Aggregation — polymorphic and polytypic

Maarten M. Fokkinga

Version of August 31, 2011, 9:41

Abstract. Repeating the work of Meertens [1] we show how to define in a few lines a *very* general "aggregation" function. It is parameterized by a binary operation $\oplus : t \times t \to t$ that performs the elementary aggregation steps on values of type t. The aggregation function, then, aggregates all t constituents of a composite $...t...$ value into a single t value; its type is: $...t... \to t$. The generality is three-fold. First, type t is arbitrary and may be different for different invocations; so the aggregation is *polymorphic*: working on many types. Second, expression $...t...$ is an arbitrary *regular type* (built from $+$, $\times$, *List* and so on) and may be different for different invocations; so the aggregation is *polytypic*, also known as *generic*: working on many type structures. Third, operation $\oplus$ may be different for different invocations; by suitable choices and a little preprocessing and postprocessing, the function aggregates the t constituents of a given structure into their summation, or their maximum, or their count, or their average, or their first one, or their longest path, and so on.

Footnote
Nothing of this material is new; a *far* more precise and *far* more general elaboration with *far* more discussion was given by Meertens [1] in 1996. Our intention is to give a self-contained exposition that is more accessible for the uninitiated reader who wants to see the idea with a minimum of formal details.

The main difference with Meerstens' exposition is our way of dealing with functors: in our exposition the only property of functors that we use is that they map sets to sets. A minor difference is the added elaboration for lists, natural numbers, rose trees, and Maybe values. Thus, this note may be viewed as a 8-page introduction in 2011 to Meertens' 16-page paper of 1996.

Notation. In the context of a structured type, we use xs, ys, zs to vary over structured values, and x, y, z to vary over constituents of a structured value. For example, we might say "a nonempty list xs is built from a value y and a sublist zs". Furthermore, we let f, g, h vary over functions, and t, a, b, c vary over types. (Type variable t is only more special than a, b, c in that it relates to the parameter $\oplus : t \times t \to t$.) Finally, *type* varies over type *expressions*.

Introduction. Aggregation is a frequently occurring concept in computing, and is programmed by many people over and over again for many kinds of data structures and many kinds of aggregation. Fortunately, SQL has a built-in facility for aggregation, thus freeing the user from reinventing the wheel, but unfortunately it does so for only a few kinds, namely counting, summation, maximization, minimization, and averaging, and only for sole data structure of SQL, namely tables. Meertens [1] gives one single definition of an aggregation that can be instantiated to many kinds (surpassing SQL) and that works, for arbitrary type t and operation $\oplus : t \times t \to t$, on all of the following types and many more:

$$
\begin{array}{l}
t \times t \\
\mathit{List}\, t \\
\mathit{List}\, (t \times t) \\
\mathit{List}\, (\mathit{List}\, t \times (\mathit{Nat} + \mathit{List}\, t)) \\
\mathit{Nat} + \mathit{List}\, (\mathit{Rose}\, (\mathit{Rose}(\mathit{List}\, (\mathit{Nat} \times t))\, (\mathit{List}\, (t + t \times t)))\, (\mathit{List}\, t))
\end{array}
$$

Here, *Nat*, *List*, and *Rose* are (built-in or user-defined) so-called *type functors* of arity 0, 1 and 2, respectively. [Footnote: the "official" functor *Rose* is 1-ary; we have made it into a 2-ary functor just for the purpose of illustration.] There is no restriction on the structure (nesting) and the user-defined type functors as long as only $+$ (disjoint union) and $\times$ (Cartesian product) and user-defined type functors are used and, in the type functor definitions, only previously or simultaneously defined type functors. Such types are called regular.

There is no conceptual difficulty at all in "doing aggregation" over such types: for an arbitrary value of such a type, go top-down through the value and at each "node", if it is of type t then just take the t value, and otherwise just delegate aggregation to the components of the node, followed by taking the aggregation results together with operation $\oplus$. The *only* problem is to express this formally for types of the form $T\ \mathit{type}_1 \ldots \mathit{type}_n$ where T is a type functor (for types like $\mathit{type}_1 \times \mathit{type}_2$ and $\mathit{type}_1 + \mathit{type}_2$, the solution is rather clear). To solve the problem, we need for arbitrary T some way of expressing inductively defined functions over $T\ \mathit{type}_1 \ldots \mathit{type}_n$; we shall define the notation $(\!|\ldots|\!)$ to serve this purpose.

