Four sorting algorithms for the price of one

We consider (finite) bags and (finite) lists of comparable elements (like integers). Lists can be considered as representations of bags, in the following way. We define a function $\mathcal{B}$ from lists to bags by:

$$\begin{aligned}
\mathcal{B}\cdot[\,] &= \emptyset \\
\mathcal{B}\cdot[b] &= \{b\} \\
\mathcal{B}\cdot(s +\!\!+ t) &= \mathcal{B}\cdot s + \mathcal{B}\cdot t
\end{aligned}$$

This is a proper definition because bag summation, like list catenation, is associative. Actually, the only difference between bags and lists is that $+$ is symmetric whereas $+\!\!+$ is not.

We use $\in$ for the "is an element of" relation both for bags and lists, with the obvious meaning; for example, for every element b and list s we have:

$$b \in s \;\equiv\; b \in \mathcal{B}\cdot s$$

Similarly, we use $\#$ for the "number of elements of" function both for bags and lists, with the equally obvious meaning. Thus, we have, for example:

$$\# s \;=\; \#(\mathcal{B}\cdot s)$$

The ascending lists are the ones satisfying predicate $\mathcal{A}$ defined by:

$$\begin{aligned}
\mathcal{A}\cdot[\,] &\equiv \text{true}\\
\mathcal{A}\cdot[b] &\equiv \text{true}\\
\mathcal{A}\cdot(s+\!\!+t) &\equiv \mathcal{A}\cdot s \wedge \mathcal{A}\cdot t \wedge (\forall b,c : b\in s \wedge c\in t : b\leq c)
\end{aligned}$$

For every bag $\overset{a}{\smile}$ unique ascending list representing that bag exists. That such list exists will be shown in the sequel, that it is unique follows from the following property, which we shall not prove here.

<u>property:</u> for all lists s,t:

$$\mathcal{A}\cdot s \wedge \mathcal{A}\cdot t \wedge \mathcal{B}\cdot s = \mathcal{B}\cdot t \quad\Rightarrow\quad s=t$$

□

We now investigate the function L with the following specification.

<u>specification:</u> L has type Bag → List, with:

$$(\forall x :: \mathcal{B}\cdot(L\cdot x) = x \wedge \mathcal{A}\cdot(L\cdot x))$$

□

From this specification we derive a definition for L, as follows. For the time being we confine our attention to the conjunct $\mathcal{B}\cdot(L\cdot x) = x$: this is the more specific half of the specification.

$$\begin{aligned}
&\mathcal{B}\cdot(L\cdot\emptyset) = \emptyset\\
\equiv\;&\quad\{\text{ definition of } \mathcal{B} \text{, to introduce a } \mathcal{B}\ \}\\
&\mathcal{B}\cdot(L\cdot\emptyset) = \mathcal{B}\cdot[\,]\\
\Leftarrow\;&\quad\{\text{ Leibniz , to eliminate the } \mathcal{B}\text{'s }\}\\
&L\cdot\emptyset = [\,]\ ,
\end{aligned}$$

and:

$$\begin{array}{ll}
 & \mathcal{B}\cdot(L\cdot\{b\}) \;=\; \{b\} \\
\equiv & \quad \{ \text{ definition of } \mathcal{B} \ \} \\
 & \mathcal{B}\cdot(L\cdot\{b\}) \;=\; \mathcal{B}\cdot[b] \\
\Leftarrow & \quad \{ \text{ Leibniz } \} \\
 & L\cdot\{b\} \;=\; [b] \quad .
\end{array}$$

Next, for any two bags x and y we try to define $L\cdot(x+y)$ in terms of $L\cdot x$ and $L\cdot y$; this is a <u>design decision</u>: although this approach seems sweetly reasonable, there is no compelling reason for it. So, we derive:

$$\begin{array}{ll}
 & \mathcal{B}\cdot(L\cdot(x+y)) \;=\; x+y \\
\equiv & \quad \{ \text{ induction hypothesis, see below } \} \\
 & \mathcal{B}\cdot(L\cdot(x+y)) \;=\; \mathcal{B}\cdot(L\cdot x) + \mathcal{B}\cdot(L\cdot y) \\
\equiv & \quad \{ \text{ specification of } \oplus \text{, see below } \} \\
 & \mathcal{B}\cdot(L\cdot(x+y)) \;=\; \mathcal{B}\cdot(L\cdot x \oplus L\cdot y) \\
\Leftarrow & \quad \{ \text{ Leibniz } \} \\
 & L\cdot(x+y) \;=\; L\cdot x \oplus L\cdot y \quad .
\end{array}$$

The appeal by induction hypothesis to L's specification is only correct, of course, provided that x and y are smaller than $x+y$. The requirement that both x and y be smaller than $x+y$ is equivalent to the requirement that both y and x be nonempty; this requirement can only be met if $x+y$ has at least 2 elements, which is the main reason why the singleton bag must be treated as a special case (as we did above).

