Quicksort for Linked Lists

by

Tom Verhoeff

Computing Science Note 93/03
Eindhoven, January 1993
This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology. Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review. Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

All rights reserved
editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.
Quicksort for Linked Lists

Tom Verhoeff*

January 1993

Abstract

I present and analyze a version of Quicksort for linked lists. On random lists its execution speed compares favorably to that of an efficient list version of Merge sort.

*Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands; E-mail: <vstol@win.tue.nl>.
1 Specification

While working on a small graphics project, I got to a point where linked lists had to be sorted (on “depth” coordinate for the painter’s algorithm [FvDFH90]). Merge sort is very appropriate for sorting lists. Nevertheless, I wondered whether I could also employ Quicksort, since for arrays Quicksort is on the average about 25% faster than Merge sort (see [Meh84, p. 67]). Knuth considers several sorting algorithms for linked lists in [Knu73], but Quicksort is not among them.

Before embarking on a specification of the problem, let me explain my notation for (finite abstract) lists, adapted from [BW88]. The list consisting of the $n$ elements $a_0, \ldots, a_{n-1}$ (in this order) is denoted by $[a_0, \ldots, a_{n-1}]$. In particular, $[]$ denotes the empty list and $[a]$ denotes the list consisting of element $a$ only. The catenation of lists $s$ and $t$ (in this order) is denoted by $s \++ t$. For element $a$ and list $s$, I abbreviate $[a] \++ s$ to $a : s$. An infix dot stands for function application; it binds stronger than catenation. Function $\#$ returns the length of its list argument, for example $\#.(s \++ t) = \#.s + \#.t$. Finally, $\text{sort}$ is a function from lists to lists such that $\text{sort}.s$ consists of the elements of $s$ in ascending order.

\begin{verbatim}

type ElementType = ...;
List = Cell;
Cell = record
  value: ElementType;
  next: List
end { Cell };

Figure 1: Type definitions to implement lists
\end{verbatim}

The Pascal type definitions in Figure 1 form the basis of an implementation of lists over some element type. For convenience’ sake, I assume that $\text{ElementType}$ is ordered by $<$. For lists $s$ and $t$ over $\text{ElementType}$, I write $s < t$ to express that each elements of $s$ precedes all elements of $t$.

The type $\text{List}$ represents the set of lists over $\text{ElementType}$. In order to express the representation invariant and abstraction function, it is convenient to define some auxiliary operators on $\text{List}$ (these are adapted from Lex Bijlsma’s course notes for Programming 3). For integer $n$ ($n \geq 0$), and for $p$ and $q$ of type $\text{List}$, define

\begin{align*}
drop.n.p &= \begin{cases} p & \text{if } n = 0 \lor p = \text{nil} \\
drop.(n-1).(p.next) & \text{if } n > 0 \land p \neq \text{nil} \end{cases} \\
p \rightarrow q &= (\exists n : n \geq 0 : \text{drop}.n.p = q) \\
p \triangleright q &= \begin{cases} [] & \text{if } p = q \\
(p.value) : (p.next \triangleright q) & \text{if } p \neq q \end{cases}
\end{align*}

Predicate $p \rightarrow q$ expresses that $q$ is ‘reachable’ from $p$. If $p \rightarrow q$ holds then $p \triangleright q$ is the list of values in the chain from $p$ to $q$.

The representation invariant $I$ and abstraction function $[]}$ on the type $\text{List}$ are now defined by

\begin{align*}
I.s &= s \rightarrow \text{nil}
\end{align*}
Note the following properties of the abstraction function.

\[ p = \text{nil} \] \quad \iff \quad [p] = []

\[ p = a : s \quad \iff \quad p.value = a \land [p].next = s \]

The specification of my sorting problem can be expressed by

```pascal
function Quicksort(s: List): List;
{ assumes \([s] = S\); returns \(z\), where \([z] = \text{sort}.S\) }
var I, r: List;
begin
if s = \text{nil} then Quicksort := s
else with \(s\) do begin \([s] = \text{value} : [\text{next}]\)
  "partition [next], using value as pivot";
  { \(\text{sort}.S = \text{sort}.([I]++[\text{value}]++[R]) \land [I] < [\text{value}] \leq [R]\) }
  { hence, \(\text{sort}.S = \text{sort}.[I]++[\text{value}]++\text{sort}.[R]\) }
  \(\text{next} := \text{Quicksort}(r)\); \{ \(\text{sort}.S = \text{sort}.[I]++[s]\) \}
  Quicksort := "catenation of Quicksort(l) and s"
end \{ else with \(s\) \}
end \{ Quicksort \};
```

Figure 2: “Standard” design for Quicksort in Pascal

In a functional style (see [BW88, p. 154]), Quicksort can be defined by

\[
\text{Quicksort}.[] = []
\]

\[
\text{Quicksort}.(a : u) = \text{Quicksort}.l ++ [a] ++ \text{Quicksort}.r
\]

where \(l = [x|x \leftarrow u; x < a]\)

\(r = [x|x \leftarrow u; a \leq x]\)

Here, \([x|x \leftarrow u; x < a]\) is the list of elements \(x\) drawn from \(u\) that are less than \(a\) (see [BW88, p. 50]). In functional terms, my goal is to eliminate ++ in favor of the more efficient : operator.

The solution is based on the observation that Quicksort satisfies

\[
\text{Quicksort}.s = \text{Quicksort}.s ++ []
\]
and, therefore, its specification can be generalized to \( Qsort \) by defining

\[
Qsort.s.t = Quicksort.s \, ++ \, t
\]

Specialization of this definition yields \( Quicksort.s = Qsort.s[\cdot] \). For \( Qsort \), I derive

\[
Quicksort[\cdot].t = Quicksort.[\cdot] \, ++ \, t = [] \, ++ \, t = t
\]

and, assuming \( l = [x|x \leftarrow u; x < a] \) and \( r = [x|x \leftarrow u; a \leq x] \),

\[
\begin{align*}
Qsort.(a : u).t &= Quicksort.(a : u) \, ++ \, t \\
&= Quicksort.l \, ++ \, [a] \, ++ \, Quicksort.r \, ++ \, t \\
&= Quicksort.l \, ++ \, (a : Quicksort.r \, ++ \, t) \\
&= Qsort.l.(a : Qsort.r.t)
\end{align*}
\]

Thus, a functional definition for \( Qsort \) is

\[
\begin{align*}
Qsort.[\cdot].t &= t \\
Qsort.(a : u).t &= Qsort.l.(a : Qsort.r.t) \\
\text{where } l &= [x|x \leftarrow u; x < a] \\
r &= [x|x \leftarrow u; a \leq x]
\end{align*}
\]

Figure 3 gives the corresponding Pascal implementation of \( Qsort \). I have coded the partition phase explicitly.

\[
\text{function } Qsort(s, t: List): List; \\
\text{\{ assumes } [s] = S \land [t] = T \text{ \}; returns } z, \text{ where } [z] = sort.S \, ++ \, T \text{ \}} \\
\text{var } a:ElementType; u, l, r, h: List; \\
\text{begin} \\
\text{if } s = \text{nil} \text{ then } Qsort := t \\
\text{else with } s \uparrow \text{ do begin} \\
\text{ \quad } a := \text{value} ; u := \text{next} \; ; \; \{ [s] = a : [u] \} \\
\text{ \quad } l := \text{nil} ; r := \text{nil} ; \\
\text{ \quad } \{ \text{invariant: } sort.S = sort.([l] \, ++ \, [a] \, ++ \, [r] \, ++ \, [u]) \land [l] < [a] \leq [r] \} \\
\text{ \quad } \text{while } u \neq \text{nil} \text{ do with } u \uparrow \text{ do \{} \\
\text{ \quad \quad } \{ [u] = \text{value} : [\text{next}] \} \\
\text{ \quad \quad } \text{if } value < a \text{ then } \{ "l, } u := \text{value} : l, \text{next" } \} \\
\text{ \quad \quad } \text{begin } h := \text{next} ; \text{next} := l ; l := u ; u := h \text{ end} \\
\text{ \quad \quad } \text{else } \{ "r, } u := \text{value} : r, \text{next" } \} \\
\text{ \quad \quad } \text{begin } h := \text{next} ; \text{next} := r ; r := u ; u := h \text{ end} ; \\
\text{ \quad } \{ sort.S = sort.([l] \, ++ \, [a] \, ++ \, sort.[r] \} \\
\text{ \quad } \text{next} := Qsort(r,t) ; \{ sort.S = sort.([l] \, ++ \, [s] \} \\
\text{ \quad } Qsort := Qsort(l,s) \\
\text{ \text{end } \{} \text{ else with } s \uparrow \text{ } \} \\
\text{\text{end } \{ Qsort \} ; \\
\end{align*}
\]

Figure 3: Pascal implementation of \( Qsort \)

The Pascal implementation of \( Quicksort \) now boils down to
function Quicksort(s: List): List;
{ assumes \([s] = S\) ; returns \([z] = sort.S\) }
begin Quicksort := Qsort(s, nil) end;

Of course, this list version of Quicksort still has quadratic worst-case time-complexity. Furthermore, it is not stable (in the sense that records with equal values retain their original relative ordering, see [Knu73, p. 4]), because each partitioning reverses the order (the lists are manipulated as stacks). Quicksort for arrays is, in general, also not stable. Stability can be guaranteed, but at the price of complicating the partitioning phase (also see [Mot81, Weg82], where “end” pointers are used). In Section 4, I suggest another version of Quicksort for linked lists. That version is stable and also allows limitation of the maximum recursion depth. The latter cannot easily be accomplished in the Pascal version of \(Qsort\), unless the compiler implements tail recursion efficiently or allows pointers to static variables. A final remark about \(Qsort\) is that the head of the list is used as partitioning pivot and that taking another element (for instance, a random element or the median of three random elements as is often done for improvement) would be costly, since lists cannot be indexed efficiently.

3 Efficiency

It is not guaranteed that the efficiency of the above list version of Quicksort is comparable to that of the array version. One has to be very careful to extrapolate performance figures (see [Sed78]). Therefore, it is interesting to compare \(Qsort\) to a (good) list version of Merge sort. In case of the array version, Quicksort is on the average 25% faster than Merge sort, see [Meh84, p. 67]. For the list versions their execution speed on random lists turns out to be comparable.

function Mergesort(s: List): List;
{ assumes \([s] = S\) ; returns \([z] = sort.S\) }
var t, u: List;
begins
if \((s = \text{nil}) \lor (s\.next = \text{nil})\) then Mergesort := s
else begin \{ \#.S \geq 2 \}
  u := s ; t := u\.next\.next ;
  \{ invariant: \#.\((s \triangleright u\.next) = \#.\((u\.next \triangleright t)\) \}
  while \((t \neq \text{nil}) \land \text{and} \((t\.next \neq \text{nil})\) do
    begin u := u\.next ; t := t\.next\.next end ;
  t := u\.next ; u\.next := \text{nil} ;
  \{ S = [u] ++ [t] \land 0 \leq \#.\([t] - \#.\([u]\} \leq 1, \text{ hence} \([s]\} \neq [\] \land [t] \neq [\] \}
  Mergesort := Merge(Mergesort(s), Mergesort(t))
end \{ else \}
end \{ Mergesort \} ;

Figure 4: Pascal implementation of Merge sort

I consider the Pascal function for Merge sort given in Figure 4. It first splits the list into two halves by cutting the list roughly in the middle. A common alternative is to “unzip” the
list into even and odd elements to obtain the two halves, but this turns out to be slightly more expensive and stability is lost. The two halves are then sorted recursively and finally merged to yield the result. The Boolean operators cor and cand stand for conditional or and respectively (also known as short-circuit boolean operators).

function Merge(s, t: List): List;
{ assumes \( \exists q : q \neq nil : s \rightarrow q \land t \rightarrow q \) \& \( [s] = S \land [t] = T \) \& \( S \neq [] \land T \neq [] \) \& sort.S = S \land sort.T = T \) \}
{ returns z, where \([z] = \text{sort.}(S ++ T)\) }
begin
  if s\[.value\] \leq t\[.value\] then with s\[do\] { “Merge := value : Merge(next, t)” }
    if next = nil then begin next := t ; Merge := s end
  else begin next := Merge(next, t) ; Merge := s end
  else with t\[do\] begin { “Merge := value : Merge(s, next)” }
    if next = nil then begin next := s ; Merge := t end
  else begin next := Merge(s, next) ; Merge := t end
end { Merge };

Figure 5: Pascal implementation of Merge

The function Merge is defined in Figure 5. The precondition of Merge expresses that its arguments should be ‘independent’ non-empty sorted lists. This is the case for the invocation in Mergesort above, but invoking Merge(s, s) would get into one kind of trouble or another. Of course, the invocation of Merge can be unfolded into a loop to obtain a more efficient program (with respect to both speed and memory usage). I leave this to the reader. By the way, the above combination of Mergesort and Merge forms a stable sorting algorithm.

I have compared the execution times on random lists for Quicksort (based on Qsort, with singleton lists done directly instead of by partitioning), Merge sort (based on Mergesort, with test \( s = nil \) eliminated (because the random lists are non-empty), and unfolded Merge), and a list version of insertion sort. The results are presented in Table 1. The execution times are relative to those for applying the identity function. All four functions were applied to the same data, by resetting the seed of the random generator before each test run.

<table>
<thead>
<tr>
<th>List length</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10,000</th>
<th>100,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeat count</td>
<td>10,000</td>
<td>1000</td>
<td>100</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>Quicksort</td>
<td>33</td>
<td>51</td>
<td>70</td>
<td>91</td>
<td>114</td>
</tr>
<tr>
<td>Merge sort</td>
<td>42</td>
<td>63</td>
<td>82</td>
<td>102</td>
<td>121</td>
</tr>
<tr>
<td>Insertion sort</td>
<td>11</td>
<td>86</td>
<td>846</td>
<td>8459</td>
<td>not run</td>
</tr>
</tbody>
</table>

Table 1: Comparison of execution times (seconds)

The advantage of Quicksort is that it contains just one loop (for partitioning), whereas Mergesort contains two loops (one for splitting and one for merging). The disadvantage of Quicksort is that partitioning may yield two lists whose lengths differ in order of magnitude, whereas Mergesort always works with a perfectly ‘balanced’ split. For random lists, the disadvantage is apparently less important than the advantage of a single loop. The break-even point for Quicksort and insertion sort occurs at lists of about fifty elements. For the
array versions, the break-even point usually occurs at a much shorter length (see [Sed78]), because array insertion involves data movement in contrast to list insertion.

4 Dummy Head Cells

The implementation of lists by the type List has a small disadvantage, viz. that all elements except possibly the first (if present) are pointed to from a next-field. This may give rise to a case distinction in programs dealing with such lists (this happens, for example, when maintaining stability in Qsort or unfolding Merge into a loop). The exceptional case can be avoided by prefixing each list with a dummy head cell. This is expressed in the following type definition, and corresponding representation invariant and abstraction function.

\[
HeadedList = List
\]

\[
I.s = s \neq \text{nil} \land s \rightarrow \text{nil}
\]

\[
[s] = s^\dagger.next \rightarrow \text{nil}
\]

Each element in a list of type HeadedList is pointed to from a next-field. In particular, the first element is pointed to by the next-field of the dummy head cell.

\[
\text{procedure } QsortH(s,t: HeadedList);
\]

\[
\{ \text{assumes } [s] = S \land [t] = T \land s^\dagger.value = A \}
\]

\[
\{ \text{establishes } [s] = \text{sort}.S ++ T \land s^\dagger.value = A \}
\]

\[
\text{var } a: ElementType; r: HeadedList; sl, rl, u: List; m, n: Integer;
\]

\[
\begin{align*}
\text{begin} & \quad \text{while } s^\dagger.next \neq \text{nil} \text{ do begin} \\
& \quad \quad r := s^\dagger.next; u := r^\dagger.next; a := r^\dagger.value; \{ [s] = a : [u] \} \\
& \quad \quad sl := s; rl := r; \{ \text{abbreviate } L = s^\dagger.next \rightarrow sl^\dagger.next \text{ and } R = r^\dagger.next \rightarrow rl^\dagger.next \} \\
& \quad \quad m := 0; n := 0; \{ m = \#.L \text{ and } n = \#.R \} \\
& \quad \quad \{ \text{invariant: } \text{sort}.S = \text{sort}.(L ++ [a] ++ R ++ [u]) \land L < [a] \leq R \} \\
& \quad \quad \quad \text{while } u \neq \text{nil} \text{ do with } u \uparrow \text{ do } \{ [u] = \text{value} : [next] \} \\
& \quad \quad \quad \quad \text{if } \text{value} < a \text{ then } \{ "L, u := L ++ [value], next" \} \\
& \quad \quad \quad \quad \quad \text{begin } sl^\dagger.next := u; sl := u; m := m + 1; u := \text{next end} \\
& \quad \quad \quad \quad \text{else } \{ "R, u := R ++ [value], next" \} \\
& \quad \quad \quad \quad \quad \text{begin } rl^\dagger.next := u; rl := u; n := n + 1; u := \text{next end}; \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \{ \text{sort}.S = \text{sort}.[s] ++ [a] ++ \text{sort}.[r] \} \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if } m < n \text{ then begin } QsortH(s,r); s := r \text{ end} \\
& \quad \quad \quad \quad \text{else begin } QsortH(r,t); t := r \text{ end} \\
& \quad \quad \quad \text{end } \{ \text{while} \} ; \\
& \quad \quad s^\dagger.next := t \\
& \quad \text{end } \{ Qsort \} ; \\
\end{align*}
\]

Figure 6: Version of Qsort in terms of HeadedList

The version of Qsort in Figure 6 is designed in terms of HeadedList. Partitioning does not reverse the order and, hence, Qsort\(H\) is stable. Its maximum recursion depth has been
limited by recursing only on the shorter of the two lists obtained by partitioning. For that purpose, counters \( m \) and \( n \) have been introduced. Observe that the (dummy) head cell of \( s \) is used as head cell of the “left” partitioning result, and the cell containing the pivot is used as head cell of the “right” partitioning result. Thus, no additional cells are needed.

Table 2 presents some timing results on random headed lists, again relative to applying the identity transformation. Entry “Ltd. Quicksort” is based on \( QsonH \) with singleton lists done directly. Entry “Quicksort” is like “Ltd. Quicksort” but without limiting the maximum recursion depth. Entry “Merge sort” is adapted from Merge sort in the preceding section. Note that each iteration through the partitioning loop of \( QSortH \) consists of two comparisons and four assignments, the same as in \( QSort \). This explains why “Ltd. Quicksort” for headed lists performs almost the same as Quick sort in Table 1, and also why Quick sort in Table 2 performs better (it has one assignment fewer in its partitioning loop). Merge sort in Table 2 performs less well that in Table 1, because only the (dummy) head cell of its argument is available (no pivot), and thus the second half (of the split) must be attached to that same head cell before recursing on it.

### 5 Concluding Remarks

I have presented a list version of Quick sort for linked lists (without an “end” pointer to the last element). The Pascal program \( Qsort \) is very short. On random lists it performs well compared to a list version of Merge sort. There are however a few side remarks to be made.

First of all, \( Qsort \) is not stable. On the other hand, most versions of Quick sort and some versions of Merge sort are also not stable. Secondly, it is harder to implement a version that does not use the head of the list as pivot (but, for instance, a random element). In Pascal it is also harder to limit the maximum recursion depth by putting the “longest” recursive call at the end and transforming it into a loop. However, when using lists with a dummy head cell, it is easy to guarantee stability and to limit the maximum recursion depth.

In the same vein, a list version of Quick sort is feasible that operates in constant-space (i.e. without recursion overhead) [Ver93].

I would like to thank Johan Lukkien and the Eindhoven Algorithm Club lead by Anne Kaldewaij for reading and commenting on an earlier version of this note.

### References


In this series appeared:

91/01 D. Alstein

D. Alstein

91/01 Dynamic Reconfiguration in Distributed Hard Real-Time

91/02 R.P. Nederpelt
H.C.M. de Swart

91/02 Implication. A survey of the different logical analyses
"if..., then...", p. 26.

91/03 J.P. Katoen
L.A.M. Schoenmakers

Parallel Programs for the Recognition of P-invariant
Segments, p. 16.

91/04 E. v.d. Sluis
A.F. v.d. Stappen

Performance Analysis of VLSI Programs, p. 31.

91/05 D. de Reus

An Implementation Model for GOOD, p. 18.

91/06 K.M. van Hee

SPECIFICATIEMETHODEN, een overzicht, p. 20.

91/07 E.Poll

CPO-models for second order lambda calculus with
recursive types and subtyping, p. 49.

91/08 H. Schepers

Terminology and Paradigms for Fault Tolerance, p. 25.

91/09 W.M.P.v.d.Aalst

Interval Timed Petri Nets and their analysis, p.53.

91/10 R.C. Backhouse
P.J. de Bruin
P. Hoogendijk
G. Malcolm
E. Voermans
J. v.d. Woude

POLYNOMIAL RELATORS, p. 52.

91/10 Relational Catamorphism, p. 31.

91/11 R.C. Backhouse
P.J. de Bruin
G. Malcolm
E. Voermans
J. van der Woude

A parallel local search algorithm for the travelling
salesman problem, p. 12.

91/12 E. van der Sluis

A note on Extensionality, p. 21.

91/13 F. Rietman

The PDB Hypermedia Package. Why and how it was
built, p. 63.

