Inductive sets, the algebraic way

Inductive sets, the algebraic way

by

Rik van Geldrop and Jaap van der Woude

Contents

1 Introduction

Inductively defined sets frequently occur in computing science. In the following we will see that it is not the set alone that is important, but the set *together* with some special functions —called 'constructors'. The pair (set, constructors) is called an algebra. Between two algebras mappings can be defined —called 'homomorphisms' and those mappings together with some knowledge about the mapping domain establishes a lot of algebraic properties which are of importance.

Below we will explain:
- what exactly is meant by an inductively defined set
- what is an algebra and what are homomorphisms
- what does it mean to be an initial algebra and what consequences initiality has.

We will do this by taking an example of an inductively defined set and try to give sufficient evidence for the reader to tackle another similarly defined set in a same way.

In order to keep this note self-contained we start with some mathematical and notational preliminaries.

2 Preliminaries

In this note functions are treated as first class citizens, they can be used to construct objects and they can be the result of constructions. Below we give a brief overview of the prerequisites and the notational issues.

We consider typed total functions from source type A to result type B, i.e. $f \in A \to B$ is defined on all elements of A and with every element $a \in A$ precisely one element $b \in B$ is given as the image of a under the function f. The resulting image b is denoted by $f.a$. The set A is called the domain $dom.f$ of f, the set B is called the codomain of f and it contains the set of all resulting images of f which is called the range $rng.f$ of f.

Whenever we deem appropriate we give a graphical representation of a function by way of a horizontal or a vertical arrow with the types on the extrema of the arrow and the name of the function as a label, e.g.

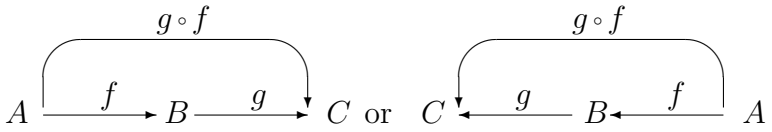
$$A \xrightarrow{\quad f \quad} B \qquad \text{or} \qquad B \xleftarrow{\quad f \quad} A$$

For any set A we have an identity function id_A given by $id_A.a = a$. When A can be inferred from the context we omit the subscript.

The composition of functions is allowed if the type in between is the same. So, for $f \in A \to B$ and $g \in B \to C$ the function $g \circ f \in A \to C$ exists and is defined by

$$(g \circ f).a = g.(f.a)$$

Function-composition is depicted as



It may happen that a function composition yields an identity function as its result, e.g. $g \circ f = id$. In such case the function g is called a left-inverse of f and the function f is called a right-inverse of g. When we are given that $g \circ f = id$ and $f \circ g = id$ then the function f is the inverse function of g, and vice-versa. Being a (left-, right-) inverse function puts its demand on the typing of the function: for g a (left-, right-) inverse of $f \in A \to B$, its type is $B \to A$. When the functions f and g are each others inverses it is said that the sets A and B are "isomorphic" (notation $A \cong B$).

For all practical purposes the result type is not very relevant, the range of the function is sufficient. Often functions given by expressions may be composed if the typing allows us to do so on the level of points, i.e. $rng.f \subseteq dom.g$.

Another relict of this lenient attitude towards the type of the results is the way we treat equality of functions:

Definition 1 [**Extensionality**] *For all functions f and g with the same domain:*

$$f = g \quad \equiv \quad \langle \forall x :: f.x = g.x \rangle$$

□

As can be seen from the definition, the result type plays no role!

Exercises. Prove the following laws.

1. id is the unit of function composition, i.e. $f \circ id = id \circ f = f$ for all f.

2. Function composition is associative.

2.1 Type constructions

Next we deal with the way to construct types and functions on them. We start with a unit type and show how to construct new types from existing ones.

2.1.1 The unit type

The smallest domain we consider for functions is a singleton. Because singleton sets are isomorphic we use a generic name, viz. **1**.
The element of a singleton set is unique and we will denote it by $*$. So $\mathbf{1} = \{*\}$.

There is only one function from **1** to **1**, viz. id.

2.1.2 Function space

For given sets A and B, the function space $A \to B$ is the set of total functions from A to B. If $f \in A \to B$ then we say that f has type $A \to B$.

There is a one-to-one correspondence between elements and functions as follows:

Property 2 $\quad X \;\cong\; \mathbf{1} \to X \qquad$ *for each set* X

Proof. We have to show the existence of two functions, say $\varphi \in X \to (\mathbf{1} \to X)$ and $\psi \in (\mathbf{1} \to X) \to X$, such that $\varphi \circ \psi = id$ and $\psi \circ \varphi = id$.
Candidates for φ and ψ are derived from their typing. As follows.
For φ: let x be an arbitrary element of X, then

$$\varphi.x \in \mathbf{1} \to X$$

i.e. $\varphi.x$ is a function. In order to define $\varphi.x$ we have to come up with an X-element as the image of $(\varphi.x).*$. A straightforward choice is

$$(\varphi.x).* = x.$$

For ψ: let f be an arbitrary element of $\mathbf{1} \to X$, then we find in a similar way:

$$\psi.f = f.*$$

The proof that $\varphi \circ \psi = id$ and $\psi \circ \varphi = id$ is left as an exercise.
□

2.1.3 (Cartesian) Product

For given sets A and B, the (cartesian) product $A \times B$ is defined to be the set

$$\{(a, b) \mid a \in A \wedge b \in B\}$$

The product comes with two *projections* denoted by π_1 and π_2:

$$\pi_1 \in A \times B \to A, \qquad \pi_1.(a, b) = a$$
$$\pi_2 \in A \times B \to B, \qquad \pi_2.(a, b) = b$$

In our graphical notation, a product looks like

$$A \xleftarrow{\pi_1} A \times B \xrightarrow{\pi_2} B$$

Exercise. Prove that for all types A, B and C the following holds

$$A \times B \to C \;\cong\; A \to (B \to C)$$

Thanks to the projections we get the following new functions out of existing ones.

Definition 3 [Split] *Given two functions $f \in A \to B$ and $g \in A \to C$ we define $f \vartriangle g$ (pronounce "f split g") by*

$$f \vartriangle g \in A \to B \times C$$
$$(f \vartriangle g).a = (f.a, g.a)$$

□

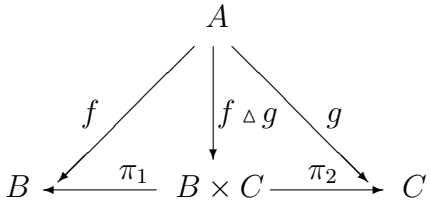
The existence and the unicity of the function $f \vartriangle g$ for every pair (f, g) of functions as above may be taken as the formal definition of the cartesian product. In this note we will not pursue such a fundamental treatment, but we do borrow the following characterization of the product valued maps that result from pairs of functions.

