Sequential Data Algebra Primitives

Jan Bergstra[*,†] Alex Sellink[†]

[*] *Utrecht University, Department of Philosophy*
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands

[†] *University of Amsterdam, Programming Research Group*
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

janb@fwi.uva.nl *alex@fwi.uva.nl*

1 Introduction

The purpose of this paper is to develop a family of data type specifications and a particular method for writing such specifications based on the four valued logic of [BBR95]. The method is an informal one cast in a number of design rules for specifications of data algebras. The phrase data algebra is used rather than abstract data type in order to emphasise that the data types are first order many sorted algebras that comply a number of requirements (design rules).

The four valued (McCarthyan) logic, the new notations for list operators, the omnipresent divergent and meaningless objects, and the very hierarchy of modules below can be viewed as primitives for data algebra. Because all operations including the logical connectives are sequential in the sense that they inspect their arguments in some definite order this style of data algebra is called sequential data algebra.

Evidently what we propose is only a choice from an infinite number of different options. So what is being developed below is in fact a set of primitives for a style of sequential data algebra.

Some style of data algebra is needed in order to avoid endless duplication and reinvention in the practice of algebraic specification. The mere presence of module libraries will not suffice to reduce the frequency of these improductive events. More importantly far more standardisation will be needed to obtain a body of algebraic specifications with high credibility. Again libraries as such will not generate confidence. In order to achieve standardisation explicit specification styles are needed. The style of sequential data algebra that we use is characterized by a number of rather restrictive design rules, that simultaneously constitute its rationale. The structuring mechanisms as well as the approach to syntax definition and prettyprinting are based on ASF+SDF. It follows that our style of sequential data algebra is geared towards and biased by ASF+SDF. This, however, has had no significant impact on the semantics of the descriptions or on the choice of operators and equations.

The specifications below can be viewed as equational initial algebra specifications. The specifications have been produced, typechecked and simulated using the ASF+SDF system. We mention [HK89] for SDF and [Kli95] for the system. We have kept the syntactic sugar provided by ASF+SDF as this may help the reader to understand the specifications.

2 Design rules for module hierarchies

Four valued logic suggests strong heuristics on how to set up a module hierarchy. These heuristics are formulated in terms of design rules below. Though quite restrictive, still, these rules allow an expressive specification style.

- *Divergent and meaningless objects.* Every sort will contain two objects D and M representing a divergent computation and a meaningless value. M will be used to represent values like the first element of an empty list, the result of division by zero in the rationals and various type errors. The divergent value D will be used to give a default meaning to terms for which no meaning can be found using the rules of a specification in a finitary way. This default meaning is a conceptual stage that can only be defined on basis of a fully developed specification. Technically it takes the form of an infinitary rule: DDR (Default Divergence Rule). In contrast the meaningless value is like all other values: equations that equate an object with the meaningless value must always be provable from the specification using finitary means.

  There are many possible ways in which computing systems may fail to converge and lead to terms that can meaningfully be assigned the value D (see Appendix A). The main purpose of having D available is different however. We imagine that a module can be implemented by means of a family of imperative programs, that compute appropriate normal forms. Then, divergence must be accepted as a possible outcome. The equations for D explain how divergence is to be propagated through expressions.

  We will work in a context of term rewriting and D and M will always be normal forms in the appropriate rewrite systems. Elements of a sort different from D and M will be called *proper elements* (or equivalently: *well-defined elements*)

- *Four valued logic.* It follows from the above that the Boolean sort of a data algebra will be extended with divergent and meaningless objects as well. In fact all modules import the four valued propositional algebra with values D, T (true), F (false) and M. Each of these must be a normal form in the rewrite systems that conform to these design rules.

  We propose a fixed convention for identifying truth-values with non-negative integers. 0 represents F, 1 represents T, 2 represents D and 3 represents M.

- *Ground semi-completeness.* Every specification after expansion of its imports constitutes a term rewriting system which is ground confluent and weakly ground terminating (prefixing with 'ground' restricts a requirement to closed terms only).

  This combination was baptized (ground) semi-completeness in [BK87]. A ground semi-complete term rewriting system has unique normal forms (for ground terms) which can be found effectively. This guarantees that the initial algebra of a ground semi-complete TRS is a computable algebra (see [BT95] and [BT83]) Notice that according to [BT95] complete term rewriting systems allow the specification of all computable data algebras (provided auxiliary functions may be used). Consequently a specification method based on ground semi-complete term rewriting specifies exactly the same computable algebras. But it turns out that the more liberal approach of semi-completeness leads to more concise specifications, mainly because long lists of equations that have to cover a number of matchings with constructor functions can often be avoided. Another even more important advantage is that by using weakly terminating specifications it is far easyer to obtain representation independence (see next item).

- *Representation independence.* If module A imports module B then (except for the boolean values D, T, F and M and for the values D and M for all other sorts) the functions and constants that are introduced in B (or in any of the modules imported by B) must not occur in lefthand sides of equations of A. We call this requirement representation independence. It allows to implement a hierarchy of modules faithfully. An implementation of A need not inspect the internal structure of objects represented by the implementation of B if the aforementioned restriction on lefthand sides of equations is met. This way the level of abstraction of data algebra descriptions is kept lower than is usual for algebraic specifications. Representation independence is a formidable advantage, however, if the specifications are meant to assist in software design.

  We mention that the requirement of representation independence leads to the use of decomposition operations such as head and tail of lists, in cases where matching on constructor functions might have been used otherwise. These decomposition operations can be fruitfully used in combination with a conditional operator and recursion. (As an example we mention the equation for eq1 in the boolean module below.) In methodological terms representation independence is a form of information hiding be it a unusual one. Indeed in object oriented languages one achieves information hiding by making a distinction between private and public operations. That distinction cannot be used to understand representation independence. The information hidden by having all imports representation independent concerns the syntax of normal forms. If lefthand sides of equations may contain operators from an imported datatype this will (for semicomplete rewriting systems at least) convey information about the syntax of normal forms. That kind of information is to be hidden in order to allow a maximal degree of freedom for implementation of the imported module. Representation independence is a requirement on structured specifications that seems to achieve just this form of information hiding.

- *Equality function available.* Every module that introduces a sort S has to provide at least a function $\mathsf{eq1}(x, y)$ which returns x if $x = \mathsf{D}$ or $x = \mathsf{M}$ and if x is defined (i.e. not equal to D or to M) it returns y if y is not defined and otherwise T resp. F depending on whether $x = y$ or not.

- *Sequential computability.* We will require that every non-constant operator transfers the values M and D along one of its arguments. In case of prefix notation we will often indicate with a numerical annotation which argument this is e.g. add2 is strict in its second argument and eq1 in its first argument. In case of binary infix notation we will often use an open circle to indicate the position that is to be inspected first. As a consequence the logical connectives available are severly restricted. We will allow the use of $\neg$ (negation), and of $\mathbin{{}_{\circ}\!\wedge}$ (left first conjunction). In the case of three valued logic this sequential logical connective is due to McCarthy [McC63]. Three other operations from [BBR95] are definable from these: $\wedge_{\circ}$, ${}^{\circ}\!\vee$ and $\vee^{\circ}$. Excluded is the use of the weakly sequential connectives ${}_{\circ}\wedge_{\circ}$ and ${}^{\circ}\!\vee^{\circ}$ as well as of the ordinary logical connectives $\vee$ and $\wedge$.

The computational status of these connectives is by no means clear. In some plausible models they can be computed sequentially (or at least with very limited parallelism) while in other models their evaluation requires genuine parallelism. For the extensions of the classical connectives $\wedge$ and $\vee$ in some plausible computational models computability cannot even be guaranteed. The requirement of sequential computability guarantees (in most cases) a very clear computational meaning of the equations. All equations should have an objective independent status which is mainly due to their algorithmic meaning. Again this contrasts the objective of abstractness implicit in the algebraic specifications tradition.

- *Focus on equations.* Names of equations as well as their forms are important. Names of modules and module hierarchies are less important. In fact naming modules is made quite difficult by the amount of bookkeeping involved. A module with natural numbers and equipped with a family of arithmetical functions is just one out of many similar conceivable modules. Thus at least the signature of a model should be part of its name if these names are to make significant distinctions. We conclude that it is the operations and the equations that are to be exported from one paper to another rather than the modular structure of a particular specification.

  Of course when module libraries are built and implemented module naming becomes a crucial part of version management control. Data algebra as such is not about such libraries in practice but about aspects of their mathematical foundations. Problems related to administration and version management are preferably ignored at this stage.

- *Minimize duplication.* Using imports and generic modules the duplication of equations should be avoided as much as possible. Preferably every module hierarchy should contain an equation only once.

- *Normalisation strategy.* A further requirement is that for each module hierarchy a reasonable terminating reduction strategy can be given. An example of such a strategy is leftmost innermost rewriting. This strategy works in a minority of cases only. More useful may be to provide an independent strategy for each function symbol. To reduce a term with that function symbol at a leading position the particular recipe for that case has to be applied. As an example consider the conditional statement which we will use in the following form: $(X\ \mathit{if}\ b\ \mathit{else}\ Y)$. The recipe is: first normalise the second argument and then apply the rule. Another example is mutiplication of naturals. Consider the rule $\mathit{mult}(X,Y) = (0\ \mathit{if}\ X=0\ \mathit{else}\ \mathit{add}(\mathit{mult}(P(X),Y),Y)$. Then a recipe may be as follows: normalise X, then determine whether $X = 0$, if so then apply the rule, otherwise reduce Y and thereafter apply the main rule. Obviously such recipes can be formalised. We will not do so in the present paper because the recipe is obvious for all operators that will occur in the specifications below. In many cases the recipes are simple and can be generated automatically. This also is a reason not to add their formalisation to a specification.

2.1 Some Remarks

We distinguish between basic modules and nonbasic ones. Within basic modules one may find the introduction of new sorts and indeed arbitrary (ground) semi-complete term rewriting systems. Nonbasic modules contain only function definitions on top of the imported structure. The definitions below are mostly encoded tail recursions.

We remark that the definition of the conditional construct requires a representation dependent import of the truth values module. Indeed if a new sort S is to be introduced the definition of the conditional construct on that sort requires the four equations that distinguish according to the four constructors of the sort of truthvalues.

In order to understand the rationale of an equation it is often necessary to know the precise hierarchy of modules in which it occurs. As an example consider the equation $sq(n) = 0 \triangleleft Z(n) \triangleright (add(sq(P(n)), add(n, P(n))))$. This equation serves well as definition of square in the absence of multiplication. Only by inspecting the entire module hierarchy in which the equation occurs it can be concluded that multiplication is not available indeed. This then establishes the rationale of the equation. For this reason the modular structure of the specification examples is specified in full detail.

3 Four valued logic

3.1 Two modules for booleans

Below we introduce a module containing a four valued propositional algebra with a sort consisting of the four constants D for the divergent value, T for true, F for false and M for meaningless, a conditional operator and an equality test. The intended rewriting strategy for the conditional operator is that it evaluates its second argument first, and applies a condTi rule thereafter. Variables b, b_i range over the sort of truth values.

$$(b_1 \triangleleft \mathsf{D} \triangleright b_2) = \mathsf{D} \qquad [\mathsf{condT1}]$$
$$(b_1 \triangleleft \mathsf{T} \triangleright b_2) = b_1 \qquad [\mathsf{condT2}]$$
$$(b_1 \triangleleft \mathsf{F} \triangleright b_2) = b_2 \qquad [\mathsf{condT3}]$$
$$(b_1 \triangleleft \mathsf{M} \triangleright b_2) = \mathsf{M} \qquad [\mathsf{condT4}]$$

$$\mathsf{eq1}(b_1, b_2) = ((\mathsf{T} \triangleleft b_2 \triangleright \mathsf{F}) \triangleleft b_1 \triangleright (\mathsf{F} \triangleleft b_2 \triangleright \mathsf{T})) \qquad [\mathsf{eqT1}]$$

We extend the module above with negation, left first conjunction and its three derivatives. Furthermore it contains a constant function T which is sensitive to its argument being proper.

$$\neg\, b = (\mathsf{F} \triangleleft b \triangleright \mathsf{T}) \qquad [\mathsf{negT1}]$$
$$b_1 \mathbin{{}_{\circ}\!\wedge} b_2 = (b_2 \triangleleft b_1 \triangleright \mathsf{F}) \qquad [\mathsf{lfcT1}]$$
$$b_1 \mathbin{{}^{\circ}\!\vee} b_2 = (\mathsf{T} \triangleleft b_1 \triangleright b_2) \qquad [\mathsf{lfdT1}]$$
$$b_1 \mathbin{\wedge_{\circ}} b_2 = b_2 \mathbin{{}_{\circ}\!\wedge} b_1 \qquad [\mathsf{rfcT1}]$$
$$b_1 \mathbin{\vee^{\circ}} b_2 = b_2 \mathbin{{}^{\circ}\!\vee} b_1 \qquad [\mathsf{rfdT1}]$$

$$\mathsf{T}(b) = \mathsf{eq1}(b, b) \qquad [\mathsf{trueT1}]$$

3.2 Conditionals over an arbitrary sort

The *generic module* below introduces conditionals, syntax for equality and syntax for the true predicate. All these concepts are defined on an arbitrary sort. We use x, x_i to indicate the a variable ranges over an arbitrary sort. Note that the equations below do not form a data algebra because no equations for equality and the true predicate are present. We will assume that on every new sort the conditional construct is available.

$$
\begin{array}{ll}
(x_1 \lhd \mathsf{D} \rhd x_2) = \mathsf{D} & [\mathsf{condX1}] \\
(x_1 \lhd \mathsf{T} \rhd x_2) = x_1 & [\mathsf{condX2}] \\
(x_1 \lhd \mathsf{F} \rhd x_2) = x_2 & [\mathsf{condX3}] \\
(x_1 \lhd \mathsf{M} \rhd x_2) = \mathsf{M} & [\mathsf{condX4}]
\end{array}
$$

3.3 Lists over an arbitrary sort

The following generic module describes the construction of lists over an arbitrary sort. We use $[\,]$ for the empty list, $[x]$ for the list with single element x and ${}^{\circ}\!\frown$ for list concatenation. The predicate E decides whether a list is empty, $\urcorner$ and $\hookrightarrow$ return the head and tail (operating on the left side of the list).

In the equations below x is a variable over an arbitrary sort, say X, whereas X, X_i are variables over lists over X.

$$
\begin{array}{lll}
[\mathsf{D}] & = \mathsf{D} & [\mathsf{lscL1}] \\
[\mathsf{M}] & = \mathsf{M} & [\mathsf{lscL2}] \\
[\,] \mathbin{{}^{\circ}\!\frown} X & = X & [\mathsf{lscL3}] \\
X \mathbin{{}^{\circ}\!\frown} [\,] & = X & [\mathsf{lscL4}] \\
\mathsf{M} \mathbin{{}^{\circ}\!\frown} X & = \mathsf{M} & [\mathsf{lscL5}] \\
\mathsf{D} \mathbin{{}^{\circ}\!\frown} X & = \mathsf{D} & [\mathsf{lscL6}] \\
[x] \mathbin{{}^{\circ}\!\frown} \mathsf{D} & = (\mathsf{D} \lhd \mathsf{T}(x) \rhd \mathsf{D}) & [\mathsf{lscL7}] \\
[x] \mathbin{{}^{\circ}\!\frown} \mathsf{M} & = (\mathsf{M} \lhd \mathsf{T}(x) \rhd \mathsf{M}) & [\mathsf{lscL8}] \\
(X_1 \mathbin{{}^{\circ}\!\frown} X_2) \mathbin{{}^{\circ}\!\frown} X_3 & = X_1 \mathbin{{}^{\circ}\!\frown} (X_2 \mathbin{{}^{\circ}\!\frown} X_3) & [\mathsf{lscL9}]
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{T}(\mathsf{D}) & = \mathsf{D} & [\mathsf{trueL1}] \\
\mathsf{T}(\mathsf{M}) & = \mathsf{M} & [\mathsf{trueL2}] \\
\mathsf{T}([\,]) & = \mathsf{T} & [\mathsf{trueL3}] \\
\mathsf{T}([x]) & = \mathsf{T}(x) & [\mathsf{trueL4}] \\
\mathsf{T}(X_1 \mathbin{{}^{\circ}\!\frown} X_2) & = \mathsf{T}(X_1) \mathbin{{}_{\circ}\!\wedge} \mathsf{T}(X_2) & [\mathsf{trueL5}]
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{E}(\mathsf{D}) & = \mathsf{D} & [\mathsf{empL1}] \\
\mathsf{E}(\mathsf{M}) & = \mathsf{M} & [\mathsf{empL2}] \\
\mathsf{E}([\,]) & = \mathsf{T} & [\mathsf{empL3}] \\
\mathsf{E}([x]) & = \neg\, \mathsf{T}(x) & [\mathsf{empL4}] \\
\mathsf{E}([x] \mathbin{{}^{\circ}\!\frown} X) & = \neg\, (\mathsf{T}(x) \mathbin{{}_{\circ}\!\wedge} \mathsf{T}(X)) & [\mathsf{empL5}]
\end{array}
$$

$$
\begin{aligned}
&\Lsh \mathsf{D} &&= \mathsf{D} && \text{[lhdL1]}\\
&\Lsh \mathsf{M} &&= \mathsf{M} && \text{[lhdL3]}\\
&\Lsh [\,] &&= \mathsf{M} && \text{[lhdL3]}\\
&\Lsh [x] &&= x && \text{[lhdL4]}\\
&\Lsh ([x] \;{}^{\circ}\!\frown\; X) &&= (x \lhd \mathsf{T}(x) \mathbin{{}_{\circ}\!\wedge} \mathsf{T}(X) \rhd \mathsf{M}) && \text{[lhdL5]}
\end{aligned}
$$

$$
\begin{aligned}
&\Lsh\!\!\downarrow \mathsf{D} &&= \mathsf{D} && \text{[lrtL1]}\\
&\Lsh\!\!\downarrow \mathsf{M} &&= \mathsf{M} && \text{[lrtL2]}\\
&\Lsh\!\!\downarrow [\,] &&= \mathsf{M} && \text{[lrtL3]}\\
&\Lsh\!\!\downarrow [x] &&= ([\,] \lhd \mathsf{T}(x) \rhd \mathsf{M}) && \text{[lrtL4]}\\
&\Lsh\!\!\downarrow ([x] \;{}^{\circ}\!\frown\; X) &&= (X \lhd \mathsf{T}(x) \rhd \mathsf{M}) && \text{[lrtL5]}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{eq1}(X_1, X_2) \;=\; & (\mathsf{E}(X_1) \mathbin{{}_{\circ}\!\wedge} \mathsf{E}(X_2)) \mathbin{{}^{\circ}\!\vee} (\neg\, \mathsf{E}(X_1) \mathbin{{}_{\circ}\!\wedge} \neg\, \mathsf{E}(X_2) && \text{[eqLN1]}\\
& \qquad \mathbin{{}_{\circ}\!\wedge} \mathsf{eq1}(\Lsh X_1, \Lsh X_2) \mathbin{{}_{\circ}\!\wedge} \mathsf{eq1}(\Lsh\!\!\downarrow X_1, \Lsh\!\!\downarrow X_2))
\end{aligned}
$$

We extend the previous module with inversion, right head ($\Rsh$), right tail ($\downarrow\!\!\!\lrcorner$) and right-first-concatenation ($\frown\!{}^{\circ}$).

$$
\mathsf{inv}(X) = ([\,] \lhd \mathsf{E}(X) \rhd \mathsf{inv}(\Lsh\!\!\downarrow X) \;{}^{\circ}\!\frown\; [\Lsh X]) \qquad \text{[invL1]}
$$

$$
\begin{aligned}
&X \Rsh &&= \Lsh \mathsf{inv}(X) && \text{[rhdL1]}\\
&X \downarrow\!\!\!\lrcorner &&= \mathsf{inv}(\Lsh\!\!\downarrow \mathsf{inv}(X)) && \text{[rltL1]}\\
&X_1 \frown\!{}^{\circ} X_2 &&= \mathsf{inv}(\mathsf{inv}(X_2) \;{}^{\circ}\!\frown\; \mathsf{inv}(X_1)) && \text{[rfcL1]}
\end{aligned}
$$

4 Digits and Naturals

In this section we provide a very straightforward library of arithmetic data algebra specifications. Its novelty is in the systematic use of the aforementioned primitives and design rules. For a comprehensive account of rewriting systems for integer arithmetic we refer to [WZ95]. Other libraries of this kind can be found in [Wie91] and [vW95].

In the sequel we will sometimes make use of conditional rules. We notice, however, that these conditions are not meant to limit the applicability of the rule but just occur to enhance readability and should be viewed as abbreviations.

4.1 Digits

Natural numbers are seen as lists of digits. In the following module we introduce the sort of digits. Variables d, d_i range over digits. In further modules arbitrary radix can be used. We will use base for radix $- 1$.

$$
\mathsf{base} = 9 \qquad \text{[base1]}
$$

$\mathsf{S}(0) = 1$	[suc0]	$\mathsf{P}(0) = \mathsf{M}$	[pred0]	$\mathsf{Z}(0) = \mathsf{T}$	[zero0]
$\mathsf{S}(1) = 2$	[suc1]	$\mathsf{P}(1) = 0$	[pred1]	$\mathsf{Z}(1) = \mathsf{F}$	[zero1]
$\mathsf{S}(2) = 3$	[suc2]	$\mathsf{P}(2) = 1$	[pred2]	$\mathsf{Z}(2) = \mathsf{F}$	[zero2]
$\mathsf{S}(3) = 4$	[suc3]	$\mathsf{P}(3) = 2$	[pred3]	$\mathsf{Z}(3) = \mathsf{F}$	[zero3]
$\mathsf{S}(4) = 5$	[suc4]	$\mathsf{P}(4) = 3$	[pred4]	$\mathsf{Z}(4) = \mathsf{F}$	[zero4]
$\mathsf{S}(5) = 6$	[suc5]	$\mathsf{P}(5) = 4$	[pred5]	$\mathsf{Z}(5) = \mathsf{F}$	[zero5]
$\mathsf{S}(6) = 7$	[suc6]	$\mathsf{P}(6) = 5$	[pred6]	$\mathsf{Z}(6) = \mathsf{F}$	[zero6]
$\mathsf{S}(7) = 8$	[suc7]	$\mathsf{P}(7) = 6$	[pred7]	$\mathsf{Z}(7) = \mathsf{F}$	[zero7]
$\mathsf{S}(8) = 9$	[suc8]	$\mathsf{P}(8) = 7$	[pred8]	$\mathsf{Z}(8) = \mathsf{F}$	[zero8]
$\mathsf{S}(9) = \mathsf{M}$	[suc9]	$\mathsf{P}(9) = 8$	[pred9]	$\mathsf{Z}(9) = \mathsf{F}$	[zero9]
$\mathsf{S}(\mathsf{M}) = \mathsf{M}$	[sucM]	$\mathsf{P}(\mathsf{D}) = \mathsf{D}$	[predD]	$\mathsf{Z}(\mathsf{M}) = \mathsf{M}$	[zeroM]
$\mathsf{S}(\mathsf{D}) = \mathsf{D}$	[sucD]	$\mathsf{P}(\mathsf{M}) = \mathsf{M}$	[predM]	$\mathsf{Z}(\mathsf{D}) = \mathsf{D}$	[zeroD]

$$\mathsf{T}(d) = \mathsf{Z}(d) \overset{\circ}{\vee} \mathsf{T} \qquad \text{[trueD1]}$$

$$\mathsf{eq1}(d_1, d_2) = (\mathsf{Z}(d_2) \lhd \mathsf{Z}(d_1) \rhd (\neg \mathsf{Z}(d_2)) \underset{\circ}{\wedge} \mathsf{eq1}(\mathsf{P}(d_1), \mathsf{P}(d_2))) \qquad \text{[eqD1]}$$

The module above is extended with the following seven useful operators.

$$\mathsf{t2d}(b) = (1 \lhd b \rhd 0) \qquad \text{[T2D]}$$

The function carry1 returns 0 if the sum of the input values is below 10 and 1 if the sum of the input values is at least 10. Together with the function modsum1 (addition modulo base+1) it is the building block for the definition of addition.

$$\mathsf{carry1}(d_1,\, d_2) = (0 \lhd \mathsf{Z}(d_1) \rhd (1 \lhd \mathsf{eq1}(d_2,\, \mathsf{base}) \rhd \mathsf{carry1}(\mathsf{P}(d_1),\, \mathsf{S}(d_2)))) \qquad \text{[Carry]}$$

$$\begin{aligned} \mathsf{modsum1}(d_1,\, d_2) \;=\; & (d_2 \lhd \mathsf{Z}(d_1) \\ & \rhd (\mathsf{P}(d_1) \lhd \mathsf{eq1}(d_2,\, \mathsf{base}) \rhd \mathsf{modsum1}(\mathsf{P}(d_1),\, \mathsf{S}(d_2)))) \end{aligned} \qquad \text{[Modsum]}$$

Similar to the function modsum1 that is used to define addition, we need a function modsub2 in order to build subtraction. The function modsub2 makes use of an auxiliary function compl (complement) that maps d to $9 - d$.

$$\mathsf{compl}(d_1) = (\mathsf{base} \lhd \mathsf{eq1}(d_1,\, 1) \rhd \mathsf{P}(\mathsf{compl}(\mathsf{P}(d_1)))) \qquad \text{[Compl]}$$

$$\begin{aligned} \mathsf{modsub2}(d_1,\, d_2) \;=\; & (d_1 \lhd \mathsf{Z}(d_2) \\ & \rhd (\mathsf{compl}(d_2) \lhd \mathsf{Z}(d_1) \rhd \mathsf{modsub2}(\mathsf{P}(d_1),\, \mathsf{P}(d_2)))) \end{aligned} \qquad \text{[Modsub]}$$

The function borrow2 is the "subtraction-version" of the function carry1. It handles underflow in subtraction whereas carry1 handles overflow in addition.

$$d_1 <_\circ d_2 = (\mathsf{F} \lhd \mathsf{Z}(d_2) \rhd (\mathsf{T} \lhd \mathsf{Z}(d_1) \rhd \mathsf{P}(d_1) <_\circ \mathsf{P}(d_2))) \qquad \text{[LessD]}$$

$$\mathsf{borrow2}(d_1,\, d_2) = \mathsf{t2d}(d_1 <_\circ d_2) \qquad \text{[Borrow]}$$

4.2 Natural Numbers

Natural numbers are seen as lists of digits without leading zeros. We define a function rlz (remove leading zeros) which removes redundant occurrences of $\mathsf{0}$. In particular it removes the $\mathsf{0}$ in $[0]$ which is also redundant since the square brackets already prevent that zero is represented by a blank space.

Furthermore we define successor (S), predecessor (P), being zero (Z) and a function length. Note that the latter function is defined on lists of digits and not on naturals, i.e. $\mathsf{rlz}(D_1) = \mathsf{rlz}(D_2)$ does not imply $\mathsf{length}(D_1) = \mathsf{length}(D_2)$. Variables n, n', n_i range over natural numbers and D, D', D_i range over lists of digits.

$$\mathsf{rlz}(n) = (n \lhd \mathsf{E}(n) \overset{\circ}{\vee} \neg\, \mathsf{Z}(\text{↰}\, n) \rhd \mathsf{rlz}(\text{↳}\, n)) \quad \text{[RLZ1]}$$

$$\mathsf{S}(n) = \mathsf{rlz}(([1] \lhd \mathsf{E}(n) \rhd (\mathsf{S}(n\, \text{↲}) \mathbin{\circ\!\frown} [0] \lhd \mathsf{eq1}(n\, \text{↱}\, , \mathsf{base}) \rhd n\, \text{↲} \mathbin{\circ\!\frown} [\mathsf{S}(n\, \text{↱})]))) \quad \text{[SucL]}$$

$$\mathsf{P}(n) = \mathsf{rlz}((\mathsf{P}(n\, \text{↲}) \mathbin{\circ\!\frown} [\mathsf{base}] \lhd \mathsf{eq1}(n\, \text{↱}\, , 0) \rhd n\, \text{↲} \mathbin{\circ\!\frown} [\mathsf{P}(n\, \text{↱})])) \quad \text{[PredL]}$$

$$\mathsf{Z}(n) = \mathsf{E}(\mathsf{rlz}(n)) \quad \text{[ZL]}$$

$$\mathsf{length}(D) = ([\ ] \lhd \mathsf{E}(D) \rhd \mathsf{S}(\mathsf{length}(\text{↳}\, D))) \quad \text{[Len1]}$$

4.2.1 Standard Definitions

Below we define the recursive equations for addition, subtraction, multiplication, division and comparison (less than). In each case we choose a notation which is strict in the second argument, because it leads to better layout than strictness in the first argument.

$$n_1 \mathbin{+\!\circ} n_2 = (\mathsf{rlz}(n_1) \lhd \mathsf{Z}(n_2) \rhd \mathsf{S}(n_1 \mathbin{+\!\circ} \mathsf{P}(n_2))) \quad \text{[AddLsp]}$$

$$n_1 \multimap n_2 = (\mathsf{rlz}(n_1) \lhd \mathsf{Z}(n_2) \rhd \mathsf{P}(n_1 \multimap \mathsf{P}(n_2))) \quad \text{[SubLsp]}$$

$$n_1 \times_\circ n_2 = ([\ ] \lhd \mathsf{Z}(n_2) \rhd n_1 \mathbin{+\!\circ} n_1 \times_\circ \mathsf{P}(n_2)) \quad \text{[MultLsp]}$$

$$n_1 \mathbin{\div\!\circ} n_2 = (\mathsf{M} \lhd \mathsf{Z}(n_2) \rhd ([\ ] \lhd n_1 <_\circ n_2 \rhd \mathsf{S}((n_1 \multimap n_2) \mathbin{\div\!\circ} n_2))) \quad \text{[DivLsp]}$$

$$n_1 <_\circ n_2 = (\mathsf{F} \lhd \mathsf{Z}(n_2) \rhd (\mathsf{T} \lhd \mathsf{Z}(n_1) \rhd \mathsf{P}(n_1) <_\circ \mathsf{P}(n_2))) \quad \text{[LessLsp]}$$

4.2.2 Standard Algorithms

Instead of the recursive definitions above we will use the following alternative — and less standard — definitions for the same notions (which include more standard algoritms). Contrary to the situation in the first alternative, the operations are now defined in terms of list manipulation.

$$\frac{d_1 \;=\; \mathsf{carry1}(n_1 \Rsh,\, n_2 \Rsh),\;\; d_2 \;=\; \mathsf{modsum1}(n_1 \Rsh,\, n_2 \Rsh)}{\begin{aligned} n_1 \mathbin{+\!\circ} n_2 \;=\;& (\mathsf{rlz}(n_1) \lhd \mathsf{Z}(n_2)\\ & \rhd (\mathsf{rlz}(n_2) \lhd \mathsf{Z}(n_1) \rhd (n_1 \Ldsh \mathbin{+\!\circ} n_2 \Ldsh \mathbin{+\!\circ} [d_1]) \mathbin{{}^{\circ}\!\frown} [d_2]))\end{aligned}} \quad \text{[Add1]}$$

$$\frac{d_1 \;=\; \mathsf{borrow2}(n_1 \Rsh,\, n_2 \Rsh),\;\; d_2 \;=\; \mathsf{modsub2}(n_1 \Rsh,\, n_2 \Rsh)}{n_1 \multimap n_2 \;=\; (\mathsf{rlz}(n_1) \lhd \mathsf{Z}(n_2) \rhd (\mathsf{M} \lhd \mathsf{Z}(n_1) \rhd (n_1 \Ldsh \multimap n_2 \Ldsh \multimap [d_1]) \mathbin{{}^{\circ}\!\frown} [d_2]))} \quad \text{[Sub1]}$$

$$\mathsf{mult2}(n, d) = ([\,] \lhd \mathsf{Z}(d) \rhd n \mathbin{+\!\circ} \mathsf{mult2}(n, \mathsf{P}(d))) \quad \text{[DMult1]}$$

$$\begin{aligned} n_1 \times_\circ n_2 \;=\;& ([\,] \lhd \mathsf{Z}(n_1) \vee^\circ \mathsf{Z}(n_2)\\ & \rhd (n_1 \times_\circ n_2 \Ldsh) \mathbin{{}^{\circ}\!\frown} [0] \mathbin{+\!\circ} \mathsf{mult2}(n_1, n_2 \Rsh))\end{aligned} \quad \text{[Mult1]}$$

$$\mathsf{fill2}(n_1, n_2) = (\mathsf{M} \lhd \mathsf{Z}(n_2) \rhd (\mathsf{rlz}(n_2) \lhd n_1 <_\circ n_2 \rhd \mathsf{fill2}(n_1, n_2 \mathbin{{}^{\circ}\!\frown} [0]))) \quad \text{[Fill1]}$$

$$\mathsf{div2}(n_1, n_2) = (\mathsf{M} \lhd \mathsf{Z}(n_2) \rhd (0 \lhd n_1 <_\circ n_2 \rhd \mathsf{S}(\mathsf{div2}(n_1 \multimap n_2, n_2)))) \quad \text{[DDiv1]}$$

$$\frac{d \;=\; \mathsf{div2}(n_1, n_3)}{\mathsf{dr2}(n_1, n_2, n_3) \;=\; ([\,] \lhd n_3 <_\circ n_2 \rhd [d] \mathbin{{}^{\circ}\!\frown} \mathsf{dr2}(n_1 \multimap \mathsf{mult2}(n_3, d), n_2, n_3 \Ldsh))} \quad \text{[DR1]}$$

$$n_1 \mathbin{\div\!\circ} n_2 = \mathsf{dr2}(n_1, n_2, \mathsf{fill2}(n_1, n_2) \Ldsh) \quad \text{[Div1]}$$

$$\mathsf{less2}(n_1, n_2) = (\mathsf{F} \lhd \mathsf{E}(n_2) \rhd (\mathsf{less2}(\Rdsh n_1, \Rdsh n_2) \lhd \mathsf{eq1}(\Lsh n_2, \Lsh n_1) \rhd \Lsh n_1 <_\circ \Lsh n_2)) \quad \text{[Less1]}$$

$$\frac{n_1' \;=\; \mathsf{rlz}(n_1),\;\; n_1'' \;=\; \mathsf{length}(n_1'),\;\; n_2' \;=\; \mathsf{rlz}(n_2),\;\; n_2'' \;=\; \mathsf{length}(n_2')}{\begin{aligned} n_1 <_\circ n_2 \;=\;& (\mathsf{F} \lhd \mathsf{Z}(n_2)\\ & \rhd (\mathsf{T} \lhd \mathsf{Z}(n_1) \rhd (\mathsf{less2}(n_1', n_2') \lhd \mathsf{eq1}(n_1'', n_2'') \rhd n_1'' <_\circ n_2'')))\end{aligned}} \quad \text{[Less2]}$$

5 Conclusions

Of course, many more data types can be introduced in this style. Extensive expirimentation will be needed to asses the usefulness of the sequential data algebra format. Concerning the validity of these identities we mention that the resulting rewrite system has been tested extensively. Correctness proofs can be given in various styles. We mention that for instance the Bird-Meertens formalism and its descendants allow useful transformations of systems of recursion equations, which can provide just the kind of proofs needed. The correctness proof will work as follows: assume the correctness of the equations of paragraph 4.2.1 (on basis of obvious correspondence with the recursion equations for the given operators) and transform these equations until the ones of paragraph 4.2.2 are obtained. As a reference we mention Meijer [HM93]. Indeed E. Meijer has shown us that his proof style can easily be adapted to cover our cases as well. An interesting alternative is to use a proof-checker — e.g. the type theory based proof construction program Coq [CCF+95] — to validate the transformations just mentioned. For Coq-based proof-checking we refer to [Sel96].

We conclude, however, that four valued logic with McCarthy's sequential connectives provides a workable point of departure for the systematic development of data algebras. Many important mechanisms have not been addressed. We mention Cartesian products, coproducts (disjoint unions) and sets. Acknowledgements are due to Joris Hillebrand for his help in configuring the ASF meta environment and to Eelco Visser for many suggestions on how to use ASF+SDF and ToLaTeX.

A Divergence in the Iterative Functional Hull

Let $A = \bigcup_\sigma A_\sigma$ be a sequential data algebra (i.e. the canonical term algebra of an SDA specification) with signature $\Sigma = \Sigma_A$. We write $\mathrm{M}(\Sigma)$ for the fragment of Σ consisting of all of its sorts and all monadic (unary) operations. The *iterative functional hull* $\mathcal{L}_{\mathit{ifh}}(\Sigma)$ of A is the collection of operations defined by the following BNF grammar:

$$\mathcal{L}_{\mathit{ifh}}(\Sigma) ::= \mathrm{I} \mid \mathrm{M}(\Sigma) \mid \mathcal{L}_{\mathit{ifh}}(\Sigma){*\!\!\rightarrow}\ \mathcal{L}_{\mathit{ifh}}(\Sigma) \mid \mathcal{L}_{\mathit{ifh}}(\Sigma)\circ\mathcal{L}_{\mathit{ifh}}(\Sigma) \mid \mathcal{L}_{\mathit{ifh}}(\Sigma) \lhd \mathcal{L}_{\mathit{ifh}}(\Sigma) \rhd \mathcal{L}_{\mathit{ifh}}(\Sigma)$$

Furthermore, if $f{*\!\!\rightarrow}\ g \in \mathcal{L}_{\mathit{ifh}}(\Sigma)$ then $f : \sigma{\rightarrow}\mathit{Bool}$ and $g : \sigma{\rightarrow}\sigma$ for some sort σ, if $f \circ g \in \mathcal{L}_{\mathit{ifh}}(\Sigma)$ then $f : \tau{\rightarrow}\sigma'$ and $g : \sigma{\rightarrow}\tau$ for some τ, σ and σ', and if $f \lhd g \rhd h \in \mathcal{L}_{\mathit{ifh}}(\Sigma)$ then $f, h : \sigma{\rightarrow}\sigma'$ and $g : \sigma{\rightarrow}\mathit{Bool}$.

The semantics of $\mathcal{L}_{\mathit{ifh}}(\Sigma)$ expressions is given as follows: for each $f \in \mathcal{L}_{\mathit{ifh}}(\Sigma)$, $f : \sigma{\rightarrow}\sigma'$, we define a computability relation $f(..) \downarrow .. \subseteq A_\sigma \times A_{\sigma'}$ as the smallest relation such that $\mathrm{I}(d) \downarrow d$, $f(d) \downarrow [\![f(d)]\!]_A$, $g{\circ}f\,(d_1) \downarrow d_3$ whenever $f(d_1) \downarrow d_2$ and $g(d_2) \downarrow d_3$, $f \lhd g \rhd h\,(d_1) \downarrow d_2$ if either $g(d_1) \downarrow \mathsf{T}$ and $f(d_1) \downarrow d_2$ or $g(d_1) \downarrow \mathsf{F}$ and $h(d_1) \downarrow d_2$, $f \lhd g \rhd h\,(d_1) \downarrow \mathsf{D}$ if $g(d_1) \downarrow \mathsf{D}$ (idem for M), and $g{*\!\!\rightarrow}\ f\,(d_1) \downarrow d_2$ if $(g{*\!\!\rightarrow}\ f){\circ}f \lhd g \rhd \mathrm{I}(d_1) \downarrow d_2$. Then, we define $[\![f(d_1)]\!]_A = d_2$ if $f(d_1) \downarrow d_2$ and $[\![f(d_1)]\!]_A = \mathsf{D}$ otherwise. Notice that $[\![f(d_1)]\!]_A$ is always a normal form in A. Note also that $[\![..]\!]_A$ can straightforwardly be extended to "mixed" terms over $\Sigma \cup \mathcal{L}_{\mathit{ifh}}(\Sigma)$.

Given an SDA specification $\langle \Sigma, E \rangle$ of A its iterative functional hull is available as well. We write A_{ifh} for A extended with all operations of $\mathcal{L}_{\mathit{ifh}}(\Sigma)$. We notice that the iterative functional hull generates D as follows: let d be a proper value in A then

$$[\![\mathsf{T}{*\!\!\rightarrow}\ \mathrm{I}(d)]\!]_A = \mathsf{D}$$

whence $A_{ifh} \models \mathsf{T}* \to \mathrm{I}(d) = \mathsf{D}$. The equations for A describe the propagation of divergence D as well as the propagation of the meaningless value M. As an example consider (in a language with booleans and lists over booleans) the terms $t = \Lsh(\,[\;]\lrcorner\,{}^{\circ}\!\frown[\mathsf{T}* \to \mathrm{I}\,(\neg\mathsf{F})]\,)$ and $t' = \Lsh(\,[\mathsf{T}* \to \mathrm{I}\,(\neg\mathsf{F})]\,{}^{\circ}\!\frown\!\llcorner[\;]\,)$. The propagation equations for D and M prescribe that an implementation of t must converge (to some kind of error message) whereas an implementation of t' diverges.

References

[BBR95] J.A. Bergstra, I. Bethke, and P. Rodenburg. A propositional logic with 4 values: true, false, divergent and meaningless. *J.Applied Non-Classical Logics*, 5(2):199–217, 1995.

[BK87] J.A. Bergstra and J.W. Klop. *Semicomplete termherschrijfsystemen.* Programmatuurkunde. Kluwer, Deventer, 1987. In Dutch.

[BT83] J.A. Bergstra and J.V. Tucker. Initial and final algebra semantics for data type specifications: two characterisation theorems. *SIAM Journal on Computing*, 12:76–90, 1983.

[BT95] J.A. Bergstra and J.V. Tucker. Equational specifications, complete term rewriting systems and computable and semicomputable algebras. *Journal of the ACM*, 1995. accepted for publication.

[CCF+95] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual. Version 5.10*, 1995.

[HK89] J. Heering and P. Klint. The syntax definition formalism SDF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Frontier Series, pages 283–296. Addison-Wesley, 1989.

[HM93] G. Hutton and E. Meijer. Functional pearls. back to basics: Deriving representation changers functionally. *J. Functional Programming*, 1(1):1–8, 1993.

[Kli95] P. Klint. *The ASF+SDF Meta-environment. User's Guide.* Centre for Mathematics and Computer Science, Amsterdam, 1995.

[McC63] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirshberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[Sel96] M.P.A. Sellink. *Computer-Aided Verification of Protocols, The Type Theoretic Approach.* PhD thesis, Utrecht University, 1996.

[vW95] J.J. van Wamel. *Verification Techniques for Elementary Data Types and Retransmission Protocols.* PhD thesis, University of Amsterdam, 1995.

[Wie91] F. Wiedijk. *Persistence in Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991.

[WZ95] H.R. Walters and H. Zantema. Rewrite systems for integer arithmetic. In J. Hsiang, editor, *Proceedings RTA '95*, volume 914, pages 324–338. Springer-Verlag, LNCS, 1995.