91/14 P. Lemmens

Eldorado: Architecture of a Functional Database

91/15 A.T.M. Aerts
K.M. van Hee

An example of proving attribute grammars correct:
the representation of arithmetical expressions by DAGs,
p. 25.

91/16 A.J.J.M. Marcelis

Transforming Functional Database Schemes to Relational
Representations, p. 21.

91/17 A.T.M. Aerts
P.M.E. de Bra
K.M. van Hee
<table>
<thead>
<tr>
<th>Paper Number</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>91/18</td>
<td>Rik van Geldrop</td>
<td>Transformational Query Solving, p. 35.</td>
</tr>
<tr>
<td>91/19</td>
<td>Erik Poll</td>
<td>Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.</td>
</tr>
<tr>
<td>91/23</td>
<td>K.M. van Hee, L.J. Somers, M. Voorhoeve</td>
<td>Z and high level Petri nets, p. 16.</td>
</tr>
<tr>
<td>91/24</td>
<td>A.T.M. Aerts, D. de Reus</td>
<td>Formal semantics for BRM with examples, p. 25.</td>
</tr>
<tr>
<td>91/25</td>
<td>P. Zhou, J. Hooman, R. Kuiper</td>
<td>A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and completeness, p. 52.</td>
</tr>
<tr>
<td>91/27</td>
<td>F. de Boer, C. Palamidessi</td>
<td>Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.</td>
</tr>
<tr>
<td>91/28</td>
<td>F. de Boer</td>
<td>A compositional proof system for dynamic process creation, p. 24.</td>
</tr>
<tr>
<td>91/30</td>
<td>J.C.M. Baeten, F.W. Vaandrager</td>
<td>An Algebra for Process Creation, p. 29.</td>
</tr>
<tr>
<td>91/31</td>
<td>H. ten Eikelder</td>
<td>Some algorithms to decide the equivalence of recursive types, p. 26.</td>
</tr>
<tr>
<td>91/33</td>
<td>W. v.d. Aalst</td>
<td>The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.</td>
</tr>
<tr>
<td>91/34</td>
<td>J. Coenen</td>
<td>Specifying fault tolerant programs in deontic logic, p. 15.</td>
</tr>
</tbody>
</table>
A note on compositional refinement, p. 27.

A compositional semantics for fault tolerant real-time systems, p. 18.

Real space process algebra, p. 42.

Program derivation in acyclic graphs and related problems, p. 90.

Conservative fixpoint functions on a graph, p. 25.

Discrete time process algebra, p.45.

The fine-structure of lambda calculus, p. 110.

On stepwise explicit substitution, p. 30.


Composition and decomposition in a CPN model, p. 55.

Demonic operators and monotype factors, p. 29.


Set theory and nominalisation, Part II, p.22.

The total order assumption, p. 10.

A system at the cross-roads of functional and logic programming, p.36.

Integrity checking in deductive databases; an exposition, p.32.

Interval timed coloured Petri nets and their analysis, p. 20.

A unified approach to Type Theory through a refined lambda-calculus, p. 30.

Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.

Are Types for Natural Language? P. 32.

Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.
<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>92/22</td>
<td>R. Nederpelt, F.Kamareddine</td>
<td>A useful lambda notation, p. 17.</td>
</tr>
<tr>
<td>92/23</td>
<td>F.Kamareddine, E.Klein</td>
<td>Nominalization, Predication and Type Containment, p. 40.</td>
</tr>
<tr>
<td>92/24</td>
<td>M.Codish, D.Dams, Eyal Yardeni</td>
<td>Bottom-up Abstract Interpretation of Logic Programs, p. 33.</td>
</tr>
<tr>
<td>92/25</td>
<td>E.Poll</td>
<td>A Programming Logic for F0, p. 15.</td>
</tr>
<tr>
<td>93/01</td>
<td>R. van Geldrop</td>
<td>Deriving the Aho-Corasick algorithms: a case study into the synergy of programming methods, p. 36.</td>
</tr>
<tr>
<td>93/02</td>
<td>T. Verhoeff</td>
<td>A continuous version of the Prisoner's Dilemma, p. 17</td>
</tr>
<tr>
<td>93/03</td>
<td>T. Verhoeff</td>
<td>Quicksort for linked lists, p. 8.</td>
</tr>
<tr>
<td>93/04</td>
<td>E.H.L. Aarts, J.H.M. Korst, P.J. Zwiering</td>
<td>Deterministic and randomized local search, p. 78.</td>
</tr>
</tbody>
</table>