Property 4 [split-characterization]
Let $f \in A \to B$, $g \in A \to C$ and $h \in A \to B \times C$, then

$$h = f \vartriangle g$$
$$\equiv$$
$$\pi_1 \circ h = f \quad \wedge \quad \pi_2 \circ h = g$$

□

In a graphical notation, $f \vartriangle g$ is depicted as



From property 4 we immediately obtain the following computation rule

Property 5 [split-self]

$$\pi_1 \circ (f \vartriangle g) = f \quad \wedge \quad \pi_2 \circ (f \vartriangle g) = g$$

□

The split function in combination with the projections yields the "product" function:

Definition 6 [Product] *Given two functions $f \in A \to C$ and $g \in B \to D$ we define $f \times g$ by*

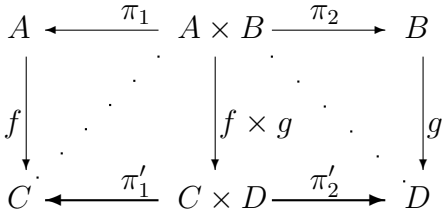
$$f \times g \in (A \times B) \to (C \times D)$$
$$(f \times g).(a, b) = (f.a, g.b)$$

or pointfree

$$f \times g = (f \circ \pi_1) \vartriangle (g \circ \pi_2)$$

□

In a graphical notation, $f \times g$ is depicted as



Exercises. Prove the following laws.

1. $(f \vartriangle g) \circ h = (f \circ h) \vartriangle (g \circ h)$
2. $(f \times g) \circ (h \vartriangle k) = (f \circ h) \vartriangle (g \circ k)$
3. $(f \times g) \circ (h \times k) = (f \circ h) \times (g \circ k)$

2.1.4 (Disjoint) Sum

For given sets A and B, the (disjoint) sum $A + B$ compares to set-union, be it that from an element of $A + B$ its origin can be inferred. To realize this distinction the sum comes with two *injections* denoted by in_1 and in_2:

$$in_1 \in A \to A + B$$
$$in_2 \in B \to A + B$$

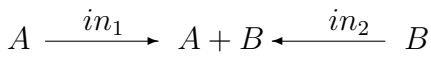
such that

$$\langle \forall a, b :: in_1.a \neq in_2.b \rangle$$

So $A + B$ can be envisaged as

$$\{in_1.a \mid a \in A\} \cup \{in_2.b \mid b \in B\}$$

In our graphical notation, a sum looks like



"Tagging" is a frequently used implementation for injections, i.e.

$$in_1.a = (0, a), \quad in_2.b = (1, b)$$
$$A + B = \{(0, a) \mid a \in A\} \cup \{(1, b) \mid b \in B\}$$

Thanks to the injections we get the following new functions out of existing ones.

Definition 7 [Case] *Given two functions $f \in A \to C$ and $g \in B \to C$ we define $f \triangledown g$ (pronounce "f case g") by*

$$f \triangledown g \in A + B \to C$$

$$(f \triangledown g).(in_1.a) = f.a$$
$$(f \triangledown g).(in_2.b) = g.b$$

□

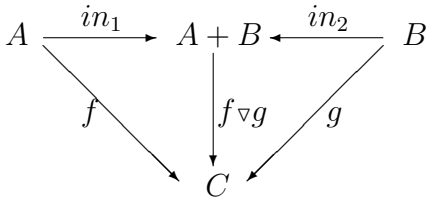
The existence and the unicity of the function $f \triangledown g$ for every pair (f, g) of functions as above may be taken as the formal definition of the disjoint sum. The unicity takes care of the "minimality" of $A + B$ in the sense that there is no junk left in the combination of A and B, while the existence (for every C) guarantees the disjointness of the images of A and B in the sum. In this note we will not pursue such a fundamental treatment, but we do borrow the following characterization of the maps on sum domains that result from pairs of functions.

Property 8 [case-characterization]
Let $f \in A \to C$, $g \in B \to C$ and $h \in A + B \to C$, then

$$\begin{array}{rl} & h = f \triangledown g \\ \equiv & \\ & h \circ in_1 = f \quad \wedge \quad h \circ in_2 = g \end{array}$$

□

In a graphical notation, $f \triangledown g$ is depicted as



From property 8 we immediately obtain the following computation rule

Property 9 [case-self]

$$(f \triangledown g) \circ in_1 = f \quad \wedge \quad (f \triangledown g) \circ in_2 = g$$

□

The case function in combination with the injections yields the "sum" function:

Definition 10 [Sum] *Given two functions $f \in A \to C$ and $g \in B \to D$ we define $f + g$ by*

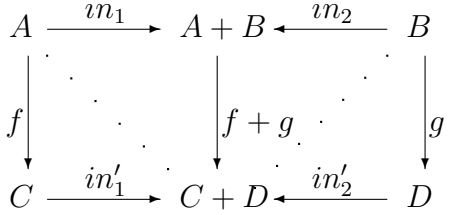
$$f + g \in (A + B) \to (C + D)$$
$$(f + g).(in_1.a) = in'_1.(f.a)$$
$$(f + g).(in_2.b) = in'_2.(g.b)$$

or pointfree

$$f + g = (in'_1 \circ f) \triangledown (in'_2 \circ g)$$

□

In a graphical notation, $f + g$ is depicted as



Exercises. Prove the following laws.

1. $h \circ (f \triangledown g) = (h \circ f) \triangledown (h \circ g)$
2. $(f \triangledown g) \circ (h + k) = (f \circ h) \triangledown (g \circ k)$
3. $(f + g) \circ (h + k) = (f \circ h) + (g \circ k)$

2.2 Auxiliary notations

Because constant functions will come up frequently we introduce a special notation for them:

Definition 11 *Let $a \in A$. Then for each set X*

$$a^\bullet \in X \to A$$
$$a^\bullet.x = a \qquad \text{for each } x \in X$$

□

For binary functions we prefer to use an infix notation. Since functions may be used as arguments to other functions we may need a prefix notation for binary function as well. Therefore we introduce the following convention

Definition 12 $a \oplus b = (\oplus).(a, b)$

□

Expressions can be moulded into functions via so-called λ-abstraction:

Definition 13 [**λ-abstraction**] *Let x be a variable of type X and e an expression of type Y, then*

$$(\lambda x \to e) \in X \to Y$$

and

$$(\lambda x \to e).a \quad = \quad e(x := a) \qquad \text{for each } a \in X$$

□

Some examples of λ-abstractions are

$$f = \lambda x \to f.x$$

For $\oplus \in X \times Y \to Z$:

$$\begin{array}{lcll} (\lambda y \to x \oplus y) & \in & Y \to Z & \textit{for any } x \in X \\ (\lambda x \to x \oplus y) & \in & X \to Z & \textit{for any } y \in Y \\ (\lambda x \to (\lambda y \to x \oplus y)) & \in & X \to (Y \to Z) & \\ (\lambda x, y \to x \oplus y) & \in & X \times Y \to Z & \end{array}$$

The former two functions are mostly abbreviated to $(x\oplus)$ and $(\oplus y)$ respectively and are called 'sections of $\oplus$'.
□

Finally we introduce some conventions to reduce the number of parenthesis in our expressions:
- Function-application has the highest priority, e.g.
  $f \circ g.x = f \circ (g.x)$
- Further priority ordering: prio $(\times)$ > prio $(+)$ > prio $(\to)$
- Function-application associates to the left, e.g.
  $f.g.x = (f.g).x$
- The type-constructor "$\to$" associates to the right, e.g.
  $A \to B \to C = A \to (B \to C)$

□

Exercises. Prove the following properties

1. $a^\bullet \circ f = a^\bullet$
2. $f \circ a^\bullet = (f.a)^\bullet$

3 Inductive definitions

In practice, finite lists may come in a variety of ways, for instance as words over an alphabet Σ, or as finite sequences of Σ-elements, or as Σ-valued functions on finite sets, etc.