Since our type functors T are built from regular types, we define regular types first, and only later define type functors (including a way to express induction in a uniform way).

Regular types. The grammar for *regular type* expressions reads as follows:

$$
\begin{array}{lcll}
\mathit{type} & ::= & t \mid a \mid b \mid c \mid \ldots & \text{type variables and constants} \\
 & \mid & () & \text{a singleton set, whose only member is denoted } () \\
 & \mid & \mathit{type} + \mathit{type} & \text{2-ary disjoint union with 2 arguments} \\
 & \mid & \mathit{type} \times \mathit{type} & \text{2-ary Cartesian product with 2 arguments} \\
 & \mid & T\ \mathit{type}\ \ldots\ \mathit{type} & n\text{-ary type functor } T \text{ with } n \text{ arguments}
\end{array}
$$

We consider disjoint union and Cartesian product well-known, except perhaps for the following notation (i ranges over $1, 2$):

$$
\begin{array}{lcll}
\mathit{in}_i & : & a_i \to a_1 + a_2 & \text{injection into } i\text{th component} \\
\mathit{ex}_i & : & a_1 \times a_2 \to a_i & \text{extraction from } i\text{th component}
\end{array}
$$

We postpone the explanation of type functors T (such as *List*, *Nat*, and *Rose*).

The definition. Let t be an arbitrary type and $\oplus : t \times t \to t$ an arbitrary operation, and, for the time being, assume that $\oplus$ has a neutral element, $\nu_\oplus$, meaning that $x \oplus \nu_\oplus = x = \nu_\oplus \oplus x$. Aggregation with $\oplus$ is denoted $\langle\!\langle \oplus \rangle\!\rangle$. For readability we make the type parameter of the aggregation explicit within the definition: $\langle\!\langle \oplus \rangle\!\rangle_{\mathit{type}} : \mathit{type} \to t$. (Presumably, for each application the type can be derived from the context.) The definition proceeds by induction on the construction of *type*. The entire definition consists of six simple lines:

$$\begin{array}{lcll}
\langle\!\langle \oplus \rangle\!\rangle_t \, x & = & x & \\
\langle\!\langle \oplus \rangle\!\rangle_a \, x & = & \nu_\oplus & \text{for } a \neq t \\
\langle\!\langle \oplus \rangle\!\rangle_{()} \, x & = & \nu_\oplus & \text{(the typing implies } x = ()) \\
\langle\!\langle \oplus \rangle\!\rangle_{type_1 + type_2} \, xs & = & \multicolumn{2}{l}{\text{case } \; xs = in_1 \, x_1 : \; \langle\!\langle \oplus \rangle\!\rangle_{type_1} \, x_1, \quad xs = in_2 \, x_2 : \; \langle\!\langle \oplus \rangle\!\rangle_{type_2} \, x_2} \\
\langle\!\langle \oplus \rangle\!\rangle_{type_1 \times type_2} \, xs & = & \multicolumn{2}{l}{\langle\!\langle \oplus \rangle\!\rangle_{type_1} \, (ex_1 \, xs) \; \oplus \; \langle\!\langle \oplus \rangle\!\rangle_{type_2} \, (ex_2 \, xs)} \\
\langle\!\langle \oplus \rangle\!\rangle_{T \; type} \, xs & = & \multicolumn{2}{l}{(\!|\, \langle\!\langle \oplus \rangle\!\rangle_{F_1 \; type \; t}, \; \ldots, \; \langle\!\langle \oplus \rangle\!\rangle_{F_k \; type \; t} \,|\!)_T \; xs}
\end{array}$$

The real work is done in the clauses where $\oplus$ appears outside the $\langle\!\langle\ \rangle\!\rangle$-brackets; the remaining clauses, including the last one, just delegate the work to constituents. For notational simplicity we have taken, in the last line, type functor T to be 1-ary; we postpone the explanation of the last clause.

Thus $\langle\!\langle + \rangle\!\rangle$ sums all numbers in a value of arbitrarily structured type.

