Reprint from
The *Mathematica* Conference

June, 1992   Boston, MA

936.9.10.1992

Abstract Data Types

Roman E. Maeder

1 Introduction

Abstract data types are both a theoretically well-defined concept and a useful tool for program development. Following the principles of abstract data type design, one arrives at a clear separation of specification and implementation.

Abstract data types are defined in terms of type names, function names, and equations. These can be realized in *Mathematica* very easily. The equations become rewrite rules. The interactive nature of *Mathematica* makes it well suited for rapid prototyping and testing of designs.

We will have a short look at the theory behind abstract data types and then show how they can be implemented in *Mathematica*. Fully worked-out examples will be discussed. We will show how user-defined data types can be made to behave like built-in ones.

Prerequisites: Participants are expected to have a basic understanding of *Mathematica* programming, including pattern matching, and functional programming. This corresponds roughly to Chapters 2–6 of *Programming in Mathematica*, Addison-Wesley, 1991.

2 The Definition of Abstract Data Types

An *algebra* is a mathematical structure that consists of a set (of elements) and a number of operations on these elements. The integers with the usual arithmetic operations are an example. Since we often use more than one type of data, we need to expand this notion into *many-sorted* algebras. The task of defining the operations becomes then a bit tricky, since we have to specify the type of all operands.

An *abstract data type* is specified in terms of three parts:

- a set S of *sort names*
- a set Σ of *function names* (or operator symbols)
- a set E of *equations*

The triplet $\langle S, \Sigma, E\rangle$ is called a *specification*. The sorts can be thought of the names of the "types" we are going to model. Usually there are more than one; we may use auxiliary types, such as **bool**, standing for Boolean values (truth values). We will print sort names in boldface, all lowercase.

In our first example **Nat1**, we will model the nonnegative integers, with Boolean values as an auxiliary type. Therefore S consists of two elements:

$$S = \{\mathbf{int}, \mathbf{bool}\}$$

The function names are broken down according to their "arity", that is the number and types of arguments and the return type. The arity is described by a sequence of elements from S, the last one being the return type, the other ones describing the argument types in the proper order. Formally, the set Σ consists of sets $\Sigma_{w,s}$, with $s \in S, w \in S^*$. (This says that w is a (possibly empty) sequence of elements of S, and s is a single element of S, the return type). Elements of $\Sigma_{\lambda,s}$ are called *constants* since they are functions without arguments (λ denotes the empty sequence).

Continuing our example, here are a few commonly defined function names:

$$\begin{array}{lcl}
\Sigma_{\mathbf{int}} & = & \{\mathbf{z}\} \\
\Sigma_{\mathbf{bool}} & = & \{\mathbf{f}, \mathbf{t}\} \\
\Sigma_{\mathbf{int\,int}} & = & \{\mathbf{s}\} \\
\Sigma_{\mathbf{int\,bool}} & = & \{\mathbf{isz}\} \\
\Sigma_{\mathbf{bool\,bool}} & = & \{\mathbf{not}\} \\
\Sigma_{\mathbf{int\,int\,int}} & = & \{\mathbf{add}, \mathbf{mult}\}
\end{array}$$

This says, for example, that **z** is a constant of type **int** and that **add** is a function of two arguments of type **int**, and returning a result also of type **int**. Σ is called the *signature*. Please note that we have not given these function names any meaning yet, even though eventually we will, of course, think of **add** as the addition function on integers.

There is nothing in the definitions so far that forces us to use the addition function as meaning for the function named **add**. In order to make our intention clear, we can give a number of equations that must be satisfied by the functions we choose to implement the operations in Σ. In order to write down equations, we will use the operation symbols themselves as purely formal functions together with variable symbols for each sort. Here are the equations that enforce the usual meaning for our operations. Variables are printed in italics. Their type is obvious from the functions in which they appear. If this were not the case we would have to formally define the types of the variables.

$$\begin{array}{rcl}
\mathbf{isz}(\mathbf{z}) & = & \mathbf{t} \\
\mathbf{isz}(\mathbf{s}(n)) & = & \mathbf{f} \\
\mathbf{not}(\mathbf{t}) & = & \mathbf{f} \\
\mathbf{not}(\mathbf{f}) & = & \mathbf{t} \\
\mathbf{add}(n, \mathbf{z}) & = & n \\
\mathbf{add}(n, \mathbf{s}(m)) & = & \mathbf{s}(\mathbf{add}(n, m)) \\
\mathbf{mult}(n, \mathbf{z}) & = & \mathbf{z} \\
\mathbf{mult}(n, \mathbf{s}(m)) & = & \mathbf{add}(\mathbf{mult}(n, m), n)
\end{array}$$

We are now ready to define Σ-*algebras*. A Σ-algebra is a mathematical structure that "fits" the signature Σ. It consists of *sets* for the sorts from S and of mappings corresponding to the operator symbols from Σ.

An *S-sorted* Σ-*algebra* **B** consists of a set B_s for each $s \in S$, called the *carrier set* of sort s, and a function

$$\sigma_B : B_{s_1} \times B_{s_2} \times \cdots \times B_{s_n} \longrightarrow B_s$$

for each $\sigma \in \Sigma_{w,s}$ with $w = s_1 s_2 \ \ldots \ s_n$ (*i.e.*, a function taking arguments from the carrier sets corresponding to its arity).

We can now endow our example **Nat1** with its intended meaning, using the nonnegative integers as carrier set of sort **int**. In order to describe the functions we use *Mathematica* notation.

$$\begin{array}{lcl}
B_{\mathbf{int}} & = & \{0, 1, 2, \ldots\} \\
B_{\mathbf{bool}} & = & \{\texttt{False}, \texttt{True}\} \\
\mathbf{z}_B & = & \texttt{0\&} \\
\mathbf{f}_B & = & \texttt{False\&} \\
\mathbf{t}_B & = & \texttt{True\&} \\
\mathbf{s}_B & = & \texttt{Function[n, n + 1]} \\
\mathbf{isz}_B & = & \texttt{Function[n, n == 0]} \\
\mathbf{not}_B & = & \texttt{Function[b, !b]} \\
\mathbf{add}_B & = & \texttt{Plus} \\
\mathbf{mult}_B & = & \texttt{Times}
\end{array}$$

It is customary to leave out the empty argument sequence for constants. Instead of writing `(False&)[]` we will simply write `False` from now on.

An S-sorted Σ-algebra **B** *satisfies the equations* if for each substitution of elements of the appropriate sorts for all variables in an equation the left and right side of the equation become equal. A Σ-algebra satisfying the equations E is called a Σ, E-algebra. One can prove by induction that the algebra **B** given above satisfies the equations of **Nat1**.

3 The Typical, or Initial, Σ-Algebra

A Σ, E-algebra does not yet capture our idea of a "data type". There are usually several different (nonisomorphic) Σ-algebras having additional properties (not required by the equations E) and it is not clear which one is the "right one". There is, however, one special algebra that satisfies the equations and has no other spurious properties. It is named $T_{\Sigma,E}$ and is constructed from the *term algebra* T_Σ (whose carrier sets consist of terms built by using the operator symbols as functions, just as in the equations above) by identifying any terms that are forced equal by the equations. The details of this abstract construction can be found in the literature about abstract data types [ADJ75, ADJ78]. This algebra is called the *initial* Σ, E-algebra. It is this algebra that captures our notion of abstract data type.

If the equations are sufficiently well behaved, we can model this algebra in *Mathematica*. In our example the equations can be thought of as *reductions*, always replacing the left side by the (simpler) right side. In this way there is one special term in each class of terms identified by the equations. It is the term that is in normal form with respect to the reductions derived from the equations. Any other term is transformed into normal form by the applicable reduction rules. We can do this in *Mathematica* by turning the equations into definitions.