For several —mathematical— reasons an algebraic formalization of the concept of lists without reference to a specific use or application domain is beneficial. The main reason for this is that a generalization of the list-concept to arbitrary constructions relieves us from the burden to prove every property for every specific construction over and over again. The mathematical disciplines of category theory and universal algebra aim at such treatments of generalizations. Although these fields are beyond the scope of this course we nevertheless intend to use some of their results in dealing with inductive datatypes.

Definition 14 [$\mathbb{L}_\alpha$, **cons-lists over** α] *Let α be an arbitrary set. The set $\mathbb{L}_\alpha$ of finite (cons-)lists over α is the smallest set X such that*

(basis) $\quad * \in X \qquad$ *(the empty list is in X)*
(compound) $\quad a \in \alpha \wedge x \in X \quad \Rightarrow \ (a, x) \in X$
□

This may be paraphrased as: $\mathbb{L}_\alpha$ is the least solution of the (set theoretic) equation

$$X :: \quad \{*\} \ \cup \ \alpha \times X \quad \subseteq \quad X \tag{15}$$

From fixpoint theory we know that (15) has a unique least solution. Moreover there is a construction for this least solution, the so-called Kleene construction:

$$\mathbb{L}_\alpha \quad = \quad \langle \bigcup n : n \in \mathbb{N} : \alpha^n \rangle$$

where α^n is an abbreviation for $\overbrace{\alpha \times \cdots \times \alpha}^{n}$ with associative set theoretic product operator $(\times)$; $\alpha^0 = \{*\}$.

As mentioned before, the concept of finite lists occurs in several places with different notation and naming conventions. As an example: in automata theory we encounter the words over alphabet Σ, usually denoted by Σ^*. The empty word is often written as ε, while the symbols within the word are just juxtaposed as in abc instead of $(a, (b, (c, *)))$.
Another example is from functional programming languages. There the cons-lists are often denoted using a language construct like

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

Here `Empty` stands for $*$ and `Cons a x` for (a, x). In practice the cons-lists are denoted by shorthands, viz. `[]` for the empty list and $a : x$ for the list with head a and tail x; short lists are often denoted by enumeration, viz. $[a, b, c]$ for $(a, (b, (c, *)))$. In the sequel we will adopt this shorthand notation.

The set $\mathbb{L}_\alpha$ satisfies the following four properties which will turn out be to useful later on.

3.1 No junk property

Because $\mathbb{L}_\alpha$ is the least solution, there are no other elements of $\mathbb{L}_\alpha$ than those that are finitely constructed from `[]` using (:). So every nonempty list is of the form $a : x$ for some $a \in \alpha$ and $x \in \mathbb{L}_\alpha$.

3.2 No confusion property

The constructions are unique because two tuples are equal if and only if they have the same length and are equal element-wise. Thus every list is constructed uniquely (and it *is* constructed indeed, see "no junk".)

A consequence of the "no confusion" property is that (total) functions over $\mathbb{L}_\alpha$ may be given by defining them on the two element shapes $[]$ and $a : x$. In the absence of the "no confusion" property a relation rather than a function would be defined: functionality might be lost. In the absence of the "no junk" property totality would be lost.

3.3 Structural induction

In order to prove $\langle \forall x : x \in \mathbb{L}_\alpha : P.x \rangle$, for some predicate P on $\mathbb{L}_\alpha$, it suffices to show that P is an invariant of the construction, i.e.

$$P.[] \quad \wedge \quad \langle \forall a, y : a \in \alpha \wedge y \in \mathbb{L}_\alpha : P.y \Rightarrow P.(a : y) \rangle$$

□

The proof of the validity of $P.[]$ is the base of the proof by structural induction. The constructive step in the proof is to show that $P.(a : y)$ holds in a context where $P.y$ holds. $P.y$ is called the induction hypothesis and it should always be mentioned in the context of a proof of $P.(a : y)$, even if it isn't used.

Remark. The above principle of structural induction may also be appreciated as standard mathematical induction with respect to the length of the list.
The induction principle is mainly used as a verification means, however as a construction tool it is very powerful too!

□

3.4 Universality property

The set $\mathbb{L}_\alpha$ together with the distinguished element $[] \in \mathbb{L}_\alpha$ and the operator $(:) \in \alpha \times \mathbb{L}_\alpha \to \mathbb{L}_\alpha$ is a so-called structured set and will be denoted by the triple $(\mathbb{L}_\alpha, [], (:))$.

We can link the triple $(\mathbb{L}_\alpha, [], (:))$ to any similarly structured set $(E, e, \oplus)$ -with $e \in E$ and $\oplus \in \alpha \times E \to E$- by stating the existence of a unique structure preserving function $\varphi \in \mathbb{L}_\alpha \to E$ such that

$$\varphi.[] = e \quad \wedge \quad \varphi.(a : x) = a \oplus \varphi.x \quad \text{for every } a \in \alpha \text{ and } x \in \mathbb{L}_\alpha \tag{16}$$

Indeed, the existence of φ as a total function follows from the "no junk" and the "no confusion" property while the adjective "structure preserving" is algebraic terminology for list functions of shape (16). For the uniqueness of φ prove, with induction, that $\psi = \varphi$ for each ψ that satisfies (16).

The property is called the universality property. Universality gives the cons-lists its importance as a datatype with algebraic and manipulative possibilities —as we will see in the next sections.

4 Algebras and their homomorphisms

The terms "structured set" and "similarly structured set", mentioned in the universality property, suggest that there is more to be known about an inductive definition than was stated before. What exactly is meant by a "structure", and, when are two structures "similar", and, do two different inductive definitions lead to "different structures"? To answer these questions we need a formalization of the notion of structure.

The notion of structure we have in mind is known in mathematics as an F-algebra. We will introduce F-algebra's by example and we will do so by starting from the inductive definition (23) and the structured set $(\mathbb{L}_\alpha, [\,], (:))$ that comes with it.

The term "structure" in structured set $(\mathbb{L}_\alpha, [\,], (:))$ clearly refers to the operators/constructors $[\,]$ and $(:)$. The empty list $[\,]$ may be thought of as an operation $[\,]^\bullet \in \mathbf{1} \to \mathbb{L}_\alpha$ (recall property 2), it takes as arguments the elements of the unit type $\mathbf{1}$ and it results in a list, the empty list. The cons $(:) \in \alpha \times \mathbb{L}_\alpha \to \mathbb{L}_\alpha$ is an operator that takes an argument from the known type α and a list and results in a list. Because of the common result type these two operations can be combined into one operator that is defined on the disjoint sum of all argument types (see def 7), i.e.

$$[\,]^\bullet \triangledown (:) \quad \in \quad \mathbf{1} + \alpha \times \mathbb{L}_\alpha \to \mathbb{L}_\alpha$$

In a similarly structured set say $(E, e, \oplus)$ with $e \in E$ and $\oplus \in \alpha \times E \to E$ the operators may be combined too:

$$e^\bullet \triangledown \oplus \quad \in \quad \mathbf{1} + \alpha \times \ E \to \ E$$

The type of the system of operators in $(\mathbb{L}_\alpha, [\,], (:))$ as well as in $(E, e, \oplus)$ satisfies a common pattern, i.e.

$$F.X \to X \qquad\qquad \text{with } F.X = \mathbf{1} + \alpha \times X$$

Note that the function F maps types to types. In order to distinguish F on types from all other functions we will call it a "functor".
This common pattern will be our formalization of the notion "structure" in our current example.

Definition 17 [F**-algebra for** $F.X = 1 + \alpha \times X$]
An F-algebra for functor F is a set E with an operator of type $F.E \to E$.
In our example, where $F.X = \mathbf{1} + \alpha \times X$, an F-algebra is a set E together with an operator $e^\bullet \triangledown \oplus \ \in \ \mathbf{1} + \alpha \times E \to E$.
□

In an F-algebra, the underlying set and the result type of the operator are the same. Therefore we sometimes take the freedom to denote an F-algebra by its operator part only.

In our graphical notation an F-algebra for $F.X = \mathbf{1} + \alpha \times X$ is presented by

Now that we have formalized the notion of structure by means of F-algebras, we introduce the concept of F-homomorphism (a special mapping between F-algebras for the same F) to formalize the notion of structure preserving functions.

Definition 18 [**F-homomorphism for $F.X = \mathbf{1} + \alpha \times X$**]
Let $(E, e^\bullet \triangledown \oplus \ \in \ \mathbf{1} + \alpha \times E \to E)$ and $(G, g^\bullet \triangledown \otimes \ \in \ \mathbf{1} + \alpha \times G \to G)$ be two F-algebras. Then function $h \in E \to G$ is an F-homomorphism from $(E, e^\bullet \triangledown \oplus)$ to $(G, g^\bullet \triangledown \otimes)$ whenever it satisfies the following scheme

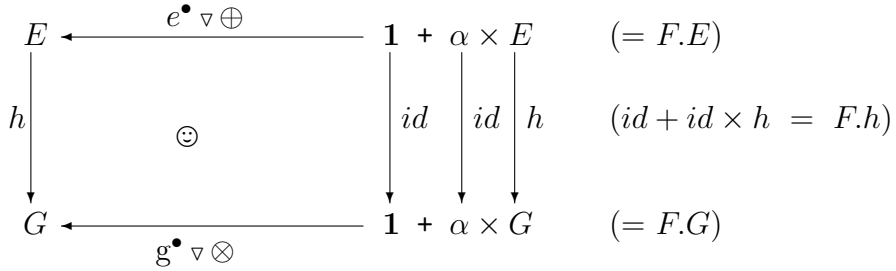
$$\begin{array}{lcll} h.e & = & g & \\ h.(a \oplus u) & = & a \otimes h.u & \text{for all } a \in \alpha,\ u \in E \end{array}$$

Or, to make the link with F more explicit,

$$h \circ (e^\bullet \triangledown \oplus) \quad = \quad (g^\bullet \triangledown \otimes) \circ F.h$$

see diagram below.
□

In our graphical notation an F-homomorphism for $F.X = \mathbf{1} + \alpha \times X$ looks like



Note that in the diagram all sets different from E, to wit $\mathbf{1}$ and α, are mapped by the identity function id, while all occurrences of E are mapped by h. I.e. the diagram is composed of the two F-algebras, h and the function $id_{\mathbf{1}} + id_\alpha \times h$ $(= F.h)$.

To see that the diagram is an adequate means to (re)construct the defining scheme of an F-homomorphism we give the following instructions for reading this diagram.

> Start from $\mathbf{1} \ + \ \alpha \times E$ and follow the paths to G.
> This can be done in two ways: via E and via $\mathbf{1} \ + \ \alpha \times G$ and then the results should be the same. It is said that "the diagram commutes" which is graphically depicted as ☺.

- For element * from source $\mathbf{1}$ we obtain:

via E : $h.e$
via $1 \texttt{ + } \alpha \times G$: g

- For element (a, u) from source $\alpha \times E$ we obtain:

via E : $h.(a \oplus u)$
via $\mathbf{1} \texttt{ + } \alpha \times G$: $a \otimes h.u$

□

After this brief introduction to F-algebras and their homomorphisms we return to the special F-algebra that was mentioned in the universality property of our inductive definition, viz. $(\mathbb{L}_\alpha, \texttt{[]}^\bullet \triangledown (:) \in \mathbf{1} + \alpha \times \mathbb{L}_\alpha \to \mathbb{L}_\alpha)$.

5 Initial F-algebras

Previously, cons-lists were defined as a set, the least solution of a set theoretic equation in which the construction mechanism played a rôle, (def 15):

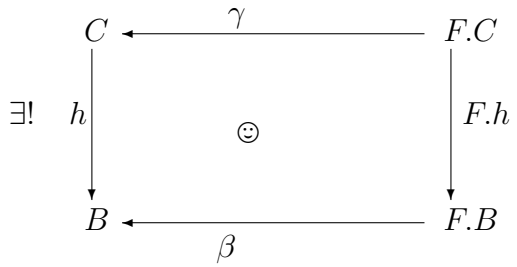
$$X \quad :: \quad \{*\} \ \cup \ \alpha \times X \ \subseteq \ X$$

In the current algebraic approach we "rewrite" the equation to an F-algebra.

$$(X, \varphi) :: \quad \varphi \in \mathbf{1} + \alpha \times X \to X$$

The least solution is then replaced by the corresponding categorical notion of initial algebra.

Definition 19 [Initial F-algebra]
Let F be a functor. An F-algebra $(C, \gamma \in F.C \to C)$ is an initial F-algebra iff for every F-algebra $(B, \beta \in F.B \to B)$ there is an unique homomorphism h from (C, γ) to (B, β). Or, in a graphical notation



□

For our example functor $F.X = \mathbf{1} + \alpha \times X$ the algebra $(\mathbb{L}_\alpha, \texttt{[]}^\bullet \triangledown (:) \in \mathbf{1} + \alpha \times \mathbb{L}_\alpha \to \mathbb{L}_\alpha)$ is an initial one, as might be clear from the universality property in section 3.4. Initiality is just a rephrasing of the universality property.

For many functors, in particular for every functor that might interest us in this course, such an initial algebra exists and it is even unique (up to isomorphism). Among all the initial algebras of $F.X = \mathbf{1} + \alpha \times X$ we choose $(\mathbb{I}\mathrm{L}_\alpha, [\,]^\bullet \triangledown (:))$ to be the canonical one for the remaining part of this note.

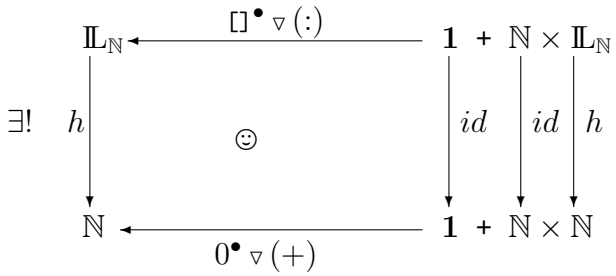
The underlying set of an initial algebra is what is called an inductive type, it has the no junk en no confusion property and it allows for structural induction. Rather than saying that $\mathbb{I}\mathrm{L}_\alpha$ is the set of cons-lists, we mention the set and its structure by saying that the cons-lists are given by the algebra $(\mathbb{I}\mathrm{L}_\alpha, [\,]^\bullet \triangledown (:) \in \mathbf{1} + \alpha \times \mathbb{I}\mathrm{L}_\alpha \to \mathbb{I}\mathrm{L}_\alpha)$.
Note that the lists themselves may be formed in other ways like adding members to the end or pasting sublists. The list per se reveals no information on its structure, while the name cons-lists does suggest it.

Inductive types or initial algebras are especially interesting because of the unique existence of homomorphisms from the initial algebra to an arbitrary algebra and such a function may be viewed as the "free" function/program resulting from the algebraic problem specification.

For example, assume we have to define a function/program which adds the elements of a list of natural numbers. Then we may recognize addition as an algebra:

$$\begin{array}{lcl} + & \in & \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ 0 & \in & \mathbb{N} \end{array}$$

i.e. $0^\bullet \triangledown (+) \in \mathbf{1} + \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ is an F-algebra for $F.X = \mathbf{1} + \alpha \times X$ where $\alpha = \mathbb{N}$. From the initiality of $(\mathbb{I}\mathrm{L}_\alpha, [\,]^\bullet \triangledown (:))$ we obtain



The induced unique homomorphism $h \in \mathbb{I}\mathrm{L}_\mathbb{N} \to \mathbb{N}$ is the required function/program:

$$\begin{array}{lcl} h.[\,] & = & 0 \\ h.(a:x) & = & a + h.x, \end{array}$$

induction completes the argument.

Similarly we may recognize algebras for counting list elements, or for determining the maximum and minimum of list elements and so on.

6 Properties of initial algebras

It is useful to have a special name/notation for homomorphisms from an initial algebra. Here we will use the name "$fold_F$" because this name is most widely used in the literature about this subject (variants of the term *fold* and *catamorphism* may also be found). The unique homomorphism from the initial algebra to an arbitrary F-algebra (B, β) will be denoted by $fold_F.\beta$. When F can be inferred from the context we omit the subscript.

The unicity of the *fold* defined on an inductive type gives rise to three easy and useful properties: characterization, computation and fusion. We will state them for our running example $F.X = \mathbf{1} + \alpha \times X$

Property 20 [*fold*-**characterization, (charn)**]
Let $h \in \mathbb{L}_\alpha \to \beta$, $e \in \beta$, $\oplus \in \alpha \times \beta \to \beta$, *then*

$$
\begin{array}{ll}
 & h = fold.(e^\bullet \triangledown \oplus) \\
\equiv & \\
 & \begin{array}{lcl} h.[\,] & = & e \\ h.(a:x) & = & a \oplus h.x \quad \textit{for all } a \in \alpha,\ x \in \mathbb{L}_\alpha \end{array}
\end{array}
$$

(Recall that the latter expresses the fact that h *is an* F*-homomorphism.)*
□

The *fold*-characterization can be used as a verification means (given h, e, $\oplus$ prove that the inductive definition of h satisfies the required recurrence pattern) but, more importantly, it may be used as a construction aid. E.g. given an inductive definition of h, construct the parameter instances e and $\oplus$ that makes h into a *fold*, i.e. construct the algebra that produces h. Another example of constructive use is given below where we derive the fusion property from the *fold*-characterization.

From property 20 we immediately obtain an explicit recursive form for $fold.(e^\bullet \triangledown \oplus)$

Property 21 [**Computation**]

$$
\begin{array}{lcl}
fold.(e^\bullet \triangledown \oplus).[\,] & = & e \\
fold.(e^\bullet \triangledown \oplus).(a:x) & = & a \oplus fold.(e^\bullet \triangledown \oplus).x
\end{array}
$$

□

For human beings this definition is not very appealing, but for computers *fold* defined in this way is a generic program, i.e. it computes whatever initial homomorphism is involved -assumed that functions are "first class citizens".

The following property gives conditions for a composition of some function with a *fold* to be a *fold* again. I.e. it gives conditions to ensure that $k \circ h$ is a *fold* whenever h is a *fold*. We prefer to derive this property as an illustration of the constructive use of the *fold*-characterization. As follows.

$$
\begin{array}{ll}
 & k \circ h = fold.(g^\bullet \triangledown \otimes) \\
\equiv & \quad \{ \text{ charn } \} \\
 & (k \circ h).[\,] = g \\
 & (k \circ h).(a : x) = a \otimes (k \circ h).x \qquad \forall a, x \\
\equiv & \quad \{ \text{ def } (\circ) \} \\
 & k.(h.[\,]) = g \\
\wedge & k.(h.(a : x)) = a \otimes k.(h.x) \qquad \forall a, x \\
\equiv & \quad \{ \ h = fold.(e^\bullet \triangledown \oplus), \text{ charn } \} \\
 & k.e = g \\
\wedge & k.(a \oplus h.x) = a \otimes k.(h.x) \qquad \forall a, x \\
\Leftarrow & \quad \{ \text{ generalize, instantiate } u := h.x \ \} \\
 & k.e = g \\
\wedge & k.(a \oplus u) = a \otimes k.u \qquad \forall a, u
\end{array}
$$

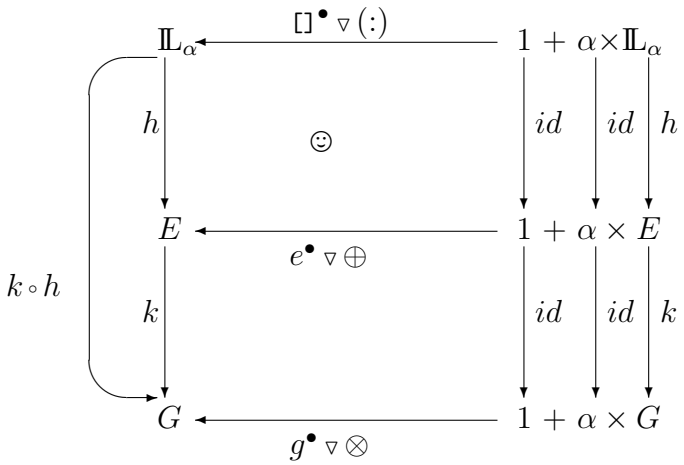
We obtain

Property 22 [Fusion]

$$
\begin{array}{ll}
 & k \circ fold.(e^\bullet \triangledown \oplus) = fold.(g^\bullet \triangledown \otimes) \\
\Leftarrow & \\
 & k.e = g \\
 & k.(a \oplus u) = a \otimes k.u \qquad \forall a, u
\end{array}
$$

□

In our graphical notation, the fusion-rule is depicted as



Note that the antecedent in the fusion-rule can be phrased as

$k \in E \to G$ is an F-homomorphism from $(E, e^\bullet \triangledown \oplus)$ to $(G, g^\bullet \triangledown \otimes)$

Resulting in the paraphrazation of fusion

k after an F-homomorphism is an F-homomorphism
if k is an F-homomorphism.

The fusion law can be used for verification as well as for construction means. The law is a basis for many program transformations.

In the next section we will illustrate (some of) the manipulative possibilities of the rules for *fold*. Here we exhibit three elementary *fold* examples:

$$\begin{array}{lcl} id & = & fold.(\texttt{[]}^\bullet \triangledown (:)\,) \\ length & = & fold.(0^\bullet \triangledown (\lambda a, v \to 1 + u)\,) \\ isempty & = & fold.(true^\bullet \triangledown false^\bullet) \end{array}$$

7 Example: run length encoding

Run length encoding is one of several methods to compress data. The idea behind this method is: replace consecutive occurrences of one and the same data item d by a pair (d, number of consecutive occurrences of d). It is understood that the best compression rate is achieved when d is paired with the maximal number of consecutive occurrences and that two consecutive replacements have different first components.

In order to illustrate our algebraic approach we will take the run length encoding problem as our example.

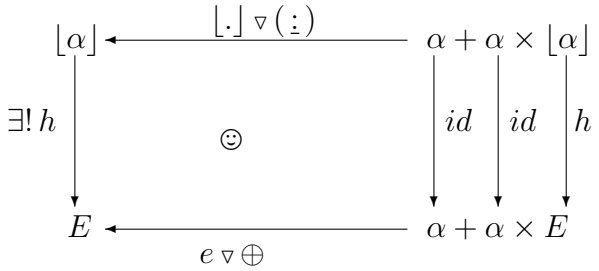
We start with the definition of the problem domain, i.e. the set of non-empty lists of data items. Because this domain slightly differs from our example-domain we also formulate (some of) its algebraic properties.

Definition 23 [non-empty cons-lists over α, denoted $\lfloor\alpha\rfloor$] *Let α be an arbitrary set. The set $\lfloor\alpha\rfloor$ of finite non-empty (cons-)lists over α is the smallest set X such that*

$$\begin{array}{lllll} \textit{(basis)} & a \in \alpha & \Rightarrow\ \lfloor a \rfloor \in X & \textit{(a singleton list is in } X\textit{)} \\ \textit{(compound)} & a \in \alpha \wedge x \in X & \Rightarrow\ a : x\ \in X & \end{array}$$

□

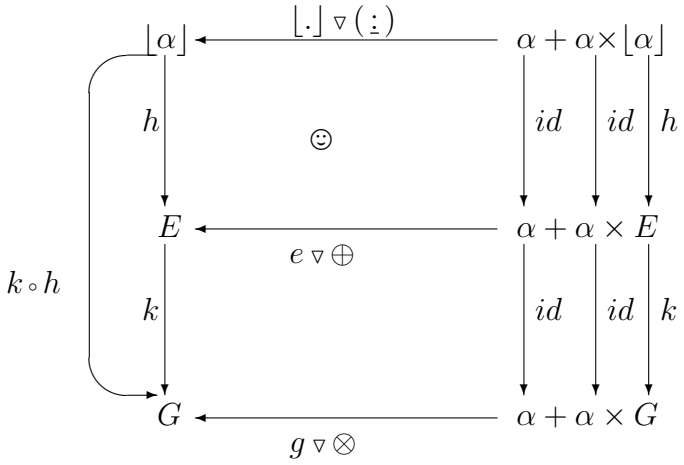
$\lfloor\alpha\rfloor$ satisfies the no junk, no confusion and the universality property and allows for structural induction. In algebraic terms, the inductive type of non-empty cons-lists over α is the underlying set of an initial algebra which we fix on $(\lfloor\alpha\rfloor, \lfloor.\rfloor \triangledown (:) \in \alpha + \alpha \times \lfloor\alpha\rfloor \to \lfloor\alpha\rfloor)$. (Here $\lfloor.\rfloor$ is shorthand for the singleton maker.) The functor F that is associated to the inductive type is given by $F.X = \alpha + \alpha \times X$, and in a graphical notation the initiality property is depicted as



The fusion-rule that comes with this functor reads

$$\begin{array}{ll} & k \circ fold.(e \triangledown \oplus) = fold.(g \triangledown \otimes) \\ \Leftarrow & \\ & k.(e.a) \ = \ g.a \\ & k.(a \oplus u) \ = \ a \otimes k.u \end{array}$$

and is depicted as



As before, the antecedent of the fusion-rule can be phrased as "k is an F-homomorphism".

Remark on notation. As mentioned before an initial algebra is unique up to F-isomorphism. For the initial algebra $(\lfloor\alpha\rfloor, \lfloor.\rfloor, (\underline{:}))$ this implies that we may consider the following isomorphic instance:

$$\begin{array}{lcl} \lfloor\alpha\rfloor & = & [\alpha]\backslash\{\texttt{[]}\} \\ \lfloor.\rfloor & = & (:\texttt{[]}) \\ (\underline{:}) & = & (:) \text{ restricted to the domain } \alpha \times \lfloor\alpha\rfloor \end{array}$$

As a consequence each $zs \in \lfloor\alpha\rfloor$ can uniquely be written as a compound $[\alpha]$-element, viz.

$$zs = b : x$$

for some $b \in \alpha$ and $x \in [\alpha]$. Because a common pattern may come in handy in calculations we choose to denote $\lfloor\alpha\rfloor$-elements by their conslist representation. Therefore the base case and the compound case of $\lfloor\alpha\rfloor$ will have shape $[a]$ and $a : b : x$ respectively. (Note that $(:)$ is used in a right-associative way.)

In the sequel we will use a conslist representation for $zss \in \lfloor\lfloor\alpha\rfloor\rfloor$, viz. $zss = z : zs$ for some $z \in \lfloor\alpha\rfloor$ and $zs \in [\lfloor\alpha\rfloor]$ which, with a conslist representation for z, looks like

$$zss = (c : y) : zs$$

for some $c \in \alpha$, $y \in [\alpha]$ and $zs \in [\lfloor\alpha\rfloor]$.
□

Another domain that plays a role in the run length encoding problem is the product of α and the positive naturals. We omit its inductive definition and appeal to the usual notation. Therefore the base case and the compound case for $\alpha \times \mathbb{N}^+$ will have shape $(a, 1)$ and $(a, n + 1)$ respectively.

7.1 Formalization of the problem

A specification of run length encoding can now be given by

$$\begin{aligned} rle \in\ & \lfloor \alpha \rfloor \to \lfloor \alpha \times \mathbb{N}^+ \rfloor \\ rle =\ & mkrepr \circ cparts \end{aligned}$$

where $cparts \in \lfloor \alpha \rfloor \to \lfloor \lfloor \alpha \rfloor \rfloor$ partitions a string into a series of constant non-empty segments each of maximal length, and $mkrepr \in \lfloor \lfloor \alpha \rfloor \rfloor \to \lfloor \alpha \times \mathbb{N}^+ \rfloor$ transforms the elements of this series (viz. constant segments) into pairs (data item, length).
□

For definitions of *cparts* and *mkrepr* we invoke the induction principle in its constructive mode. For $cparts \in \lfloor \alpha \rfloor \to \lfloor \lfloor \alpha \rfloor \rfloor$:

$$\begin{array}{lcl} cparts.[a] & = & [[a]] \\ cparts.(a : b : x) & = & \text{let } ((c : y) : zs) = cparts.(b : x) \text{ in} \\ & & \text{if } a = c \to (a : c : y) : zs \\ & & [\!] \ a \neq c \to [a] : (c : y) : zs \\ & & \text{fi} \end{array}$$

For $mkrepr \in \lfloor \lfloor \alpha \rfloor \rfloor \to \lfloor \alpha \times \mathbb{N}^+ \rfloor$ we need a simultaneous computation of some element (e.g. the head) of a list and the length of that list.

$$\begin{array}{lcl} (hd \vartriangle length).[a] & = & (a, 1) \\ (hd \vartriangle length).(a : b : x) & = & \text{let } (c, n) = (hd \vartriangle length).(b : x) \text{ in} \\ & & (a, n + 1) \end{array}$$

Then *mkrepr* can be defined by

$$\begin{array}{lcl} mkrepr.[as] & = & [(hd \vartriangle length).as] \\ mkrepr.(as : bs : xs) & = & (hd \vartriangle length).as \ : \ mkrepr.(bs : xs) \end{array}$$

Functional programmers may recognize *mkrepr* as $map.(hd \vartriangle length)$.

7.2 Construction of *folds* via characterization

Having definitions for the components of our problem, we will explore their algebraic properties. Is *cparts* a $fold$? Indeed it is. The $fold$-characterization shows that

$$\begin{array}{ll} & cparts = fold.(e \triangledown \oplus) \\ \equiv & \quad \{ \text{ charn } \} \\ & cparts.[a] = e.a \\ & cparts.(a : b : x) = a \ \oplus \ cparts.(b : x) \end{array}$$

$$
\begin{array}{ll}
\equiv & \{ \text{ def } cparts \} \\
& [[a]] = e.a \\
& \text{let } ((c:y):zs) = cparts.(b:x) \text{ in} \\
& \left.\begin{array}{l} \text{if } a = c \ \rightarrow \ (a:c:y):zs \\ [\!] \ a \neq c \rightarrow [a]:(c:y):zs \\ \text{fi} \end{array}\right\} = a \oplus ((c:y):zs) \\
\Leftarrow & \{ \text{ generalize, instantiate } ((c:y):zs) := cparts.(b:x) \} \\
& [[a]] = e.a \\
& \left.\begin{array}{l} \text{if } a = c \ \rightarrow \ (a:c:y):zs \\ [\!] \ a \neq c \rightarrow [a]:(c:y):zs \\ \text{fi} \end{array}\right\} = a \oplus ((c:y):zs)
\end{array}
$$

I.e.

$$
\begin{array}{lll}
cparts = & fold.([[.]] \triangledown \oplus) & \\
& \text{where} & \\
& [[.]].a = [[a]] & \\
& a \oplus ((c:y):zs) & = \text{if } a = c \rightarrow (a:c:y):zs \\
& & \quad [\!] \ a \neq c \rightarrow [a]:(c:y):zs \\
& & \quad \text{fi}
\end{array}
$$

□

The reader is encouraged to prove that $(hd \vartriangle length)$ and $mkrepr$ are $fold$'s too.

7.3 Construction of *folds* via fusion

The next question is: is rle a fold? There are two ways to answer this question. The first one is using the *fold*-characterization, but this is rather laborious since an inductive definition of rle is not known yet. The other way is to use the fusion law because rle is defined as the composition of $mkrepr$ after a $fold$:

$$
\begin{array}{lll}
& rle & = \ fold.(g \triangledown \otimes) \\
\equiv & \{ \text{ previous section } \} & \\
& mkrepr \circ fold.([[.]] \triangledown \oplus) & = \ fold.(g \triangledown \otimes) \\
\Leftarrow & \{ \text{ fusion } \} & \\
& mkrepr.[[a]] & = \ g.a \\
& mkrepr.(a \oplus u) & = \ a \otimes mkrepr.u
\end{array}
$$

Knowing the definitions of $mkrepr$, $[[.]]$ and $\oplus$ we may try to calculate the parameters g and $\otimes$. A definition of g follows from the first conjunct of the antecedent:

$$g.a = mkrepr.[[a]] = [(a, 1)]$$

For the definition of $\otimes$ we explore the second conjunct of the antecedent:

$$mkrepr.(a \oplus u) = a \otimes mkrepr.u$$

Since $u \in \lfloor\lfloor\alpha\rfloor\rfloor$ we assume that $u = (b : x) : ys$ for some $b \in \alpha$, $x \in [\alpha]$ and $ys \in [\lfloor\alpha\rfloor]$. Now we use the definition of $\oplus$ and calculate

(**case** $a = b$)

$$\begin{array}{lll}
& mkrepr.((a : b : x) : ys) & = \; a \otimes mkrepr.((b : x) : ys) \\
\equiv & \quad \{ \text{ def } mkrepr \} & \\
& (hd \vartriangle length).(a : b : x) \; : \; mkrepr.ys & = \; a \otimes ((hd \vartriangle length).(b : x) \; : \; mkrepr.ys) \\
\equiv & \quad \{ \text{ defs } hd, length \} & \\
& (a, 1 + length.(b : x)) \; : \; mkrepr.ys & = \; a \otimes ((b, length.(b : x)) \; : \; mkrepr.ys) \\
\Leftarrow & \quad \{ \text{ generalize, instantiate } n := length.(b : x),\; vs := mkrepr.ys \} & \\
& (b, 1 + n) \; : \; vs & = \; a \otimes ((b, n) : vs)
\end{array}$$