Thus we have obtained the following definition for L; in this definition the new operator $\oplus$ occurs.

<u>definition</u>:

$$\begin{array}{lll} L\cdot\emptyset & = & [\,] \\ L\cdot\{b\} & = & [b] \\ L\cdot z & = & \underline{\text{if}}\ \#z \geq 2 \ \rightarrow\ L\cdot x \oplus L\cdot y \\ & & \qquad\qquad |[\ x,y :\ x+y = z\ \wedge\ x \neq \emptyset\ \wedge\ y \neq \emptyset\ ]| \\ & & \underline{\text{fi}} \end{array}$$

□

<u>specification</u>: $\oplus$ has type $List \times List \rightarrow List$, with:

$$(\forall s,t ::\ \mathcal{B}\cdot(s \oplus t)\ =\ \mathcal{B}\cdot s + \mathcal{B}\cdot t\ )$$

□

Before designing definitions for $\oplus$ we consider the remaining proof obligation from L's specification; we do so, of course, for each of the three cases:

$$\begin{array}{ll} & \mathcal{A}\cdot(L\cdot\emptyset) \\ \equiv & \quad \{\ \text{above definition of } L\ \} \\ & \mathcal{A}\cdot[\,] \\ \equiv & \quad \{\ \text{definition of } \mathcal{A}\ \} \\ & true\ , \end{array}$$

and:

$$\begin{array}{ll} & \mathcal{A}\cdot(L\cdot\{b\}) \\ \equiv & \quad \{\ \text{similarly}\ \} \\ & true\ , \end{array}$$

and:

$$\begin{array}{ll} & \mathcal{A}\cdot(L\cdot(x+y)) \\ \equiv & \quad \{\ \text{definition of } L\ \} \end{array}$$

$$\begin{array}{ll} & \mathcal{A}\cdot(L\cdot x \oplus L\cdot y) \\ \Leftarrow & \quad \{ \text{ additional specification of } \oplus \text{ , see below } \} \\ & \mathcal{A}\cdot(L\cdot x) \ \wedge \ \mathcal{A}\cdot(L\cdot y) \\ \equiv & \quad \{ \text{ specification of } L \text{, by induction hypothesis } \} \\ & \text{true} \ . \end{array}$$

So, the above definition for L satisfies its specification provided that $\oplus$ also satisfies:

$$(\forall s,t :: \ \mathcal{A}\cdot(s\oplus t) \ \Leftarrow \ \mathcal{A}\cdot s \wedge \mathcal{A}\cdot t\,)$$

Because $\oplus$ is now applied to ascending lists only, the first part of its specification —on the previous page— may be weakened accordingly. Thus we obtain a new specification for $\oplus$.

<u>specification</u>: for lists s and t list $s\oplus t$ satisfies:

$$\begin{array}{ll} & \mathcal{B}\cdot(s\oplus t) = \mathcal{B}\cdot s + \mathcal{B}\cdot t \quad \wedge \quad \mathcal{A}\cdot(s\oplus t) \\ \Leftarrow & \\ & \mathcal{A}\cdot s \ \wedge \ \mathcal{A}\cdot t \end{array}$$

□

(This specification just states that, within the realm of ascending lists representing bags, $\oplus$ represents $+$.)

Function L maps bags onto their ascending list representations. If the bags themselves are represented by (not necessarily ascending) lists, the process is known as <u>sorting</u>. That is, the function $L\circ\mathcal{B}$ maps a list onto the ascending list representing the same bag. $\mathcal{B}$'s definition is easily "fused" with the above definition of L.