Before proceeding to define the notion of *type functor* (with examples *List*, *Nat*, and *Rose*) and subsequently explaining the last clause in the definition, we first need the notion of plain *functor*.

Functor. A (unary) *functor* F is a mapping of sets to sets, satisfying the several properties which we don't need in the sequel and therefore do not spell out:

$$a \text{ is a set} \quad \Rightarrow \quad F\,a \text{ is a set}$$

Similarly for n-ary functors. The $+$ and $\times$ discussed above are "built-in" 2-ary functors.

Type functor. A type functor is a special functor. In this paper it suffices to say that a type functor is defined by means of a datatype declaration. We shall eleborate four well-known examples which together cover the main cases: lists (1-ary type functor), natural numbers (0-ary type functor), and rose trees (2-ary type functor, defined in terms of one other type functor), and finally the so-called Maybe concept (it is defined 'directly' whereas the previous three are defined 'inductively'). These examples are intended to be sufficient for the reader to get a feeling for the really general case.

Footnote. The definitions claim the existence of certain objects with certain properties, thus we have to prove that this is true. However, for the general case these proofs are beyond the scope of this paper, and for the simple examples the proofs are left to the reader.

Lists. The 1-ary type functor *List* for 'possibly-empty finite lists' is defined as follows:

$$\textbf{data } \mathit{List}\,a \quad \textbf{has} \quad \mathit{nil} : () \to \mathit{List}\,a, \qquad \mathit{cons} : a \times \mathit{List}\,a \to \mathit{List}\,a$$

The semantics of such a definition consists of these two parts:

- By definition, *List a* is a set, for arbitrary set a, and there exist so-called *constructor*s *nil* and *cons*, typed as above. (The name 'constructor' derives from the fact that these are the sole means to construct elements of the set *List a*.)

- By definition, the so-called *induction principle* holds. It says that for arbitrary $f_1 : () \to b$ and $f_2 : a \times b \to b$ there exists precisely one function $h : \mathit{List}\ a \to b$ satisfying the equations:

$$\begin{array}{lcl} h\,(\mathit{nil}\,()) & = & f_1\,() \\ h\,(\mathit{cons}\,(x, \mathit{xs})) & = & f_2\,(x,\ h\,\mathit{xs}) \end{array}$$

  Thus h systematically (=inductively) replaces the constructors *nil*, *cons* in its argument by f_1, f_2. These equations are conventionally interpreted as "defining h by induction on the construction of its argument". Since h is fully determined by f_1, f_2 and the form of the *List* definition, we write $h = (\!|f_1, f_2|\!)_{\mathit{List}}$, and call it the *List catamorphism* determined by f_1, f_2. Schematically, this function transforms

$$\begin{array}{l} \mathit{cons}(x_1, \mathit{cons}(x_2, \mathit{cons}(x_3, \ldots \mathit{nil}()))) \quad \text{to} \\ \ f_2\ \ (x_1,\ \ f_2\ \ (x_2,\ \ f_2\ \ (x_3, \ldots\ f_1\ ())))) . \end{array}$$

Naturals. The 0-ary type functor *Nat* for natural numbers is defined as follows:

$$\textbf{data}\ \mathit{Nat}\quad \textbf{has}\quad \mathit{zero} : () \to \mathit{Nat}, \quad \mathit{succ} : \mathit{Nat} \to \mathit{Nat}$$

The semantics consists of these two parts:

- *Nat* is a set, and there exist so-called constructors *zero* and *succ*, typed as above.
- The induction principle holds; it now says that for arbitrary $f_1 : () \to a$ and $f_2 : a \to a$ there exists precisely one function $h : \mathit{Nat} \to a$ satisfying:

$$\begin{array}{lcl} h\,(\mathit{zero}\,()) & = & f_1\,() \\ h\,(\mathit{succ}\,\mathit{xs}) & = & f_2\,(h\,\mathit{xs}) \end{array}$$

  Thus h replaces systematically (=inductively) the constructors *zero*, *succ* in its argument by f_1, f_2. For such h we write $h = (\!|f_1, f_2|\!)_{\mathit{Nat}}$. Schematically, this function transforms

$$\begin{array}{l} \mathit{succ}(\mathit{succ}(\mathit{succ}(\ldots \mathit{zero}()))) \quad \text{to} \\ \ f_2\ \ (\ \ f_2\ \ (\ \ f_2\ \ (\ldots\ \ f_1\ \ ()))) . \end{array}$$

Rose trees. Here we use a previously defined type functor, *List*, in the definition of the 2-ary type functor *Rose* for rose trees. [Footnote: the "official" functor *Rose* is 1-ary; we have made it into a 2-ary functor just for the purpose of illustration.] A rose tree over a and b is a tree where each node has two components, of type a and b, respectively, and a list of children (which are rose trees again):

$$\textbf{data}\ \mathit{Rose}\ a\ b\quad \textbf{has}\quad \mathit{fork} : a \times b \times \mathit{List}\,(\mathit{Rose}\ a\ b) \to \mathit{Rose}\ a\ b$$

The semantics consists of these two parts:

- *Rose* a b is a set, for arbitrary sets a and b, and there exists a so-called constructor *fork*, typed as above.

- The induction principle holds; it now says that for arbitrary $f_1 : a \times b \times List\ c \to c$ there exists precisely one function $h : Rose\ a\ b \to c$ satisfying:

$$h\,(fork\,(x, y, zs)) \quad = \quad f_1\,(x,\ y,\ (List\,h)\,zs)$$

  Thus h replaces systematically the constructor *fork* in its argument by f_1. For such h we write $h = (\!|f_1|\!)_{Rose}$. Schematically, this function transforms

$$\begin{array}{l} fork(x_1, y_1, cons(fork(x_2, y_2, ...), cons(fork(x_3, y_3, ...), cons(\ldots nil()))))\quad \text{to} \\ \ f_1\ \ (x_1, y_1, cons(\ \ f_1\ \ (x_2, y_2, ...), cons(\ \ f_1\ \ (x_3, y_3, ...), cons(\ldots nil())))). \end{array}$$

Maybe. The 1-ary type functor *Maybe* for “possibly absent value” is defined as follows:

$$\textbf{data}\ Maybe\ a \quad \textbf{has} \quad none : () \to Maybe\ a, \quad one : a \to Maybe\ a$$

The semantics consists of these two parts:

- $Maybe\ a$ is a set, for arbitrary set a, and there exist so-called constructors *none* and *one*, typed as above.
- The induction principle holds; it now says that for arbitrary $f_1 : () \to a$ and $f_2 : a \to a$ there exists precisely one function $h : Maybe\ a \to a$ satisfying:

$$\begin{array}{lcl} h\,(none\,()) & = & f_1\,() \\ h\,(one\,x) & = & f_2\,x \end{array}$$

  In more conventional words, this function is a case distinction; it maps arbitrary x to ‘case $x = none()$: $f_1()$, $x = one\,x_1$: $f_2\,x_1$’. Thus h systematically replaces the constructors *none*, *one* in its argument by f_1, f_2. For such h we write $h = (\!|f_1, f_2|\!)_{Maybe}$.

In general. Now the general case for 1-ary type functors, leaving the generalization to arbitrary n to the imagination of the reader. A 1-ary type functor T is defined by the following declaration:

$$\textbf{data}\ T\ a \quad \textbf{has} \quad in_1 : type_1 \to T\ a, \quad \ldots, \quad in_k : type_k \to T\ a$$

Here, each $type_j$ is a regular type expression which has no occurrences of T except in the form ‘$T\ a$’. Previously (and simultaneously) defined type functors may occur in the $type_j$, see the example of rose trees. The expression can be written as $type_j = F_j\ a\,(T\ a)$, where F_j is the $n{+}1$-ary functor defined by $F_j\ a\ b =$ “$type_j$ in which each occurrence of $T\ a$ is replaced by b” (for a fresh variable b). The semantics consists of two three parts:

- $T\ a$ is a set, for arbitrary set a, and there exist so-called constructors in_j, typed as above.
- The induction principle holds, saying that for arbitrary $f_j : F_j\ a\ b \to b$ $(j = 1\,..\,k)$ there exists precisely one function $h : T\ a \to b$ satisfying:

$$h\,(in_j\,x) \quad = \quad f_j\,((F_j\ id_a\,h)\,x), \qquad \text{for } j = 1\,..\,k$$

  Thus h replaces systematically each constructor in_j in its argument by f_j. For such h we write $h = (\!|f_1, \ldots, f_k|\!)_T$.

Remaining explanation. We are now able to explain the crucial clause of the aggregation definition:

$$\langle\!\langle\oplus\rangle\!\rangle_{T\,type}\,xs \;=\; (\!|\langle\!\langle\oplus\rangle\!\rangle_{F_1\,type\,t},\;\ldots,\;\langle\!\langle\oplus\rangle\!\rangle_{F_k\,type\,t}|\!)_T\;xs$$

We claim that here aggregation is delegated, by induction on the construction of argument xs, to the constituents of xs. We make this clear by elaborating the clause for the example type functors that we have given earlier.

Remember, for lists we have $k = 2$ and $F_1\;a\;b = ()$ and $F_2\;a\;b = a \times b$, so that:

$$\begin{array}{lcl}
\langle\!\langle\oplus\rangle\!\rangle_{List\,type}\,xs & = & (\!|\langle\!\langle\oplus\rangle\!\rangle_{F_1\,type\,t},\;\langle\!\langle\oplus\rangle\!\rangle_{F_2\,type\,t}|\!)_{List}\;xs \\
 & = & (\!|\langle\!\langle\oplus\rangle\!\rangle_{()},\;\langle\!\langle\oplus\rangle\!\rangle_{type\times t}|\!)_{List}\;xs \\
 & = & A\,xs, \text{ where } A : List\;type \to t \text{ is inductively defined by:} \\
 & & \qquad \begin{array}{lcl} A\,(nil\,()) & = & \langle\!\langle\oplus\rangle\!\rangle_{()}\,() \\ A\,(cons\,(x_1, x_2)) & = & \langle\!\langle\oplus\rangle\!\rangle_{type\times t}\,(x_1, A\,x_2) \end{array} \\
 & = & A\,xs, \text{ where } A : List\;type \to t \text{ is inductively defined by:} \\
 & & \qquad \begin{array}{lcl} A\,(nil\,()) & = & \nu_\oplus \\ A\,(cons\,(x_1, x_2)) & = & \langle\!\langle\oplus\rangle\!\rangle_{type}\,x_1 \;\oplus\; A\,x_2 \end{array}
\end{array}$$

So, when invoked at a list, the aggregation performs its calculation by induction on the *nil*, *cons* structure of its argument, meaning in this case that it aggregates the t values, if any, in all nodes of the list.

Remember, for naturals we have $k = 2$ and $F_1\;b = ()$ and $F_2\;b = b$, so that:

$$\begin{array}{lcl}
\langle\!\langle\oplus\rangle\!\rangle_{Nat}\,xs & = & (\!|\langle\!\langle\oplus\rangle\!\rangle_{F_1\,t},\;\langle\!\langle\oplus\rangle\!\rangle_{F_2\,t}|\!)_{Nat}\;xs \\
 & = & (\!|\langle\!\langle\oplus\rangle\!\rangle_{()},\langle\!\langle\oplus\rangle\!\rangle_{t}|\!)_{Nat}\;xs \\
 & = & A\,xs, \text{ where } A : Nat \to t \text{ is inductively defined by:} \\
 & & \qquad \begin{array}{lcl} A\,(zero\,()) & = & \langle\!\langle\oplus\rangle\!\rangle_{()}\,() \\ A\,(succ\,xs) & = & \langle\!\langle\oplus\rangle\!\rangle_{t}\,(A\,xs) \end{array} \\
 & = & \nu_\oplus
\end{array}$$

Again, when invoked at a natural, the aggregation performs it calculation by induction on the *zero*, *succ* structure of the natural, meaning in this case that it finally yields just $\nu_\oplus$, because there are no t constituents in a natural.

Remember, for Maybe values we have $k = 2$ and $F_1\;a\;b = ()$ and $F_2\;a\;b = a$, so that:

$$\begin{array}{lcl}
\langle\!\langle\oplus\rangle\!\rangle_{Maybe\,type}\,xs & = & (\!|\langle\!\langle\oplus\rangle\!\rangle_{F_1\,type\,t},\;\langle\!\langle\oplus\rangle\!\rangle_{F_2\,type\,t}|\!)_{Maybe}\;xs \\
 & = & (\!|\langle\!\langle\oplus\rangle\!\rangle_{()},\;\langle\!\langle\oplus\rangle\!\rangle_{type}|\!)_{Maybe}\;xs \\
 & = & A\,xs, \text{ where } A \text{ is defined by:} \\
 & & \qquad \begin{array}{lcl} A\,(none\,()) & = & \langle\!\langle\oplus\rangle\!\rangle_{()}\,() \\ A\,(one\,x) & = & \langle\!\langle\oplus\rangle\!\rangle_{type}\,x \end{array} \\
 & = & A\,xs, \text{ where } A \text{ is defined by:} \\
 & & \qquad \begin{array}{lcl} A\,(none\,()) & = & \nu_\oplus \\ A\,(one\,x) & = & \langle\!\langle\oplus\rangle\!\rangle_{type}\,x \end{array}
\end{array}$$

So, when invoked at one *type* value or none, the aggregation performs its calculation by "trivial induction" on the *none*, *one* structure of its argument, meaning in this case that it aggregates the t values in the one value.

Remember, for rose trees we have $k = 1$ and $F_1\ a\ b\ c = a \times b \times \mathit{List}\ c$, so that:

$$\begin{array}{lcl}
\langle\!\langle\oplus\rangle\!\rangle_{\mathit{Rose}\ \mathit{type}_1\ \mathit{type}_2}\ \mathit{xs} & = & (\![\langle\!\langle\oplus\rangle\!\rangle_{F_1\ \mathit{type}_1\ \mathit{type}_2\ t}]\!)_{\mathit{Rose}}\ \mathit{xs} \\
 & = & (\![\langle\!\langle\oplus\rangle\!\rangle_{\mathit{type}_1 \times \mathit{type}_2 \times \mathit{List}\ t}]\!)_{\mathit{Rose}}\ \mathit{xs} \\
 & = & A\ \mathit{xs}, \text{ where } A : \mathit{Rose}\ \mathit{type}_1\ \mathit{type}_2 \to t \text{ is inductively defined by:} \\
 & & A\,(\mathit{fork}(x, y, \mathit{zs}))\ \ =\ \ \langle\!\langle\oplus\rangle\!\rangle_{\mathit{type}_1 \times \mathit{type}_2 \times \mathit{List}\ t}\,(x,\ y,\ (\mathit{List}\ A)\ \mathit{zs}) \\
 & = & A\ \mathit{xs}, \text{ where } A : \mathit{Rose}\ \mathit{type}_1\ \mathit{type}_2 \to t \text{ is inductively defined by:} \\
 & & A\,(\mathit{fork}(x, y, \mathit{zs}))\ \ =\ \ \langle\!\langle\oplus\rangle\!\rangle_{\mathit{type}_1}\ x\ \oplus \\
 & & \qquad\qquad\qquad\qquad\ \ \langle\!\langle\oplus\rangle\!\rangle_{\mathit{type}_2}\ y\ \oplus \\
 & & \qquad\qquad\qquad\qquad\ \ \langle\!\langle\oplus\rangle\!\rangle_{\mathit{List}\ t}((\mathit{List}\ A)\ \mathit{zs})
\end{array}$$

So, when invoked at a rose tree, the aggregation performs it calculation by induction on the *fork* structure of its argument, meaning in this case that it aggregates the t values in all nodes of the rose tree.

Refinements

At this point, I (hope to) have achieved my goal: an introduction for laymen for parameterized, polymorphic, and polytypic aggregation. It is just for the sake of completeness that I very briefly mention two refinements given by Meertens: these are too beautiful to skip.

Including some preprocessing. Aggregation often comes with some preprocessing: subjecting all t values in a structured value to some function, before aggregating them. We generalize aggregation, for arbitrary t, t' and $f : t \to t'$ and $\oplus : t' \times t' \to t'$ to:

$$\langle\!\langle\oplus, f\rangle\!\rangle_{\mathit{type}} \quad : \quad \mathit{type} \to t'$$

The definition has exactly the same clauses as we have given for normal aggregation, except for these modifications: replace all occurrences of $\langle\!\langle\oplus\rangle\!\rangle_{\mathit{type}}$ by $\langle\!\langle\oplus, f\rangle\!\rangle_{\mathit{type}}$, and replace the right-hand side 'x' in the very first clause by '$f\ x$'. It follows that $\langle\!\langle\oplus, \mathit{id}\rangle\!\rangle_{\mathit{type}} = \langle\!\langle\oplus\rangle\!\rangle_{\mathit{type}}$, so this generalized aggregation is really a generalization of the basic aggregation. Now:

- Membership of element e of type t in an arbitrary structure is expressed by $\langle\!\langle\vee, (e{=})\rangle\!\rangle$, where $\vee$ is logical 'or', and function '$(e{=})$' checks whether its argument equals e.
- The number of t constituents ("the size") of an arbitrary structure is expressed by $\langle\!\langle +, K_1\rangle\!\rangle$, where K_1 is the function that maps each argument to just 1.
- The average of numerical constituents of an arbitrary structure is expressed by $g \circ \langle\!\langle\oplus, f\rangle\!\rangle$, where f is the function that maps a numerical value x to $(x, 1)$, operation $\oplus$ maps two pairs (x, n) and (x', n') to $(x{+}x', n{+}n')$, and g is the function that maps a pair (x, n) to x/n.

Absence of values Sometimes a structured value may have no t constituents at all, such as the empty list. In such cases we have defined the neutral element $\nu_\oplus$ to be the outcome of the aggregation. However, several operations do not have a neutral element, for example: the operation that returns its left argument, and the operation on natural numbers that returns the minimum of its arguments. In these cases the Maybe type comes to rescue, and can be built-in into the aggregation as follows. Define:

$$\langle\!\langle \oplus, f \rangle\!\rangle^{Maybe} \quad = \quad \langle\!\langle \oplus^{Maybe},\ one \circ f \rangle\!\rangle$$

where $\oplus^{Maybe}$ is defined as follows:

$$\begin{array}{lcl} one\, x \oplus^{Maybe} one\, y & = & one\,(x \oplus y) \\ one\, x \oplus^{Maybe} none & = & one\, x \\ none \oplus^{Maybe} one\, y & = & one\, y \\ none \oplus^{Maybe} none & = & none \end{array}$$

It follows that $\oplus^{Maybe}$ has a neutral element, namely *none*. Now:

- The first (left-most) element of type t in a structured value, if any, is expressed by $\langle\!\langle \otimes, id \rangle\!\rangle^{Maybe}$ where $x \otimes y = x$. The outcome is *none* in case the structured value has no t constituent. (This example shows that non-associative operations, like the current $\otimes$, can be really useful in an aggregration!)
- The depth of the deepest element of type t, if any, is expressed by $\langle\!\langle \oplus, f \rangle\!\rangle^{Maybe}$ where $x \oplus y = \max(x, y) + 1$ and $f\, x = 0$. The outcome is *none* in case the structured value has no t constituent.

Final remark

So far this exposition only contains definitions and examples. Meertens also shows that the definitions can be *used*. For example, he presents the following *theorems*:

- If $h(x \oplus y) = h\, x \otimes h\, y$ and $h\, \nu_\oplus = \nu_\otimes$, then $h \cdot \langle\!\langle \oplus, f \rangle\!\rangle = \langle\!\langle \otimes,\ h \cdot f \rangle\!\rangle$. The latter equation may help to express aggregations in a way that can be more efficiently computed.
- If $\oplus$ is associative, then $\langle\!\langle \oplus \rangle\!\rangle = \langle\!\langle \oplus \rangle\!\rangle_{List} \cdot \mathit{flatten}$, where $\mathit{flatten} = \langle\!\langle +\!\!+, [\_] \rangle\!\rangle$. So, for associative $\oplus$, the structure does't matter; only the order of the elements counts.

References

[1] Lambert Meertens. Calculate polytypically! In Herbert Kuchen and S. Doaitse Swierstra, editors, *Programming Languages: Implementations Logics, and Programs, Proc. Eighth International Symposium PLILP '96*, LNCS 1140, pages 1–16. Springer-Verlag, 1996.