diagrams:

$$\underline{\text{let}}\ \frac{C, D : \mathsf{Cont}\ n}{\mathbf{cPlus}\ C\ D : \mathsf{Cont}\ n}$$

$$\begin{array}{l}
\mathbf{cPlus}\ C\ D \Leftarrow \underline{\text{case}}\ C \\
\quad \mathbf{cPlus}\ (\mathsf{cont}\ S\ P)\ D \Leftarrow \underline{\text{case}}\ D \\
\quad \mathbf{cPlus}\ (\mathsf{cont}\ S\ P)\ (\mathsf{cont}\ T\ Q) \\
\qquad \Rightarrow \mathsf{cont}\ (\mathsf{Plus}\ S\ T) \\
\qquad\qquad (\lambda i \Rightarrow \mathsf{PPlus}\ (P\ i)\ (Q\ i))
\end{array}$$





Let's turn our attention to products: on shapes again this is just the type-theoretic product, Times—each component has a shape. Given two containers

$C = \mathsf{cont}\ S\ P, D = \mathsf{cont}\ T\ Q$, as above, what are the positions in a product shape $\mathsf{Pair}\ s\ t : \mathsf{Times}\ S\ T$? 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:

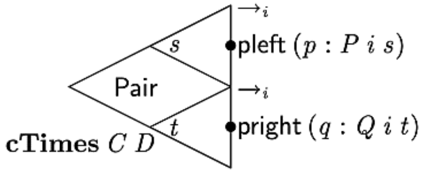
$$\underline{\text{data}}\ \frac{P : A \to \star \quad Q : B \to \star \quad ab : \mathsf{Times}\ A\ B}{\mathsf{PTimes}\ P\ Q\ ab : \star}\ \underline{\text{where}}$$

$$\frac{p : P\ a}{\mathsf{pleft}\ p : \mathsf{PTimes}\ P\ Q\ (\mathsf{Pair}\ a\ b)} \quad \frac{q : Q\ b}{\mathsf{pright}\ q : \mathsf{PTimes}\ P\ Q\ (\mathsf{Pair}\ a\ b)}$$

and we define the product of containers as:

$$\underline{\text{let}}\ \frac{C, D : \mathsf{Cont}\ n}{\mathbf{cTimes}\ C\ D : \mathsf{Cont}\ n}$$

$$\begin{array}{l}
\mathbf{cTimes}\ C\ D \Leftarrow \underline{\text{case}}\ C \\
\quad \mathbf{cTimes}\ (\mathsf{cont}\ S\ P)\ D \Leftarrow \underline{\text{case}}\ D \\
\quad\quad \mathbf{cTimes}\ (\mathsf{cont}\ S\ P)\ (\mathsf{cont}\ T\ Q) \\
\quad\quad\quad \Rightarrow \mathsf{cont}\ (\mathsf{Times}\ S\ T) \\
\quad\quad\quad\quad\quad (\lambda i \Rightarrow \mathsf{PTimes}\ (P\ i)\ (Q\ i))
\end{array}$$



Exercise 14. Define an operation on containers which interprets constant expo-

nentation as described in section 4.3, i.e. define

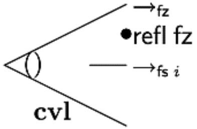
$$\underline{\text{let}}\ \frac{A : \star \quad C : \mathsf{Cont}\, n}{\mathbf{cArr}\, A\, C : \mathsf{Cont}\, n}$$

5.4 Structural operations

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

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

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



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 auxiliary operator on positions:

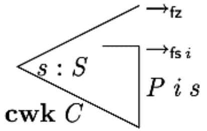
$$\underline{\text{let}}\ \frac{P : \mathsf{Fin}\, n \to S \to \star \quad i : \mathsf{Fin}\, (\mathsf{suc}\, n) \quad s : S}{\mathbf{Pwk}\, P\, i\, s : \star}$$

$$\begin{array}{l} \mathbf{Pwk}\, P\; i\; s \;\Leftarrow\; \underline{\text{case}}\, i \\ \quad \mathbf{Pwk}\, P\, \mathsf{fz}\, s \;\Rightarrow\; \mathsf{Zero} \\ \quad \mathbf{Pwk}\, P\, (\mathsf{fs}\, i)\, s \;\Rightarrow\; P\; i\; s \end{array}$$

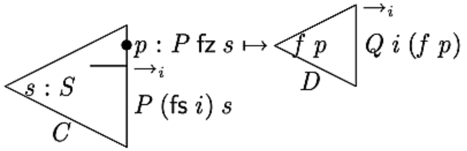
and use this to define:

$$\underline{\text{let}}\ \frac{C : \mathsf{Cont}\, n}{\mathbf{cwk}\, C : \mathsf{Cont}\, (\mathsf{suc}\, n)}$$

$$\begin{array}{l} \mathbf{cwk}\, C \;\Leftarrow\; \underline{\text{case}}\, C \\ \quad \mathbf{cwk}\, (\mathsf{cont}\, S\; P) \;\Rightarrow\; \mathsf{cont}\, S\, (\mathbf{Pwk}\, P) \end{array}$$



The case of local definition is more interesting. We assume as given two containers: $C = \mathsf{cont}\, S\; P \;:\; \mathsf{Cont}\, (\mathsf{suc}\, n), D = \mathsf{cont}\, T\; Q \;:\; \mathsf{Cont}\, n$. We create a new n-ary container by *binding* variable fz of C to D, hence attaching D-structures to each fz-indexed position of a C-structure. The i-positions of the result correspond either to i-positions of some inner D, or the *free* $(\mathsf{fs}\, i)$-positions of the outer C.



To record the shape of the whole thing, we need to store the outer C shape, some $s : S$, and the inner D shapes: there is one for each outer fz-position, hence we need a function $f : P\,\mathsf{fz}\,s \rightarrow T$. As before we abstract from the specific position types and define abstractly:

$$\underline{\text{data}}\ \frac{S : \star \quad P_0 : S \rightarrow \star \quad T : \star}{\mathsf{Sdef}\ S\ P_0\ T : \star}\ \underline{\text{where}}\ \frac{s : S \quad f : P_0\,s \rightarrow T}{\mathsf{sdef}\ s\ f\ : \mathsf{Sdef}\ S\ P_0\ T}$$

What is a position in the new container, for a given index i? It must either be a 'free' outer position, given by $P\,(\mathsf{fs}\,i)$, or the pair of a 'bound' outer position with an inner position given by $Q\,i$. Hence, we define a general operator for positions in Sdef, which we can instantiate suitably for each index:

$$\underline{\text{data}}\quad \frac{P_0, P' : S \rightarrow \star \quad Q : T \rightarrow \star \quad x : \mathsf{Sdef}\ S\ P_0\ T}{\mathsf{Pdef}\ P_0\ P'\ Q\ x : \star}$$

$$\underline{\text{where}} \quad \frac{p \,:\, P'\, s}{\mathsf{ppos}\, p \,:\, \mathsf{Pdef}\; P_0\; P'\; Q\; (\mathsf{sdef}\; s\; f)} \quad \frac{p \,:\, P_0\, s \quad q \,:\, Q\,(f\,p)}{\mathsf{qpos}\; p\; q \,:\, \mathsf{Pdef}\; P_0\; P'\; Q\; (\mathsf{sdef}\; s\; f)}$$

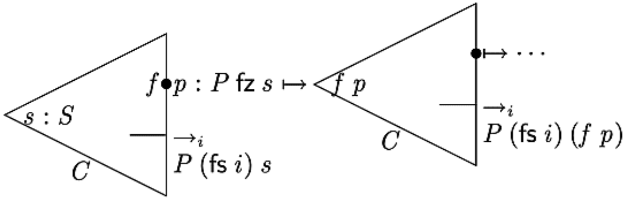
Putting the components together, we can present the definition operator:

$$\underline{\text{let}} \quad \frac{C \,:\, \mathsf{Cont}\,(\mathsf{suc}\, n) \quad D \,:\, \mathsf{Cont}\; n}{\mathbf{cdef}\; C\; D \,:\, \mathsf{Cont}\; n}$$

$$\begin{array}{l}
\mathbf{cdef}\; C\; D \;\Leftarrow\; \underline{\text{case}}\; C \\
\quad \mathbf{cdef}\; (\mathsf{cont}\; S\; P)\; D \;\Leftarrow\; \underline{\text{case}}\; D \\
\quad\quad \mathbf{cdef}\; (\mathsf{cont}\; S\; P)\; (\mathsf{cont}\; T\; Q) \\
\quad\quad\quad\quad \Rightarrow\; \mathsf{cont}\; (\mathsf{Sdef}\; S\; (P\; \mathsf{fz})\; T)\; (\lambda i \Rightarrow \mathsf{Pdef}\; (P\; \mathsf{fz})\; (P\; (\mathsf{fs}\; i))\; (Q\; i))
\end{array}$$

5.5 Inductive types (μ)

To interpret the mu constructor we take an $n+1$-ary container $C = \mathsf{cont}\; S\; P$: $\mathsf{Cont}\,(\mathsf{suc}\, n)$ and try to find a container which represents the initial algebra with respect to the 'bound' index fz. For each shape $s : S$, $P\; \mathsf{fz}\; s$ gives the positions of recursive subobjects. Meanwhile the positions for i-indexed payload at each node are given by $P\; (\mathsf{fs}\; i)$.



Clearly, to be able to construct a tree at all, there must be at least one 'base case' s for which $P\ \mathsf{fz}\ s$ is empty. Otherwise there are no leaves and the corresponding tree type is empty.

How can we describe the shapes of these trees? At each node, we must supply the top-level shape, together with a function which gives the shape f the subtrees. This is given exactly by $\mathsf{W}\ S\ (P\ \mathsf{fz})$. Given a shape in form of a W-tree, the positions at index i correspond to path leading to a $P\ (\mathsf{fs}\ i)$ somewhere in the tree. We can define the types of paths in a tree in general:

$$\underline{\text{data}}\ \ \frac{S : \star \quad P_0, P' : S \to \star \quad x : \mathsf{W}\ S\ P_0}{\mathsf{PW}\ S\ P_0\ P'\ x : \star}$$

$$\underline{\text{where}}\ \ \frac{x : P'\ s}{\mathsf{here}\ x : \mathsf{PW}\ S\ P_0\ P'\ (\mathsf{Sup}\ s\ f)} \quad \frac{p : P_0\ s \quad r : \mathsf{PW}\ S\ P_0\ P'\ (f\ p)}{\mathsf{under}\ p\ r : \mathsf{PW}\ S\ P_0\ P'\ (\mathsf{Sup}\ s\ f)}$$

The idea is that a path either exits at the top level node here at a position in $P'\ s$ or continues into the subtree under a positions in $P_0\ s$. Putting shapes and paths together we arrive at the following definition:

$$\underline{\text{let}}\ \frac{C\ :\ \mathsf{Cont}\,(\mathsf{suc}\,n)}{\mathbf{cMu}\ C\ :\ \mathsf{Cont}\ n}$$

$$\begin{array}{l} \mathbf{cMu}\ C\ \Leftarrow\ \underline{\text{case}}\ C \\ \quad \mathbf{cMu}\,(\mathsf{cont}\ S\ P)\ \Rightarrow\ \mathsf{cont}\,(\mathsf{W}\ S\ (P\ \mathsf{fz}))\,(\lambda i \Rightarrow \mathsf{PW}\ S\ (P\ \mathsf{fz})\,(P\ (\mathsf{fs}\ i))) \end{array}$$

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:

$$\underline{\mathrm{let}}\ \ \frac{a\ :\ \mathsf{Ucf}\ n}{\mathbf{evalC}\ a\ :\ \mathsf{Cont}\ n}$$

$$\begin{array}{l}
\mathbf{evalC}\ a\ \Leftarrow\ \underline{\mathrm{rec}}\ a \\
\ \ \mathbf{evalC}\ a\ \Leftarrow\ \underline{\mathrm{case}}\ a \\
\ \ \ \ \mathbf{evalC}\ \mathsf{vl}\ \Rightarrow\ \mathbf{cvl} \\
\ \ \ \ \mathbf{evalC}\ (\mathsf{wk}\ a)\ \Rightarrow\ \mathbf{cwk}\ (\mathbf{evalC}\ a) \\
\ \ \ \ \mathbf{evalC}\ `\mathsf{0}'\ \Rightarrow\ \mathbf{cZero} \\
\ \ \ \ \mathbf{evalC}\ (`\mathsf{plus}'\ a\ b)\ \Rightarrow\ \mathbf{cPlus}\ (\mathbf{evalC}\ a)\ (\mathbf{evalC}\ b) \\
\ \ \ \ \mathbf{evalC}\ `\mathsf{1}'\ \Rightarrow\ \mathbf{cOne} \\
\ \ \ \ \mathbf{evalC}\ (`\mathsf{times}'\ a\ b)\ \Rightarrow\ \mathbf{cTimes}\ (\mathbf{evalC}\ a)\ (\mathbf{evalC}\ b) \\
\ \ \ \ \mathbf{evalC}\ (\mathsf{def}\ f\ a)\ \Rightarrow\ \mathbf{cdef}\ (\mathbf{evalC}\ f)\ (\mathbf{evalC}\ a) \\
\ \ \ \ \mathbf{evalC}\ (`\mathsf{mu}'\ f)\ \Rightarrow\ \mathbf{cMu}\ (\mathbf{evalC}\ f)
\end{array}$$

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

$$\underline{\mathrm{let}}\ \ \frac{a\ :\ \mathsf{Ucf}\ n \quad Xs\ :\ \mathsf{Fin}\ n \to \star}{\mathbf{eval}\ a\ Xs\ :\ \star} \quad \mathbf{eval}\ a\ Xs\ \Rightarrow\ \mathsf{Ext}\ (\mathbf{evalC}\ a)\ Xs$$

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.

So far we have only interpreted the type names, i.e. the inhabitants of $\mathsf{Ucf}\, n$, what about the elements, i.e. the inhabitants of $\mathsf{Elcf}\, a\, as$? Using Ext we can define a semantic version of Elcf:

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

$$\underline{\text{let}}\ \frac{Cs : \mathsf{Tel}\, n \quad i : \mathsf{Fin}\, n}{\mathbf{TelEl}\, Cs\, i : \star}$$

```
$\mathbf{TelEl}\ Cs\ i \Leftarrow \underline{\text{rec}}\ Cs$
 $\mathbf{TelEl}\ Cs\ i \Leftarrow \underline{\text{case}}\ i$
  $\mathbf{TelEl}\ Cs\ \mathsf{fz} \Leftarrow \underline{\text{case}}\ Cs$
   $\mathbf{TelEl}\ (\mathsf{tcons}\ C\ Cs)\ \mathsf{fz} \Rightarrow \mathsf{Ext}\ C\ (\mathsf{TelEl}\ Cs)$
  $\mathbf{TelEl}\ Cs\ (\mathsf{fs}\ i) \Leftarrow \underline{\text{case}}\ Cs$
   $\mathbf{TelEl}\ (\mathsf{tcons}\ C\ Cs)\ (\mathsf{fs}\ i) \Rightarrow \mathbf{TelEl}\ Cs\ i$
```

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

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:

$$\underline{\text{let}}\ \frac{A : \star}{\mathbf{DecEq}\,A : \star}$$

$$\mathbf{DecEq}\,A \Rightarrow \forall a, a' : A \Rightarrow \mathsf{Dec}\,(a = a')$$

$$\underline{\text{data}}\ \ \frac{n\ :\ \mathsf{Nat}}{\mathsf{SCont}\ n\ :\ \star}$$

$$\underline{\text{where}}\ \ \frac{S\ :\ \star\quad eqS\ :\ \mathbf{DecEq}\ S\quad P\ :\ \mathsf{Fin}\ n \to S \to \mathsf{Nat}}{\mathsf{scont}\ S\ eqS\ P\ :\ \mathsf{SCont}\ n}$$

$$\underline{\text{data}}\ \ \frac{C\ :\ \mathsf{SCont}\ Xs\ :\ \mathsf{Fin}\ n \to \star}{\mathsf{SExt}\ C\ Xs\ :\ \star}$$

$$\underline{\text{where}}\ \ \frac{s\ :\ S\quad f\ :\ \forall i\!:\!\mathsf{Fin}\ n \Rightarrow \mathsf{Fin}\ (P\ i\ s) \to Xs\ i}{\mathsf{sext}\ s\ f\ :\ \mathsf{SCont}\ S\ eq\ P}$$

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

$$\underline{\text{let}}\ \ \frac{C, D\ :\ \mathsf{SCont}\ n}{\mathbf{SCTimes}\ C\ D\ :\ \mathsf{Cont}\ n}$$

$$\begin{array}{l}
\mathbf{SCTimes}\ C\ D\ \Leftarrow\ \underline{\text{case}}\ C \\
\quad \mathbf{SCTimes}\ (\mathsf{scont}\ S\ eqS\ P)\ D\ \Leftarrow\ \underline{\text{case}}\ D \\
\quad\quad \mathbf{SCTimes}\ (\mathsf{scont}\ S\ eqS\ P)\ (\mathsf{scont}\ T\ eqT\ Q)
\end{array}$$

$$\begin{array}{l}\Rightarrow\ \ \mathsf{scont}\,(\mathsf{Times}\,S\ T)\\ \qquad\qquad(\mathbf{TimesEq}\ \mathit{eqS}\ \mathit{eqT})\\ \qquad\qquad\ (\lambda i; s \Rightarrow \mathbf{plus}\,(P\ i\ s)\,(Q\ i\ s))\end{array}$$

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

$$\underline{\mathrm{let}}\ \ \frac{\mathit{eqS}\ :\ \mathbf{DecEq}\,S \quad \mathit{eqT}\ :\ \mathbf{DecEq}\ T}{\mathbf{TimesEq}\ \mathit{eqS}\ \mathit{eqT}\ :\ \mathbf{DecEq}\,(\mathsf{Times}\,S\ T)}$$

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

$$\underline{\mathrm{let}}\ \ \frac{C\ :\ \mathsf{SCont}\,n \quad Xs\ :\ \mathsf{Fin}\,n \to \star \quad \mathit{eqs}\ :\ \forall i\!:\!\mathsf{Fin}\ n \Rightarrow \mathsf{DecEq}\ (Xs\ i)}{\mathbf{SContEq}\ C\ Xs\ \mathit{eqs}\ :\ \mathsf{DecEq}\,(\mathsf{SExt}\ C\ Xs)}$$

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:

$$\underline{\text{let}}\ \frac{f, g : S \to T \quad a, b : S \quad p : f = g \quad q : a = b}{\mathbf{applEq}\ p\ q\ :\ f\ a = g\ b}$$

$$\begin{aligned}
&\mathbf{applEq}\ p\ q \ \Leftarrow\ \underline{\text{case}}\ p\\
&\quad \mathbf{applEq}\ \mathsf{refl}\ q \ \Leftarrow\ \underline{\text{case}}\ q\\
&\quad\quad \mathbf{applEq}\ \mathsf{refl}\ \mathsf{refl} \ \Rightarrow\ \mathsf{refl}
\end{aligned}$$

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

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

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

6 Derivatives

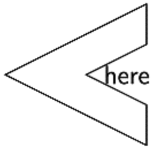
In [24] Huet introduced the *zipper* as a datatype to represent a position within a tree. The basic idea is that at every step on the path to the current position, we

remember the context left over. E.g. in the example of unlabelled binary trees,

$$\underline{\text{data}}\ \frac{}{\mathsf{BT} : \star}\ \underline{\text{where}}\ \frac{}{\mathsf{leaf} : \mathsf{BT}}\quad \frac{l, r : \mathsf{BT}}{\mathsf{node}\, l\, r : \mathsf{BT}}$$

the corresponding zipper type is:

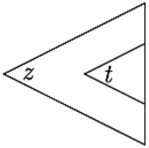
$$\begin{array}{ll} \underline{\text{data}} & \frac{}{\mathsf{Zipper} : \star} \\ \underline{\text{where}} & \frac{l : \mathsf{Zipper} \quad r : \mathsf{BT}}{\mathsf{left}\, l\, r : \mathsf{Zipper}}\ \frac{l : \mathsf{BT} \quad r : \mathsf{Zipper}}{\mathsf{right}\, l\, r : \mathsf{Zipper}}\ \frac{}{\mathsf{here} : \mathsf{Zipper}} \end{array}$$



We can think of a Zipper as a tree with one subtree chopped out at the place marked here. One of the operations on a zipper is to plug a binary tree into its hole, i.e. we define:[9]

```
$\underline{\text{let}}\ \dfrac{z : \mathsf{Zipper} \quad t : \mathsf{BT}}{\textbf{plug}\, z\, t : \mathsf{BT}}$

    $\textbf{plug}\, z\, t \Leftarrow \underline{\text{rec}}\, z$
     $\textbf{plug}\, z\, t \Leftarrow \underline{\text{case}}\, z$
      $\textbf{plug}\, (\mathsf{left}\, l\, r)\, t \Rightarrow \mathsf{node}\, (\textbf{plug}\, l\, t)\, r$
      $\textbf{plug}\, (\mathsf{right}\, l\, r)\, t \Rightarrow \mathsf{node}\, l\, (\textbf{plug}\, r\, t)$
      $\textbf{plug}\, \mathsf{here}\, t \Rightarrow t$
```



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 realised 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 $\mu X.F\,X$, e.g. $\mu X.1 + X \times X$ in the example of binary trees, then the corresponding zipper is $\mathsf{List}(\partial F\ (\mu X.F\ X))$. In the binary tree example $F\,X = 1 + X \times X$ and $\partial F\,X = 2 \times X$. Indeed Zipper is isomorphic to $\mathsf{List}\,(2 \times \mathsf{BT})$.

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:

$$\underline{\text{let}}\ \frac{a : \mathsf{Ucf}\ n \quad i : \mathsf{Fin}\ n}{\mathbf{partial}\ a\ i\ :\ \mathsf{Ucf}\ n}$$

[9] Here we have chosen the root-to-hole representation of contexts. Huet's hole-to-root presentation uses the same datatype. Both are useful, with the choice depending on where you need the most rapid access.

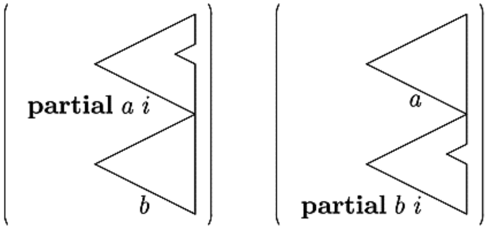
The parameter i denotes the argument on which we take the derivative, indeed the partial derivative really is a variable binding operation, this is obliterated by the usual notation $\frac{\partial F}{\partial X}$ which really binds X.

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:

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

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 first component, leaving the second intact or symmetrically, a hole in the second, leaving the first intact. Hence we arrive at

partial (‘times’ a b) i $\Rightarrow$ ‘plus’ (‘times’ (**partial** a i) b) (‘times’ a (**partial** b i))



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

partial ‘0’ i $\Rightarrow$ ‘0’
partial ‘1’ i $\Rightarrow$ ‘0’

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 $i =$ fz and none otherwise, hence we have:

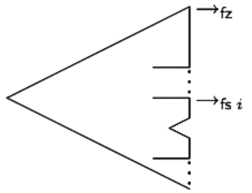
$$\begin{aligned}&\textbf{partial}\ \mathsf{vl}\ \mathsf{fz} \Rightarrow \text{‘}\mathsf{1}\text{’}\\ &\textbf{partial}\ \mathsf{vl}\ (\mathsf{fs}\ i) \Rightarrow \text{‘}\mathsf{0}\text{’}\end{aligned}$$

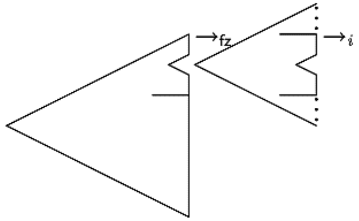
In the case of $\mathsf{wk}\ a$ the situation is reversed, there is no choice if $i = \mathsf{fz}$ and otherwise we recur structurally:

$$\begin{aligned}&\textbf{partial}\ (\mathsf{wk}\ a)\ \mathsf{fz} \Rightarrow \text{‘}\mathsf{0}\text{’}\\ &\textbf{partial}\ (\mathsf{wk}\ a)\ (\mathsf{fs}\ i) \Rightarrow \mathsf{wk}\ (\textbf{partial}\ a\ i)\end{aligned}$$

The case of local definitions $\mathsf{def}\ f\ a$ corresponds to the chain rule in Calculus. An i- hole in an element of $\mathsf{def}\ f\ a$ is either a $(\mathsf{fs}\ i)$ hole in the outer f, or it is a hole in f for the defined variable fz together with an i-hole in some a. More formally we have:

$$\begin{aligned}&\textbf{partial}\ (\mathsf{def}\ f\ a)\ i\\ &\Rightarrow\ \text{‘}\mathsf{plus}\text{’}\ (\mathsf{def}\ (\textbf{partial}\ f\ (\mathsf{fs}\ i))\ a)\ \ (\text{‘}\mathsf{times}\text{’}\ (\mathsf{def}\ (\textbf{partial}\ f\ \mathsf{fz})\ a)\ (\textbf{partial}\ a\ i))\end{aligned}$$



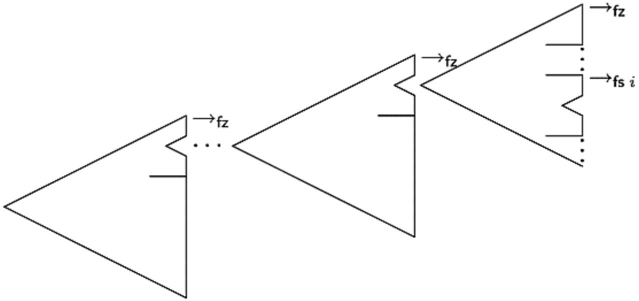


The case for initial algebras 'mu' f has no counterpart in calculus. However, it can be derived using the chain rule above: we know that 'mu' f is isomorphic to def f ('mu' f). Now using the chain rule we arrive at

'plus' (def (**partial** f (fs i)) ('mu' f))
('times' (def (**partial** f fz) ('mu' f)) (**partial** ('mu' f) i))

This expression is recursive in **partial** ('mu' f) i hence we obtain the formal derivative by taking the initial algebra of it, recording the contexts for a sequence of internal steps through the tree, terminated by the node with the external hole:

$$\textbf{partial}\,(\text{‘mu’}\,f)\;i \;\Rightarrow\; \text{‘mu’}\,(\text{‘plus’}\,(\mathsf{wk}\,(\mathsf{def}\,(\textbf{partial}\,f\,(\mathsf{fs}\;i))\,(\text{‘mu’}\,f)))$$
$$(\text{‘times’}\,(\mathsf{wk}\,(\mathsf{def}\,(\textbf{partial}\,f\;\mathsf{fz})\,(\text{‘mu’}\,f)))\;\mathsf{vl}))$$



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 ‘mu’(‘plus’(wk**list**)(‘times’vl(wkvl))) or reexpressed in a more standard notation $\mu X.(\textbf{list}\ A)+A\times X$, which can be easily seen to correspond to lists with a hole for A.

We summarize the definition of **partial**:

```
partial a i ⇐ rec a
  partial a i ⇐ case a
    partial vl i ⇐ case i
      partial vl fz ⇒ '1'
      partial vl (fs i) ⇒ '0'
    partial (wk a) i ⇐ case i
      partial (wk a) fz ⇒ '0'
      partial (wk a) (fs i) ⇒ wk (partial a i)
    partial '0' i ⇒ '0'
    partial ('plus' a b) i ⇒ 'plus' (partial a i) (partial b i)
    partial '1' i ⇒ '0'
    partial ('times' a b) i  ⇒
       'plus' ('times' (partial a i) b) ('times' a (partial b i))
    partial (def f a) i  ⇒
       'plus' (def (partial f (fs i)) a) ('times' (def (partial f fz) a) (partial a i))
    partial ('mu' f) i ⇒ 'mu' ('plus' (wk (def (partial f (fs i)) ('mu' f)))
                                ('times' (wk (def (partial f fz) ('mu' f))) vl))
```

Exercise 17. Calculate (by hand) the derivative of rose trees, i.e. the value of **partial fz rt**

6.2 Generic plugging

To convince ourselves that the definition of derivatives as one hole contexts given above is correct we derive[10] a generic version of the generic plugging operation:

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

That is given an element x of a partial derivative of a at i we can fill the hole with an element of the corresponding type of the telescope, obtaining an element of a.

We construct **gplug** by recursion over x, however, unlike in the previous examples, which were completely data driven we have to analyse the type directly, i.e. we have to invoke $\underline{\text{case}}\ a$. We discuss the cases:

variable

$$\begin{array}{l}\mathbf{gplug}\ \mathsf{vl}\ \mathsf{fz}\ \mathsf{void}\ y \Rightarrow y \\ \mathbf{gplug}\ \mathsf{vl}\ (\mathsf{fs}\ i)\ x\ y \Leftarrow \underline{\text{case}}\ x\end{array}$$

If the index is fz the argument y is the filler we are looking for, otherwise the derivative is the empty type and we eliminate it by a vacuous case analysis.

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

weakening

$$\begin{aligned}\mathbf{gplug}\ (\mathsf{wk}\ a)\ \mathsf{fz}\ x\ y\ &\Leftarrow\ \underline{\mathsf{case}}\ x\\ \mathbf{gplug}\ (\mathsf{wk}\ a)\ (\mathsf{fs}\ i)\ (\mathsf{pop}\ x)\ (\mathsf{pop}\ y)\ &\Rightarrow\ \mathsf{pop}\ (\mathbf{gplug}\ a\ i\ x\ y)\end{aligned}$$

This is in some way dual to the previous case: if the index is fz we have the empty derivative, otherwise we recur.

constant types

$$\begin{aligned}\mathbf{gplug}\ \text{‘0’}\ i\ x\ y\ &\Leftarrow\ \underline{\mathsf{case}}\ x\\ \mathbf{gplug}\ \text{‘1’}\ i\ x\ y\ &\Leftarrow\ \underline{\mathsf{case}}\ x\end{aligned}$$

are easy because impossible, since the derivative is the empty type.

disjoint union

$$\begin{aligned}\mathbf{gplug}\ (\text{‘plus’}\ a\ b)\ i\ (\mathsf{inl}\ xa)\ y\ &\Rightarrow\ \mathsf{inl}\ (\mathbf{gplug}\ a\ i\ xa\ y)\\ \mathbf{gplug}\ (\text{‘plus’}\ a\ b)\ i\ (\mathsf{inr}\ xb)\ y\ &\Rightarrow\ \mathsf{inr}\ (\mathbf{gplug}\ b\ i\ xb\ y)\end{aligned}$$

the injections are just carried through.

Product

$$
\begin{aligned}
&\mathbf{gplug}\,(\text{`}\mathsf{times}\text{'}\ a\ b)\ i\ (\mathsf{inl}\ (\mathsf{pair}\ xa\ xb)) \ \Rightarrow\ \mathsf{pair}\ (\mathbf{gplug}\ a\ i\ xa\ y)\ xb \\
&\mathbf{gplug}\,(\text{`}\mathsf{times}\text{'}\ a\ b)\ i\ (\mathsf{inr}\ (\mathsf{pair}\ xa\ xb)) \ \Rightarrow\ \mathsf{pair}\ xa\ (\mathbf{gplug}\ b\ i\ xb\ y)
\end{aligned}
$$

The derivative records the information in which component of the pair we can find the hole.

Local definition

$$
\begin{aligned}
&\mathbf{gplug}\,(\mathsf{def}\ f\ a)\ i\ (\mathsf{inl}\ (\mathsf{push}\ x))\ y \ \Rightarrow\ \mathsf{push}\ (\mathbf{gplug}\ f\ (\mathsf{fs}\ i)\ x\ (\mathsf{pop}\ y)) \\
&\mathbf{gplug}\,(\mathsf{def}\ f\ a)\ i\ (\mathsf{inr}\ (\mathsf{pair}\ (\mathsf{push}\ x)\ q)\ y \ \ \Rightarrow \\
&\qquad\qquad\qquad \mathsf{push}\ (\mathbf{gplug}\ f\ \mathsf{fz}\ x\ (\mathsf{top}\ (\mathbf{gplug}\ a\ i\ q\ y)))
\end{aligned}
$$

In the first case the hole is in top-level (f) tree but not at the first variable, which is used in the definition. In the 2nd case the hole is in a subtree (a) which means we have to plug the hole there and then use the result to plug a hole in the top-level tree.

Exercise 18. Complete (using pen and paper) the definition of **gplug** by implementing the case for mu.

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:

$$\underline{\text{data}}\ \frac{A : \star \quad a : A}{\mathsf{Minus}\ A\ a : \star}\ \underline{\text{where}}\ \frac{a' : A \quad na : \mathbf{Not}\,(a = a')}{\mathsf{minus}\ a\ na : \mathsf{Minus}\ A\ a'}$$

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

$$\underline{\text{let}}\ \frac{m : \mathsf{Minus}\ A\ a}{\mathbf{emb}\ m : A} \quad \begin{array}{l} \mathbf{emb}\ m \Leftarrow \underline{\mathsf{case}}\ m \\ \quad \mathbf{emb}\,(\mathsf{minus}\ a'\ na) \Rightarrow a' \end{array}$$

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

$$\underline{\text{data}}\ \frac{a; a' : A}{\mathsf{MinusV}\ a\ a'}$$

$$\underline{\text{where}}\quad \frac{}{\mathsf{same}\ a\ :\ \mathsf{MinusV}\ a\ a}\qquad \frac{m\ :\ \mathsf{Minus}\ A\ a}{\mathsf{other}\ m\ :\ \mathsf{MinusV}\ a\ (\mathbf{emb}\ m)}$$

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

$$\underline{\text{let}}\quad \frac{a, a'\ :\ A \quad eq\ :\ \mathsf{Dec}\,(a = a')}{\mathbf{minusV'}\ a\ a'\ eq\ :\ \mathsf{MinusV}\ a\ a'}$$

$$\begin{array}{l} \mathbf{minusV'}\ a\ a'\ eq\ \Leftarrow\ \underline{\text{case}}\ eq \\ \quad \mathbf{minusV'}\ a\ a\ (\mathsf{yes\ refl})\ \Rightarrow\ \mathsf{same}\ a \\ \quad \mathbf{minusV'}\ a\ a'\ (\mathsf{no}\ f)\ \Rightarrow\ \mathsf{other}\ (\mathsf{minus}\ a'\ f) \end{array}$$

$$\underline{\text{let}}\quad \frac{eqA\ :\ \mathbf{DecEq}\ A \quad a, a'\ :\ A}{\mathbf{minusV}\ eqA\ a\ a'\ :\ \mathsf{MinusV}\ a\ a'}$$

$$\mathbf{minusV}\ eqA\ a\ a'\ \Rightarrow\ \mathbf{minusV'}\ a\ a'\ (eqA\ a\ a')$$

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

$$\underline{\text{let}}\ \ \frac{P : S \to \star \quad sp : \mathsf{Sigma}\,S\,P}{\mathbf{derivP}\ P\ sp : \star}$$

$$\begin{aligned}
\mathbf{derivP}\ P\ sp\ &\Leftarrow\ \underline{\text{case}}\ sp\\
\mathbf{derivP}\ P\ (\mathsf{tup}\ s\ p)\ &\Rightarrow\ \mathsf{Minus}\ (P\ s)\ p
\end{aligned}$$

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

$$\underline{\text{let}}\ \ \frac{C : \mathsf{UCont}}{\mathbf{derivC}\ C : \mathsf{UCont}}$$

$$\begin{aligned}
\mathbf{derivC}\ C\ &\Leftarrow\ \underline{\text{case}}\ C\\
\mathbf{derivC}\ (\mathsf{ucont}\ S\ P)\ &\Rightarrow\ \mathsf{ucont}\ (\mathsf{Sigma}\ S\ P)\ (\mathbf{derivP}\ P)
\end{aligned}$$

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:

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

Given a decidable unary container we can define the function **uplug** which given an element of the extension of the derivative of a container $x : \mathsf{UExt}(\mathbf{derivC}\,C)\,X$

and an element $y : X$ we can plug the hole with y thus obtaining an element of $\mathsf{UExt}\ C\ X$:

$$\underline{\text{let}}\ \frac{\begin{array}{l} eq : \mathsf{DecEq}\ A \quad a : A \\ f : \mathsf{Minus}\ A\ a \to X \\ x : X \quad a' : A \end{array}}{\mathbf{mplug}\ eq\ a\ f\ x\ a' : X}$$

```
$\mathbf{mplug}\ eq\ a\ f\ x\ a' \Leftarrow \underline{\text{view}}\ \mathbf{minusV}\ eq\ a\ a'$
 $\mathbf{mplug}\ eq\ a\ f\ x\ a \Rightarrow x$
 $\mathbf{mplug}\ eq\ a\ f\ x\ (\mathbf{emb}\ m) \Rightarrow f\ m$
```

$$\underline{\text{let}}\ \frac{\begin{array}{l} C : \mathsf{UCont} \quad d : \mathsf{DecUCont}\ C \\ x : \mathsf{UExt}\ (\mathbf{derivC}\ C)\ X \quad y : X \end{array}}{\mathbf{uplug}\ C\ d\ x\ y : \mathsf{UExt}\ C\ X}$$

```
$\mathbf{uplug}\ C\ d\ x\ y \Leftarrow \underline{\text{case}}\ C$
 $\mathbf{uplug}\ (\mathsf{ucont}\ S\ P)\ d\ x\ y \Leftarrow \underline{\text{case}}\ d$
  $\mathbf{uplug}\ (\mathsf{ucont}\ S\ P)\ (\mathsf{decUCont}\ decP)\ x\ y \Leftarrow \underline{\text{case}}\ x$
   $\mathbf{uplug}\ (\mathsf{ucont}\ S\ P)\ (\mathsf{decUCont}\ decP)\ (\mathsf{uext}\ sp\ f)\ y \Leftarrow \underline{\text{case}}\ sp$
    $\mathbf{uplug}\ (\mathsf{ucont}\ S\ P)\ (\mathsf{decUCont}\ decP)\ (\mathsf{uext}\ (\mathsf{tup}\ s\ p)\ f)\ y$
              $\Rightarrow \mathsf{uext}\ a\ (\mathbf{mplug}\ (decP\ a)\ b\ f\ y)$
```

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

$$\underline{\text{let}}\ \frac{C : \mathsf{Cont}\ n \quad i : \mathsf{Fin}\ n}{\mathbf{partialC}\ C\ i : \mathsf{Cont}\ n}$$

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 very, 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 work shows that it is possible to extend both the syntactic and the semantic approach to capture families of types, see [34, 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* [10] 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 Colloqium (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 Mor-

ris. Indexed containers. Manuscript, available online, February 2006.

9. Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Generic Programming*, 2003. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloss Dagstuhl, July 2002.

10. Thorsten Altenkirch and Conor McBride. Towards observational type theory. Manuscript, available online, February 2006.

11. Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.

12. Thorsten Altenkirch, Conor McBride, and Peter Morris. Code for generic programming with dependent types, 2007. `http://www.e-pig.org/downloads/GPwDT`.

13. Thorsten Altenkirch and Tarmo Uustalu. Normalization by evaluation for $\lambda^{\to 2}$. In *Functional and Logic Programming*, number 2998 in LNCS, pages 260 – 275, 2004.

14. Andres Loeh, Johan Jeuring (editors); Dave Clarke, Ralf Hinze, Alexey Rodriguez, Jan de Wit. Generic Haskell User's Guide – Version 1.42 (Coral). Technical Report UU-CS-2005-004, Institute of Information and Computing Sciences, Utrecht University, 2005.

15. Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic Programming—An Introduction. In S. Doaitse Sweierstra, Pedro R. Henriques, and José N. Oliveira, editors, *Advanced Functional Programming, Third International Summer School (AFP '98); Braga, Portugal*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1998.

16. M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs

in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003.

17. Karl Crary, Stephanie Weirich, , and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.
18. P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. *Typed Lambda Calculi and Applications*, 1581:129–146, 1999.
19. P. Dybjer and A. Setzer. Indexed induction–recursion. *Journal of Logic and Algebraic Programming*, 66(1):1–49, 2006.
20. Peter Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. CUP, 1991.
21. Ralf Hinze. Generic programs and proofs. Habilitationsschrift, Universitt Bonn, 2000.
22. Ralf Hinze and Andres Löh. "Scrap Your Boilerplate" Revolutions. In Tarmo Uustalu, editor, *Mathematics of Program Construction, 2006*, volume 4014 of *LNCS*, pages 180–208. Springer-Verlag, 2006.
23. Ralf Hinze, Andres Löh, 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.
24. Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.
25. Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, Netherlands, September 2004.
26. Zhaohui Luo and Robert Pollack. LEGO Proof Development System: User's Man-

ual. Technical Report ECS-LFCS-92-211, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.
27. 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.
28. Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis·Napoli, 1984.
29. Conor McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. Available online, 2001.
30. Conor McBride. Epigram, 2004. `http://www.e-pig.org/`.
31. 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.
32. Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1), 2004.
33. Fred McBride. *Computer Aided Manipulation of Symbols*. PhD thesis, Queen's University of Belfast, 1970.
34. Peter Morris, Thorsten Altenkirch, and Neil Ghani. Constructing strictly positive families. In *The Australasian Theory Symposium (CATS2007)*, January 2007.
35. 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.

36. H. Pfeifer and H. Rueß. Polytypic abstraction in type theory. In Roland Backhouse and Tim Sheard, editors, *Workshop on Generic Programming (WGP'98)*. Dept. of Computing Science, Chalmers Univ. of Techn. and Göteborg Univ., June 1998.
37. Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's type theory: an introduction*. Oxford University Press, 1990.
38. 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.
39. Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of POPL '87*. ACM, 1987.
40. Stephanie Weirich. RepLib: A library for derivable type classes. In Andres Löh, editor, *Proceedings of the ACM Haskell Workshop, 2006*, 2006.