(**case** $a \neq b$)

$$\begin{array}{lll}
& mkrepr.([a] : (b : x) : ys) & = \; a \otimes mkrepr.((b : x) \; : \; ys) \\
\equiv & \quad \{ \text{ def } mkrepr \} & \\
& (hd \vartriangle length).[a] \; : \; mkrepr.((b : x) \; : \; ys) & = \; a \otimes mkrepr.((b : x) \; : \; ys) \\
\equiv & \quad \{ \text{ defs } hd, length \} & \\
& (a, 1) \; : \; mkrepr.((b : x) \; : \; ys) & = \; a \otimes mkrepr.((b : x) \; : \; ys) \\
\Leftarrow & \quad \{ \text{ generalize, instantiate } ((b : n) : vs) := mkrepr.((b : x) \; : \; ys) \} & \\
& (a, 1) \; : \; ((b, n) \; : \; vs) & = \; a \otimes ((b, n) \; : \; vs)
\end{array}$$

I.e. $rle = fold.(g \triangledown \otimes)$
where
$g.a = [(a, 1)]$
$a \otimes ((b, n) \; : \; vs) =$ if $a = b \rightarrow (b, n + 1) : vs$
$[\!]\; a \neq b \rightarrow (a, 1) : (b, n) : vs$
fi

□

7.4 Construction of decoding via fusion

Another way to exploit the fusion law is in the construction of a decoding algorithm, viz. a function $k \in \lfloor\alpha \times \mathbb{N}^+\rfloor \rightarrow \lfloor\alpha\rfloor$ such that $k \circ rle = id$ -indeed $id = fold.(\lfloor . \rfloor \triangledown (\underline{:}))$. For this case the fusion law reads

$$
\begin{array}{l}
\quad k \circ fold.(g \triangledown \otimes) = fold.(\lfloor . \rfloor \triangledown (:)) \\
\Leftarrow \\
\quad k.(g.a) = \lfloor a \rfloor \\
\quad k.(a \otimes ((b,n) : vs)) = a : k.((b,n) : vs)
\end{array}
$$

which is now to be regarded as an equation in k. The requirements put on k become clear by substitution of the definitions of g and $\otimes$

$$
\begin{array}{lcll}
k.[(a,1)] & = & \lfloor a \rfloor & \\
k.((a,n+1) : vs) & = & a : k.((a,n) : vs) & \\
k.((a,1) : (b,n) : vs) & = & a : k.((b,n) : vs) & \text{if } a \neq b
\end{array}
$$

Now we recall that $\lfloor . \rfloor = (: [])$ and $(a : x) = (a : x)$ provided $x \neq []$. So, when we define k to be the total function

$$
\begin{array}{lcll}
k.[(a,1)] & = & [a] & \\
k.[(a,n+1)] & = & a : k.[(a,n)] & \text{if } n \neq 0 \\
k.((a,1) : vs) & = & a : k.vs & \text{if } vs \neq [] \\
k.((a,n+1) : vs) & = & a : k.((a,n) : vs) & \text{if } n \neq 0 \text{ and } vs \neq []
\end{array}
$$

we obtain a decoding algorithm for a run length encoded string.

It is not difficult to come up with a different definition for k (profit from the fact that the requirements on k mention no restriction on occurrences of consecutive elements with the same first component -such as $[(a,m),(a,n)]$ for instance). Indeed, a leftinverse ψ to function φ is not necessarily unique because it only puts requirements on the domain elements of ψ that are in the range of φ. In the above case the range of rle contains only lists with different first components in consecutive elements.

8 Summary

Summarizing the algebraic properties for inductive datatypes discussed sofar

- An inductive datatype is a collection of freely constructed elements together with their constructions in the sense that all elements are finitely constructed and different construction lead to different elements. This enables structural induction.
- The structure of a datatype, i.e. the carrier and the constructing functions, is algebraically given by a functor. The inductive datatype is the initial datatype with that functor as structure, it is the least fixpoint of that functor. (We only consider structuring functors that admit a least fixpoint.)
- The initiality of the datatype allows only unique homomorphisms from the inductive datatype to other datatypes of the same structure. (The basis for the structural induction.) Those unique homomorphisms, also known as "fold"s, are therefore characterized by their homomorphy, this is the so-called fold-characterization.
- Since homomorphy is preserved under composition, the composition of a homomorphism after a fold is a fold again. This so-called fusion-law allows for a simple calculational means to construct and recognize folds.

Final Remarks

The theory mentioned above shows just a glimpse of the results that are established by the study of F-algebras: there exists a similar theory for inductive datatypes with laws, and for mutually recursive datatypes.

Since F-coalgebras are dual to F-algebras, dual theories can be given for coinductive datatypes in all their variants.

□

* * *

Additional material and links.

A paper with roughly the same content but approached from a different perspective is

- Hutton, G.(1999). A tutorial on the universality and expressiveness of fold. Journal of Functional Programming, 9(4), Cambridge University Press.

In this paper Graham Hutton approaches folds as functional language constructions, it is a syntax directed view rather than a categorical one. The universality and the algebraic laws are discussed as well as the constructive use of fusion.

Two other useful references with a more categorical approach are the monographs

- Fokkinga, M.(1992). Law and order in algorithmics. Ph.D.thesis, Technical University Twente, The Netherlands.
- Jeuring, J.T.(1993). Theories for algorithm calculation. Ph.D.thesis, University of Utrecht, The Netherlands.

A Exercises

For each of the following inductive definitions define

- the functor associated to the inductive definition.
- the notion of an F-algebra for this functor.
- the notion of an F-homomorphism for this functor.
- Fix the initial algebra and formulate the characterization, computation and fusion property.

Naturals. The set $\mathbb{N}$ of natural numbers is the smallest set X such that

$$\begin{array}{lll} & & 0 \in X \\ n \in X & \Rightarrow & n+1 \in X \end{array}$$

Leaf trees over α. Let α be an arbitrary set. The set $\mathcal{T}_\alpha$ of finite leaf trees over α is the smallest set such that

$$\begin{array}{lll} a \in \alpha & \Rightarrow \langle a \rangle \in X & \textit{leaf} \\ lt \in X \ \wedge \ rt \in X & \Rightarrow \langle lt, rt \rangle \in X & \textit{node} \end{array}$$

Rose trees over α. Let α be an arbitrary set. The set $\mathcal{R}_\alpha$ of finite rose trees over α is the smallest set X such that

$$a \in \alpha \ \wedge \ rs \in \mathbb{L}_X \ \Rightarrow \langle a, rs \rangle \in X$$

□