<u>specification</u>: function sort has type List → List , with:

$$\mathrm{sort} = L \circ \mathcal{B}$$

□
<u>definition</u>:

```
sort·u  =  if #u<2  →  u
           ▯ #u≥2  →  sort·s ⊕ sort·t
                       |[ s,t : 𝓑·s + 𝓑·t = 𝓑·u  ∧
                                s ≠ [] ∧ t ≠ []
                       ]|
           fi
```

□
<u>corollary:</u> $(\forall s :: \mathcal{B}\cdot(\mathrm{sort}\cdot s) = \mathcal{B}\cdot s \;\wedge\; \mathcal{A}\cdot(\mathrm{sort}\cdot s))$
□

In order to obtain a fully-fledged sorting algorithm we still must choose appropriate definitions for the operator ⊕ and for the lists s,t as a partitioning of list u. The specification of s,t leaves a considerable amount of freedom; by restricting this freedom we can impose upon s and t additional requirements, which, in turn, may make it easier to design a suitable definition for ⊕. In the next sections we illustrate this by means of 4 variations.

• Merge sort

The most complicated algorithm arises when we impose no additional requirements upon s and t. This leaves a maximal amount of freedom for s and t but gives rise to quite a complicated operator $\oplus$. The resulting algorithm is called merge sort.

definition:

```
msort.u = if #u<2 → u
          [] #u≥2 → msort.s ⊕ msort.t
                   |[ s = u↑n  & t = u↓n
                    & n = #u div 2
                   ]|
          fi
          |[ [] ⊕ t     = t
           & s ⊕ []     = s
           & (b;s) ⊕ (c;t)
                        = if b≤c → b; (s ⊕ (c;t))
                          [] c≤b → c; ((b;s) ⊕ t)
                          fi
          ]|
```

□

(noot: ik denk dat de afleiding van de definitie van $\oplus$ een aparte paragraaf verdient.
)

The choice $s = u{\uparrow}n$ & $t = u{\downarrow}n$ is just one from many possibilities; the choice $n = \#u$ div 2 is, of course, based upon efficiency considerations.

- Quicksort

The requirement $\mathcal{B}.(s \oplus t) = \mathcal{B}.s + \mathcal{B}.t$ in the specification of $\oplus$ and the rule $\mathcal{B}.(s +\!\!+ t) = \mathcal{B}.s + \mathcal{B}.t$ in the definition of $\mathcal{B}$ suggest $+\!\!+$ as a candidate for $\oplus$. If we now substitute $+\!\!+$ for $\oplus$ in the other requirement in $\oplus$'s specification we obtain:

$$\begin{array}{ll}
 & \mathcal{A}.(s +\!\!+ t) \Leftarrow \mathcal{A}.s \wedge \mathcal{A}.t \\
\equiv & \quad \{ \text{ definition of } \mathcal{A} \ \} \\
 & \mathcal{A}.s \wedge \mathcal{A}.t \wedge (\forall b,c : b \in s \wedge c \in t : b \le c) \Leftarrow \mathcal{A}.s \wedge \mathcal{A}.t \\
\Leftarrow & \quad \{ \text{ propositional calculus } \} \\
{}^{*)} & (\forall b,c : b \in s \wedge c \in t : b \le c) \ .
\end{array}$$

So, we may indeed use $+\!\!+$ for $\oplus$, provided we weaken its specification by strengthening its precondition with $(\forall b,c : b \in s \wedge c \in t : b \le c)$.

This has consequences for the partitioning of u into s and t; we have:

$$\begin{array}{ll}
 & b \in \text{sort}.s \\
\equiv & \quad \{ \text{ property of } \in \ \} \\
 & b \in \mathcal{B}.(\text{sort}.s) \\
\equiv & \quad \{ \text{ specification of sort } \} \\
 & b \in \mathcal{B}.s \\
\equiv & \quad \{ \text{ property of } \in \ \} \\
 & b \in s \ .
\end{array}$$

Therefore, we may rewrite the precondition of $\text{sort}.s +\!\!+ \text{sort}.t$ as follows:

---

*) On second thought I would rather abbreviate this to $s \le t$.

$$
\begin{array}{ll}
& (\forall b,c : b \in sort\cdot s \wedge c \in sort\cdot t : b \leq c) \\
\equiv & \quad \{ \text{ above } \} \\
& (\forall b,c : b \in s \wedge b \in t : b \leq c) \ .
\end{array}
$$

The latter condition can be added to the specification of s and t. That is, we need a function part with the following specification.