The term algebra T_Σ can be modeled in *Mathematica* by using the operator symbols as unspecified functions. To implement $T_{\Sigma,E}$ we define the equations as rewrite rules. Any term not in normal form will be simplified into normal form by these rules as we have seen. This technique is called *executable specification*. Here is the code for **Nat1**. Note that we turn the variables in the equations into "named blanks" in the rules.

```
(* the initial algebra for the specification Nat1 *)

(* sorts *)

int
bool

(* Sigma *)

{z}                  (* -> int *)
{t, f}                  (* -> bool *)
{s}                  (* int -> int *)
{isz}                  (* int -> bool *)
{not}                  (* bool -> bool *)
{add, mult}          (* int int -> int *)


(* equations turned into rewrite rules *)

isz[z] := t
isz[s[n_]] := f

not[t] := f
not[f] := t

add[n_, z] := n
add[n_, s[m_]] := s[add[n, m]]

mult[n_, z] := z
mult[n_, s[m_]] := add[mult[n, m], n]
```

The abstract data type **Nat1** in *Mathematica*.

Here are a few sample computations with **Nat1**. The integer n is represented as n-fold nesting of **s** applied to **z**.

This performs the addition $1 + 2 = 3$.

```
In[2]:= add[ s[z], s[s[z]] ]

Out[2]= s[s[s[z]]]
```

The product of 2 and 3. A trace of this evaluation would show the rather large amount of computation going on.

```
In[3]:= mult[ s[s[z]], s[s[s[z]]] ]

Out[3]= s[s[s[s[s[s[z]]]]]]
```

This is the test `1 =! 0`, returning true.

```
In[4]:= not[isz[s[z]]]

Out[4]= t
```

4 A Practical Approach

You wouldn't want to perform arithmetic with this abstract version of integers. We can work with a more suitable Σ, E-algebra as long as it is isomorphic to $T_{\Sigma,E}$. We have already given such an implementation of **Nat1** in terms of *Mathematica*'s built-in integers and arithmetic operations. One can show that our implementation of **Nat1** is isomorphic to the abstract term algebra given above. The integers $0, 1, 2, \ldots$ are in one-to-one correspondence with the terms `0, s[0], s[s[0]], ...`.

Still, this purely formal approach is of great value. It allows the formulation of the intended behavior of a data type without referring to its implementation. Any implementation can later be used for the actual computation. Moreover if all application code uses exclusively the operator symbols from Σ the implementation can be changed without changing any application code! This is probably the most important aspect of using abstract data types. It allows a clear separation between specification and implementation. Often there is no need to go all the way to a formal specification $\langle S, \Sigma, E\rangle$, but it suffices to classify the functions into *constructors*, *selectors*, *constants*, and *predicates*.

Consider the example **Mod**, intended to model $\mathbf{Z}_p$, the ring of integers modulo some positive integer p. We can build on top of **Nat1**, extending S by a sort **mod**, and adding the following operations:

$$\begin{array}{lcl}\Sigma_{\mathbf{int}} & = & \{\mathbf{p}\}\\ \Sigma_{\mathbf{mod\,int}} & = & \{\mathbf{rep}\}\\ \Sigma_{\mathbf{int\,mod}} & = & \{\mathbf{makemod}\}\\ \Sigma_{\mathbf{mod\,mod\,mod}} & = & \{\mathbf{add}, \mathbf{mult}\}\\ \Sigma_{\mathbf{int\,int\,int}} & = & \{\mathbf{rem}\}\end{array}$$

There is no problem with using the same operation name (here **add** and **mult**) in different sets $\Sigma_{w,s}$. They are distinguished by the type of arguments and return values. **rep** is meant to give a *representative* of a modular number, an integer that can be used to implement arithmetic in **mod**, as we will see in the equations below. **rem** is the integer remainder function, whose definition is omitted. It is needed only to express the desired properties of **rep** in the equations. **makemod** turns an integer into a modular number. Here are the equations:

$$\begin{array}{rcl}\mathbf{makemod}(\mathbf{rep}(m)) & = & m\\ \mathbf{rem}(n - \mathbf{rep}(\mathbf{makemod}(n)), \mathbf{p}) & = & \mathbf{0}\\ \mathbf{add}(m_1, m_2) & = & \mathbf{makemod}(\mathbf{add}(\mathbf{rep}(m_1), \mathbf{rep}(m_2)))\\ \mathbf{mult}(m_1, m_2) & = & \mathbf{makemod}(\mathbf{mult}(\mathbf{rep}(m_1), \mathbf{rep}(m_2)))\end{array}$$

The second equation says that the representative of a modular number made from an integer n is congruent to n modulo p ("-" being integer subtraction). This is usually expressed as

$$\mathbf{rep}(\mathbf{makemod}(n)) \equiv n \pmod{p}.$$

Addition and multiplication are then defined in terms of their integer counterparts through the use of **rep**.

Here is one possible implementation, using expressions of the form `mod[`n`]`, n an integer, to hold elements of $B_{\mathbf{mod}}$. The use of the built-in function `Mod[`n`, `p`]` in **makemod** ensures that a unique representative is chosen (an integer in the range $0 \ldots p-1$). Putting this computation into the constructor removes all worries about the choice of representative from the rest of the code and trivially satisfies the first and second equation.

```
makemod[n_Integer] := mod[ Mod[n, p] ]
rep[ mod[n_] ] := n

mod/: m1_mod + m2_mod := makemod[ rep[m1] + rep[m2] ]
mod/: m1_mod * m2_mod := makemod[ rep[m1] * rep[m2] ]
```

Modular numbers as abstract data type.

Note that we have *overloaded* the built-in addition and subtraction functions. This corresponds directly to the use of the same names in the specification.

This sets the modulus to 5 for the computations that follow.

```
In[2]:= p = 5

Out[2]= 5
```

This makes two modular numbers and assigns them to the two variables `m2` and `m3`.

```
In[3]:= m2 = makemod[2]; m3 = makemod[3]

Out[3]= mod[3]
```

Their sum is 0 (modulo 5).

```
In[4]:= m2 + m3

Out[4]= mod[0]
```

Their product is 1 (modulo 5).

```
In[5]:= m2 m3

Out[5]= mod[1]
```

We call **makemod** a *constructor*, because it creates values of a certain type (here **mod**). **rep** is a *selector*, since it returns parts of an expression (**mod** elements have only one part, the representative of the modular number). Later on we will also encounter *predicates*. These are functions that return a Boolean value.

5 Design Principles for Abstract Data Types

From the previous example, we can derive the general recommended way of defining and implementing an abstract data type.

This is how to define a data type:

- Define the names of the sorts to be used.
- Define the constants to be used.
- Define constructors and selectors.
- Write down the equations that should hold.

Here is how to implement a data type:

- Choose a representation for elements of the sorts defined.
  Usually you can use a normal expression having the name of the sort as head. This corresponds roughly to a record in many programming languages.
- Derive rules for normal forms for the data elements.
  This ensures that elements are stored in a unique way. The rules are derived from the equations and can normally be put into the constructors.
- Define constructors and selectors.
  These are the only operations that are allowed access to the details of the internal representations.
- Define the other operations.
  An operation for which exactly one equation exists can usually be turned into a simple rewrite rule. (This is called a *derived* operation.) The variables in the left side of the rules are in the form *n_type*, restricting the arguments of the functions implemented to the correct types. Use selectors only to access parts of the data elements on the right side of the rules.
- Use overloading where appropriate.
  Overloading built-in functions is most easily done by defining up-values, rules of the form `g/: f[n_g,...] := ...`.
- Choose a suitable output representation of data elements.
  If you do this, you can consider declaring the sorts themselves in the private implementation part of your package. Users are then prevented from accessing them directly.
- Define automatic type conversions where useful.
  This is a convenience only.

In addition, the usual framework of a package should be used, with documentation for the exported functions in the usual way. Below is a sample package for computation with modular numbers that shows some of the ideas above, continuing the example **Mod**.

```
BeginPackage["Modular`"]

SetModulus::usage = "SetModulus[p] sets the modulus to be used."
MakeModular::usage = "MakeModular[n] returns n mod p."

Begin["`Private`"]

`theModulus      (* a private static variable *)
`mod             (* the private name of the data type *)

(* constructors *)

MakeModular[n_Integer] := mod[ Mod[n, theModulus] ]

SetModulus[p_Integer?Positive] := (theModulus = p)

(* Selectors *)

Representative[mod[n_]] := n

(* Arithmetic *)

mod/: a_mod + b_mod :=
        MakeModular[ Representative[a] + Representative[b] ]
mod/: a_mod * b_mod :=
        MakeModular[ Representative[a] * Representative[b] ]
mod/: a_mod ^ q_Integer :=
        MakeModular[ PowerMod[Representative[a], q, theModulus] ]

mod/: a_mod + b_Integer := a + MakeModular[b]
mod/: a_mod * b_Integer := a * MakeModular[b]

(* Output Formatting *)

Format[m_mod] :=
        SequenceForm[Representative[m], " mod ", theModulus]


End[]

EndPackage[]
```

`Modular.m`: A package for modular numbers.

6 More Examples

6.1 LISP

The data structure of the programming language LISP can easily be modeled in *Mathematica*. We treat all *Mathematica* atoms as LISP atoms.

- Atoms: numbers, symbols, strings (all *Mathematica* atoms)
- Constants: `nil` and all atoms.
  Note that this gives an infinite number of constants. Nothing in the theory of abstract data types demands a finite number of operation symbols.
- Constructors: `cons[`e_1`, `e_2`]`, `list[`e_1`, `e_2`, …, `e_n`]`, $n \geq 0$,
  (`list` stands for an infinite number of operator symbols, one for each number of arguments.) `list` is not a fundamental constructor, but a very convenient derived operator that makes it easy to build lists.
- Selectors: `car[`e`]`, `cdr[`e`]`
- Predicates: `pairQ[`e`]`, `atomQ[`e`]`, `nullQ[`e`]`

- Equations:

$$
\begin{aligned}
\texttt{car[cons[}e_1\texttt{, }e_2\texttt{]]} &= e_1 \\
\texttt{cdr[cons[}e_1\texttt{, }e_2\texttt{]]} &= e_2 \\
\texttt{atomQ[}a\texttt{]} &= \texttt{True} \text{ (one equation for each atom } \texttt{a}\text{)} \\
\texttt{atomQ[cons[}e_1\texttt{, }e_2\texttt{]]} &= \texttt{False} \\
\texttt{pairQ[}e\texttt{]} &= \texttt{!atomQ[}e\texttt{]} \\
\texttt{nullQ[nil]} &= \texttt{True} \\
\texttt{nullQ[cons[}e_1\texttt{, }e_2\texttt{]]} &= \texttt{False} \\
\texttt{list[]} &= \texttt{nil} \\
\texttt{list[}e_1\texttt{, }e_2\texttt{, }\ldots\texttt{, }e_n\texttt{]} &= \texttt{cons[}e_1\texttt{, list[}e_2\texttt{, }\ldots\texttt{, }e_n\texttt{]]}, n > 0
\end{aligned}
$$

(Note that the number of equations is infinite. There is one equation for `atomQ` for each atom a and one equation for `list` for each $n \geq 0$.)

The implementation is simple. We use `cons` as head for the LISP cells (this corresponds to a constructor without any rules attached to it). The auxiliary function `listToList` is used to speed up the special output formatting that makes these lists print the way they do in LISP.

Here is a sample LISP list.

```
In[2]:= t = list[a, b, c, d, e]

Out[2]= (a b c d e)
```

This is a typical recursive LISP definition for appending an element to a list.

```
In[3]:= append[l_, e_] :=
          If[ nullQ[l], list[e],
              cons[car[l], append[cdr[l], e]] ]
```

This is a (slow) way of reversing a list.

```
In[4]:= reverse[l_] :=
          If[ nullQ[l], nil,
              append[reverse[cdr[l]], car[l]] ]
```

It works just as it does in LISP.

```
In[5]:= reverse[t]

Out[5]= (e d c b a)
```

In LISP, we cannot take the `car` of an atom. In our theory this leads merely to a term that cannot be simplified further since no equations apply to it.

```
In[6]:= car[atom]

Out[6]= car[atom]
```

```
BeginPackage["Lisp`"]

cons::usage = "cons[a, b] gives the dotted pair (a .b)."
car::usage = "car[pair] gives the first element of pair."
cdr::usage = "cdr[pair] gives the second element of pair."
list::usage = "list[e1, e2,..., en] gives the Lisp
        list (e1 e2 ... en)."

nil::usage = "nil is the empty Lisp list."

pairQ::usage = "pairQ[e] is true, if e is a dotted pair."
atomQ::usage = "atomQ[e] is true, if e is an atom."
nullQ::usage = "nullQ[l] is true, if l is the empty list."

Begin["`Private`"]

$RecursionLimit = Infinity (* necessary for long lists *)

(* constructors *)

list[] = nil
list[e_, r___] := cons[e, list[r]]

(* selectors *)

car[cons[e_, l_]] := e
```

```
cdr[cons[e_, l_]] := l

(* predicates *)

atomQ[_?AtomQ] = True  (* leave undefined otherwise *)
atomQ[_cons] = False
pairQ[e_] := !atomQ[e]
nullQ[nil] = True
nullQ[_cons] = False

(* output formats *)

listToNest[nil] := {}
listToNest[c_cons] := {car[c], listToNest[cdr[c]]}
listToList[l_] := Flatten[listToNest[l]]

listQ[nil] = True      (* this atom is also a list *)
listQ[l_cons] := listQ[cdr[l]]
listQ[_] = False

Format[nil] = "()"
Format[l_cons?listQ] :=
    SequenceForm["(", Infix[listToList[l], " "], ")"]
Format[l_cons] := SequenceForm["(", Infix[l, " . "], ")"]

(* fix Infix problem for functions with one argument *)

protected = Unprotect[Infix]
Infix[_[e_], h_:Null] := e
Protect[Evaluate[protected]]

End[]
Protect[ car, cdr, cons, pairQ, atomQ, nullQ, list ]
EndPackage[]
```

LISP in *Mathematica*

■ 6.2 Fractal Curves

In Vol. 1, No. 4 of *The Mathematica Journal* I presented a data type for *fractal curves*. There I defined a type **curve** and then overloaded Show[] so that such a curve can be plotted as if it were one of the existing graphics type, such as Graphics, SurfaceGraphics, etc.

■ 6.3 Infinite Lists

This code is taken from "Computations with Infinite Structures," *The Mathematica Journal*, Vol. 1, No. 2. It shows how infinite lists can be implemented. They are made to behave as much as possible like their finite counterparts, the built-in List structures. It is even possible to thread them over listable functions.

```
(*
  Functions for manipulating finite and infinite streams.
  The following system functions know about streams:
  First, Rest, Take, Part, Map, Select, Join, Thread, Accumulate,
  ReplaceAll, Nest, MemberQ.
  Listable functions thread over streams as they do over lists.
  Beware of $RecursionLimit and $IterationLimit!
*)

(* This is an updated version of the package for the Journal,
   Vol. 1, No. 2 *)

BeginPackage["Streams`"]

Stream::usage = "Stream is the head of streams."
EmptyStream::usage = "EmptyStream returns an empty stream."
EmptyStreamQ::usage = "EmptyStreamQ[s] is true if s is
        the empty stream, false otherwise."
MakeStream::usage = "MakeStream[e, s] make a new stream by prepending
        the element e in front of the stream s."
```

```
RememberedStream::usage = "RememberedStream[s] makes a new stream
        from s in which delayed parts are evaluated only once."
First::usage = StringJoin[ First::usage, " First[stream] returns
        the first element of a stream." ]
Rest::usage = StringJoin[ Rest::usage, " Rest[stream] returns stream
        without its first element." ]
Map::usage = StringJoin[ Map::usage, " Map[f, s] returns a stream
        with f applied to each element of s." ]
Select::usage = StringJoin[ Select::usage, " Select[s, crit] returns
        a stream of only those elements of stream that satisfy crit." ]
Join::usage = StringJoin[ Join::usage, " Join[s1, s2, ...]
        concatenates the streams si." ]
Union::usage = StringJoin[ Union::usage, " Union[s] returns a stream
        with all duplicate elements removed." ]
Interleave::usage = "Interleave[s1, s2, ...] interleaves the streams si."
Thread::usage = StringJoin[ Thread::usage, " Thread[f[s1, s2, ...]]
        threads f over the streams si." ]
Take::usage = StringJoin[ Take::usage, " Take[s, n] returns the
        first n elements of a stream s." ]
Accumulate::usage = StringJoin[ Accumulate::usage, " Accumulate[f, s]
        returns the stream {s1, f[s1, s2], f[s1, f[s2, s3]],...}." ]
FoldStream::usage = "FoldStream[f, x, s]
        returns the stream {x, f[x, s1], f[x, f[s1, s2]],...}."
Nest::usage = StringJoin[ Nest::usage, " Nest[f, x]
        returns the stream {x, f[x], f[f[x]], ...}." ]
Part::usage = StringJoin[ Take::usage, " s[[i]] returns the
        i-th element of a stream s." ]
ConstantStream::usage = "ConstantStream[c] generates an infinite stream
        of c's."
RangeStream::usage = "RangeStream[(low), high, (step)] generates the
        stream of expressions from low to high with stride step.
        (high may be set to Infinity, of course.)"

Begin["`Private`"]

protected = Unprotect[Thread, Nest]

EmptyStream = {}

SetAttributes[Stream, HoldRest]
SetAttributes[MakeStream, HoldRest]        (* THIS IS WHY IT WORKS! *)

MakeStream[e_, s_] := Stream[e, s]

RememberedStream[Stream[e_, s_]] :=
        Module[{sid}, Stream[e, s, sid]]

Stream[e, l_List] := Prepend[l, e]

Stream /: First[Stream[ e_, __ ]] := e
Stream /: Rest[Stream[ _, r_ ]] := r
Stream /: Rest[ Stream[ _, r_, sid_ ] ] :=
              If[ ValueQ[sid], sid, sid=RememberedStream[r] ]

(* {s1, s2,..., sn} *)

Stream /: Take[s_Stream, 0] := EmptyStream
Stream /: Take[s_Stream, n_Integer?Positive] :=
        First /@ NestList[ Rest, s, n-1 ]

(* s[[i]] *)

Stream /: Part[s_Stream, n_Integer?Positive] := First[Nest[Rest, s, n-1]]
Stream /: Part[s_Stream, n1_, n__] := Part[Part[s, n1], n]


EmptyStreamQ[_Stream] = False        (* always at least one element *)
EmptyStreamQ[l_List] := Length[l] == 0

(* f /@ s --> { f[s1], f[s2], ... } *)

Stream /: Map[f_, Stream[h_, r_]] :=
        MakeStream[ f[h], Map[f, r] ] (* faster special case *)
Stream /: Map[f_, s_Stream] :=
        MakeStream[ f[First[s]], Map[f, Rest[s]] ]
```

```
(* { si_1, si_2, ... }, with p[si_j] true *)

Stream /: Select[Stream[h_, r_], p_] :=
    If[ p[h], MakeStream[ h, Select[r, p] ], Select[r, p] ]
Stream /: Select[s_Stream, p_] :=
    If[ p[First[s]], MakeStream[ First[s], Select[Rest[s], p] ],
                     Select[Rest[s], p]
    ]

(* {s11, s12, s13,..., s21, s22, s23, ... } *)

(* these rules are bit tricky. If all streams are lists, the usual code
 * for Join takes over. We need these rules for the case where at least
 * one of them is a Stream[]. The mixed case introduces these complexities.
 *)
Stream /: Join[Stream[h_, r_], s__] := MakeStream[h, Join[r, s]]
Stream /: Join[s1_Stream, s__] :=
        MakeStream[First[s1], Join[Rest[s1], s]]
Stream /: Join[EmptyStream, s2___, s3_Stream, s4___] := Join[s2, s3, s4]

(* {s11, s21,..., sn1, s12, s22,..., sn2, s13, ... }
   empty streams are removed from list  *)

Interleave[s_] := s
Interleave[EmptyStream, s__] := Interleave[s]
Interleave[Stream[h_, r_], s__] := MakeStream[ h, Interleave[s, r] ]
Interleave[s1_Stream, s__] :=
        MakeStream[ First[s1], Interleave[s, Rest[s1]] ]


Stream /: Union[Stream[h_, r_]] :=
        MakeStream[ h, Select[Union[r], # != h&] ]
Stream /: Union[s_Stream] :=
        MakeStream[ First[s], Select[Union[Rest[s]], # != First[s]&] ]

Stream /: MemberQ[s_Stream, el_] :=
        If[ First[s] === el, True, MemberQ[Rest[s], el] ]

(* {f[s11, s21,..., sn1], f[s12, s22,..., sn2], ...} *)

Thread[f_[s__Stream]] :=
        MakeStream[ f @@ First /@ {s}, Thread[f @@ Rest /@ {s}]]

(* make streams threadable *)

SetAttributes[stag, HoldAll]

restrules = { stag[x___, Stream[_, r_], z___] :> stag[x, r, z],
              stag[x___, s_Stream, z___] :> stag[x, Rest[s], z] }
Stream /: f_Symbol[a___, b_Stream, c___] :=
        With[{firsts = Replace[#, s_Stream :> First[s]]& /@ List[a, b, c],
              rests = stag[a, b, c] //. restrules},
             MakeStream @@ (Hold[ f @@ firsts, rests ] /. stag -> f)
        ] /; MemberQ[Attributes[f], Listable]

(* FoldStream, like FoldList *)

Stream /: FoldStream[f_, x_, Stream[h_, r_]] :=
        MakeStream[ x, FoldStream[ f, f[x, h], r ] ]
Stream /: FoldStream[f_, x_, s_Stream] :=
        MakeStream[ x, FoldStream[ f, f[x, First[s]], Rest[s] ] ]
Stream /: Accumulate[f_, s_Stream] :=
        FoldStream[f, First[s], Rest[s]]

(* Note: Fold[] doesn't make sense for infinite streams! *)

(* rules thread over streams *)

Stream /: s_Stream /.  rules_ := # /.  rules & /@ s
Stream /: s_Stream //. rules_ := # //. rules & /@ s

Nest[f_, x_] := MakeStream[ x, Nest[f, f[x]] ]

ConstantStream[c_] := MakeStream[c, ConstantStream[c]]

RangeStream[low_:1, high_, step_:1] :=
        If[ (high - low)/step < 0,
```

```
                EmptyStream,
                With[{newl = low + step},
                    MakeStream[low, RangeStream[newl, high, step]]
                ],
                EmptyStream
        ]

(* formats *)

Format[ Stream[e_, s_, Optional[Blank[], Null]] ] :=
    SequenceForm[ "{", e, ", ", Short[HoldForm[s]], " ...}" ]

Protect[ Evaluate[protected] ]

End[]

Protect[ EmptyStream, MakeStream, RememberedStream, Interleave,
         FoldStream, ConstantStream, RangeStream ]
SetAttributes[Stream, {ReadProtected, Protected}]

EndPackage[]
```

Streams: infinite lists.

7 References

[ADJ78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In Raymond T. Yeh, editor, *Current Trends in Programming Methodology IV*. Prentice-Hall, 1978.

[ADJ75] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Abstract data types and correctness of data representations. In Allen Klinger et al., editors, *Proceedings of the Conference on Computer Graphics, Pattern Recognition and Data Structure*, 1975.

[Maeder91] Roman E. Maeder. *Programming in Mathematica*. Addison-Wesley, Second Edition, 1991.

[Scott76] Dana S. Scott. Data types as lattices. *SIAM Journal of Computing*, 5:522–587, 1976.

[Stoy80] J. E. Stoy. Foundations of denotational semantics. In D. Bjorner, editor, *Abstract Software Specification*, volume 86 of *SLNCS*. Springer-Verlag, New York, 1980.

[Wirsing90] Martin Wirsing. Algebraic specification. In Jan van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*. Elsevier, 1990.