<u>specification</u>: function part has type $List \rightarrow List \times List$, with:

$$
\begin{array}{l}
(\forall s,t,u :: \langle s,t \rangle = part\cdot u \ \wedge \ \#u \geq 2 \\
\qquad \Rightarrow \\
\qquad\quad \mathcal{B}\cdot s + \mathcal{B}\cdot t = \mathcal{B}\cdot u \ \wedge \ s \neq [] \wedge t \neq [] \ \wedge \\
\qquad\quad (\forall b,c : b \in s \wedge c \in t : b \leq c) \\
)
\end{array}
$$

□

With this function we obtain (a variation of) the famous Quicksort algorithm.

<u>definition</u>:

$$
\begin{array}{lll}
qsort\cdot u & = & \underline{if}\ \#u < 2 \ \rightarrow \ u \\
& & \ \ [] \ \#u \geq 2 \ \rightarrow \ qsort\cdot s \mathbin{+\!\!+} qsort\cdot t \\
& & \qquad\qquad\qquad |[\ \langle s,t \rangle = part\cdot u \ ]| \\
& & \underline{fi}
\end{array}
$$

□

(<u>noot</u>: implementatie van part ook in een aparte paragraaf ?)

- <u>Insertion sort</u>

One of the two recursive applications of sort can be eliminated if we impose the additional requirement $\#s = 1$:

$$\begin{array}{ll}
 & \text{sort}\cdot s \oplus \text{sort}\cdot t \\
= & \quad \{ \ \#s=1 : \text{ definition of sort } \} \\
 & s \oplus \text{sort}\cdot t \\
= & \quad \{ \ \#s=1, \text{ hence } s=[b] \text{ for some element } b \ \} \\
 & [b] \oplus \text{sort}\cdot t \\
= & \quad \{ \text{ introduction of a new operator } \otimes \ \} \\
 & b \otimes \text{sort}\cdot t \ .
\end{array}$$

The operator $\otimes$ must satisfy $b \otimes t = [b] \oplus t$; using the specification of $\oplus$ we thus obtain a "direct" specification of $\otimes$.

<u>specification</u>: for element b and list t :

$$\begin{array}{ll}
 & \mathcal{B}\cdot(b \otimes t) = \{b\} + \mathcal{B}\cdot t \ \wedge \ \mathcal{A}\cdot(b \otimes t) \\
\Leftarrow & \\
 & \mathcal{A}\cdot t
\end{array}$$

$\square$

After a few minor embellishments we obtain the following definition, which is known as <u>insertion sort</u> .

<u>definition</u>

```
isort.[]     = []
isort.(b;t)  = b ⊗ isort.t
               |[ b ⊗ []     = [b]
               & b ⊗ (c;t)   = if b ≤ c → b;c;t
                               [] c ≤ b → c;(b ⊗ t)
                               fi
               ]|
```

□

- <u>Selection sort</u>

By combining the two (independent) requirements $s \leq t$ — cf. the footnote on page 7 — and $\#s = 1$ we obtain our final variation, which is known as <u>selection sort</u>. Notice that the requirement

$$\mathcal{B}.s + \mathcal{B}.t = \mathcal{B}.u \quad \wedge \quad s \leq t \quad \wedge \quad \#s = 1$$

amounts to the requirement that the singleton list s hold the minimum element of u.

As a result we obtain, again with a few embellishments, the following definition. The derivation of a suitable definition for the function select is left as an exercise.

<u>specification</u>:

$$(\forall b,t,u :: \langle b,t\rangle = \text{select}.u \ \wedge \ \#u \geq 1$$
$$\Rightarrow \quad \{b\} + \mathcal{B}.t = \mathcal{B}.u \ \wedge \ b \leq t \quad)$$

□

<u>definition</u> :

```
ssort·[]  =  []
ssort·u   =  if #u ≥ 1  →  b ; ssort·t
                              |[ ⟨b,t⟩ = select·u ]|
             fi
```

□

The 4 sorting algorithms can be arranged in the following diagram. That insertion sort and selection sort can be considered as special cases —degenerate cases if you like— of merge sort and Quicksort was new to me.

$$\begin{tikzcd}
\text{msort} \arrow[r, "s\leq t"] \arrow[d, "\#s=1"] & \text{qsort} \arrow[d, "\#s=1"] \\
\text{isort} \arrow[r, "s\leq t"] & \text{ssort}
\end{tikzcd}$$

Eindhoven, 21 june 1994
Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology