C. DETAILED ANALYSIS RESULTS

An overview of the results of the binding-time analysis, given binding-time patterns for each goal function, are shown in Tables I–II. By manually looking at each program, we have listed the optimal analysis result, i.e. the fewest generalisations and specialisation point insertions necessary to guarantee termination of specialisation, and whether the prototype is able to achieve this or is more conservative.

Even though the sorting functions have been rewritten as previously described, the analyser is not able to detect that they terminate. This is because during the reordering of the list, our rather crude size approximations lose track of the list sizes. One could patch on this problem by passing around a measure of the list lengths (and decreasing them whenever the lists got shorter), but that would not be a natural way to write the sorting functions.

The function for rewriting an expression with an associative operator, \texttt{assocrw}, cannot be proven by the analyser to terminate. This should come as no surprise, as it requires an advanced size measure not only keeping track of the number of cons nodes but also the structure of the syntax tree.

The example program \texttt{nestimeql} shows one shortcoming of the size approximation function \(E\): for the function

\[
\text{immatcopy } x = \begin{cases} 
\text{if } x = [] & \text{then } [] \\
\text{else cons (car x) (immatcopy (cdr x))}
\end{cases}
\]

our size approximation cannot detect that the size of the return value of a call \texttt{immatcopy x} is the same as the size of \(x\), resulting in conservative generalisation. Similar problems occur with a \texttt{revapp} call in \texttt{permute}, and a \texttt{reverse} call in \texttt{shuffle}.

The remaining examples are handled without resulting in overconservative re-
### Program Description

<table>
<thead>
<tr>
<th>Program</th>
<th>Goal</th>
<th>BT</th>
<th>Result</th>
<th>Optimal result</th>
<th>Conservative</th>
</tr>
</thead>
<tbody>
<tr>
<td>int-loop</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>SP</td>
<td>SP</td>
</tr>
<tr>
<td>int-while-d</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>G</td>
<td>SP</td>
</tr>
<tr>
<td>int-while-s</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>no</td>
<td>SP</td>
</tr>
<tr>
<td>lambdaint</td>
<td>s</td>
<td>SP</td>
<td>SP</td>
<td>G</td>
<td>SP</td>
</tr>
<tr>
<td>parsexp</td>
<td>s</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>turing</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>SP</td>
<td>no</td>
</tr>
<tr>
<td>lambdaint</td>
<td>s</td>
<td>SP</td>
<td>SP</td>
<td>no</td>
<td>SP</td>
</tr>
<tr>
<td>binom</td>
<td>s</td>
<td>d</td>
<td>d</td>
<td>SP</td>
<td>SP</td>
</tr>
<tr>
<td>gcd-1</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>no</td>
<td>SP</td>
</tr>
<tr>
<td>gcd-2</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>no</td>
<td>SP</td>
</tr>
<tr>
<td>graphcol-1</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>SP</td>
</tr>
<tr>
<td>graphcol-2</td>
<td>d</td>
<td>s</td>
<td>SP</td>
<td>SP</td>
<td>no</td>
</tr>
<tr>
<td>graphcol-3</td>
<td>d</td>
<td>s</td>
<td>SP</td>
<td>SP</td>
<td>no</td>
</tr>
<tr>
<td>match</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>SP</td>
<td>no</td>
</tr>
<tr>
<td>power</td>
<td>d</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>no</td>
</tr>
<tr>
<td>reach</td>
<td>d</td>
<td>d</td>
<td>s</td>
<td>SP</td>
<td>SP</td>
</tr>
<tr>
<td>rematch</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>SP</td>
<td>no</td>
</tr>
<tr>
<td>strmatch</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>SP</td>
<td>no</td>
</tr>
<tr>
<td>typeinf</td>
<td>s</td>
<td>s</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>add</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>SP</td>
<td>no</td>
</tr>
<tr>
<td>addlists</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>anchored</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>append</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>assocwr</td>
<td>s</td>
<td>SP</td>
<td>SP</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>badd</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>SP</td>
<td>no</td>
</tr>
<tr>
<td>contrived-1</td>
<td>s</td>
<td>d</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>contrived-2</td>
<td>s</td>
<td>d</td>
<td>SP</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>decrease</td>
<td>s</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>deeprev</td>
<td>s</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>disjconj</td>
<td>s</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>duplicate</td>
<td>s</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>equal</td>
<td>s</td>
<td>SP</td>
<td>SP</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>evenodd</td>
<td>s</td>
<td>SP</td>
<td>SP</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>exponential</td>
<td>s</td>
<td>s</td>
<td>SP</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

**s = static, d = dynamic,**

**SP = insert specialisation point(s), G = generalise variable(s) to ensure termination**

### Table I. Results of binding-time analysis, part I.

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
<table>
<thead>
<tr>
<th>Program</th>
<th>Goal</th>
<th>Result</th>
<th>Optimal result</th>
<th>Program description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fold</td>
<td>d s</td>
<td>SP</td>
<td>no</td>
<td>fold a fixed operator over a list: fold(elt, list)</td>
</tr>
<tr>
<td>game</td>
<td>s s</td>
<td>SP, G</td>
<td>no</td>
<td>play a small game with two players</td>
</tr>
<tr>
<td>increase</td>
<td>s</td>
<td>SP, G</td>
<td>no</td>
<td>no recursive calls with increasing argument</td>
</tr>
<tr>
<td>intlookup</td>
<td>d s</td>
<td>SP</td>
<td>no</td>
<td>no look up variable values like in an interpreter</td>
</tr>
<tr>
<td>letexp</td>
<td>s s</td>
<td>SP, G</td>
<td>no</td>
<td>no example using the let construction</td>
</tr>
<tr>
<td>list</td>
<td>s</td>
<td>SP</td>
<td>no</td>
<td>no check for a list data structure</td>
</tr>
<tr>
<td>lte</td>
<td>d s</td>
<td>SP</td>
<td>no</td>
<td>check whether x is a substructure of y</td>
</tr>
<tr>
<td>map</td>
<td>s</td>
<td>SP</td>
<td>no</td>
<td>no map a fixed function along a list</td>
</tr>
<tr>
<td>member</td>
<td>d s</td>
<td>SP</td>
<td>no</td>
<td>check for list membership: member(element, list)</td>
</tr>
<tr>
<td>mergelists</td>
<td>s d</td>
<td>SP</td>
<td>no</td>
<td>merge two sorted lists</td>
</tr>
<tr>
<td>nil</td>
<td>s d</td>
<td>SP</td>
<td>no</td>
<td>no multiply two unary numbers</td>
</tr>
<tr>
<td>naiverew</td>
<td>s</td>
<td>SP</td>
<td>no</td>
<td>no naive reverse: append reversed tail to head element</td>
</tr>
<tr>
<td>nestdec</td>
<td>s</td>
<td>SP</td>
<td>no</td>
<td>no nested call returns cdr of its argument</td>
</tr>
<tr>
<td>nesteq</td>
<td>s</td>
<td>SP, G</td>
<td>no</td>
<td>no nested call returns a copy of its argument</td>
</tr>
<tr>
<td>nestmeq</td>
<td>s</td>
<td>SP, G</td>
<td>yes</td>
<td>yes nested call to a function that constructs a copy of its argument only from constants</td>
</tr>
<tr>
<td>nestinc</td>
<td>s</td>
<td>SP, G</td>
<td>no</td>
<td>no nested call returns a cons cell containing its argument</td>
</tr>
<tr>
<td>noloxid</td>
<td>s</td>
<td>SP, G</td>
<td>no</td>
<td>no nested call returns a cons cell containing its argument</td>
</tr>
<tr>
<td>ordered</td>
<td>s</td>
<td>SP</td>
<td>no</td>
<td>no check whether a list is ordered</td>
</tr>
<tr>
<td>overlap</td>
<td>s d</td>
<td>SP</td>
<td>no</td>
<td>no check for non-empty set intersection</td>
</tr>
<tr>
<td>permute</td>
<td>s</td>
<td>SP, G</td>
<td>yes</td>
<td>yes compute all the permutations of a list</td>
</tr>
<tr>
<td>revapp</td>
<td>s d</td>
<td>SP</td>
<td>no</td>
<td>no reverse list and append to list: revapp(list1, list2)</td>
</tr>
<tr>
<td>select</td>
<td>s</td>
<td>SP</td>
<td>no</td>
<td>no pick out an element and cons it onto the remaining list</td>
</tr>
<tr>
<td>shuffle</td>
<td>s</td>
<td>SP, G</td>
<td>yes</td>
<td>yes shuffle a list</td>
</tr>
<tr>
<td>spl</td>
<td>s d</td>
<td>SP</td>
<td>no</td>
<td>no mutual recursion requiring specialisation points</td>
</tr>
<tr>
<td>subsets</td>
<td>s</td>
<td>SP</td>
<td>no</td>
<td>no compute all subsets of a set</td>
</tr>
<tr>
<td>thetrick</td>
<td>s d</td>
<td>SP</td>
<td>no</td>
<td>no example using the trick for dynamic if conditionals</td>
</tr>
<tr>
<td>vangelder</td>
<td>s d</td>
<td>SP</td>
<td>no</td>
<td>quasi-terminating example invented by Van Gelder</td>
</tr>
<tr>
<td>merge0r</td>
<td>s</td>
<td>SP, G</td>
<td>yes</td>
<td>yes sort list by splitting, recursive sorting, and merging</td>
</tr>
<tr>
<td>minsort</td>
<td>s</td>
<td>SP, G</td>
<td>yes</td>
<td>yes sort list: extract min elt, cons it onto the sorted rest</td>
</tr>
<tr>
<td>quicksort</td>
<td>s</td>
<td>SP, G</td>
<td>yes</td>
<td>yes sort list by splitting by size, sorting and appending</td>
</tr>
</tbody>
</table>

s = static, d = dynamic, SP = insert specialisation point(s), G = generalise variable(s) to ensure termination

<table>
<thead>
<tr>
<th>Program description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fold a fixed operator over a list: fold(elt, list)</td>
</tr>
<tr>
<td>play a small game with two players</td>
</tr>
<tr>
<td>no recursive calls with increasing argument</td>
</tr>
<tr>
<td>no look up variable values like in an interpreter</td>
</tr>
<tr>
<td>no example using the let construction</td>
</tr>
<tr>
<td>no check for a list data structure</td>
</tr>
<tr>
<td>check whether x is a substructure of y</td>
</tr>
<tr>
<td>no map a fixed function along a list</td>
</tr>
<tr>
<td>check for list membership: member(element, list)</td>
</tr>
<tr>
<td>merge two sorted lists</td>
</tr>
<tr>
<td>no multiply two unary numbers</td>
</tr>
<tr>
<td>no naive reverse: append reversed tail to head element</td>
</tr>
<tr>
<td>no nested call returns cdr of its argument</td>
</tr>
<tr>
<td>no nested call returns a copy of its argument</td>
</tr>
<tr>
<td>yes nested call to a function that constructs a copy of its argument only from constants</td>
</tr>
<tr>
<td>no nested call returns a cons cell containing its argument</td>
</tr>
<tr>
<td>no a terminating function with no lexicographical</td>
</tr>
<tr>
<td>no ordering, cf. Example 7.2.1</td>
</tr>
<tr>
<td>no check whether a list is ordered</td>
</tr>
<tr>
<td>no check for non-empty set intersection</td>
</tr>
<tr>
<td>yes compute all the permutations of a list</td>
</tr>
<tr>
<td>no reverse list and append to list: revapp(list1, list2)</td>
</tr>
<tr>
<td>no pick out an element and cons it onto the remaining list</td>
</tr>
<tr>
<td>yes shuffle a list</td>
</tr>
<tr>
<td>no mutual recursion requiring specialisation points</td>
</tr>
<tr>
<td>no compute all subsets of a set</td>
</tr>
<tr>
<td>no example using the trick for dynamic if conditionals</td>
</tr>
<tr>
<td>quasi-terminating example invented by Van Gelder</td>
</tr>
<tr>
<td>yes sort list by splitting, recursive sorting, and merging</td>
</tr>
<tr>
<td>yes sort list: extract min elt, cons it onto the sorted rest</td>
</tr>
<tr>
<td>yes sort list by splitting by size, sorting and appending</td>
</tr>
</tbody>
</table>

Table II. Results of binding-time analysis, part II.

The programs and results are given in detail in the following sections. The binding times are given by

- **B**: Static and of bounded variation
- **S**: Static but possibly of unbounded variation
- **D**: Dynamic

and are shown before propagating the effects of specialization points. In some cases this would change more variables from B or S to D.
C.1 Interpreters

C.1.1 \textit{int-loop}

\begin{verbatim}
;; Small 1st order interpreter for LOOP programs
(define (run p l input)
  (let* ((f0 (car (car p)))
         (ef (lookbody f0 p))
         (nf (lookname f0 p)))
    (eeval ef (cons nf '()) (cons input '()) l p)))

(define (eeval e ns vs l p)
  (if (equal? (car e) 1) ; constants
      (cdr e)
      (if (equal? (car e) 2) ; variable
          (lookvar (cdr e) ns vs)
          (if (equal? (car e) 3) ; basefcn
              (let* ((v1 (eeval (car (cdr (cdr e))) ns vs l p))
                      (v2 (eeval (car (cdr (cdr (cdr e))) ns vs l p))))
                (apply (car (cdr e)) v1 v2))
              (if (equal? (car e) 4) ; if
                  (if (equal? (eeval (car (cdr e)) ns vs l p) 'T)
                      (eeval (car (cdr (cdr e))) ns vs l p)
                      (eeval (car (cdr (cdr (cdr e))) ns vs l p)))
              (if (equal? (car e) 5) ; ==
                  (if (equal? (eeval (car (cdr e)) ns vs l p)
                          (eeval (car (cdr (cdr e))) ns vs l p))
                      'T
                      'F))
          (let* ((ef (lookbody (car (cdr e)) p))
                  (nf (lookname (car (cdr e)) p))
                  (v (eeval (car (cdr (cdr e))) ns vs l p)))
              (if (equal? l '()) '()
                  (eeval ef (cons nf '()) (cons v '()) (cdr l) p)))))

(define (lookvar x ns vs)
  (if (equal? x (car ns)) (car vs) (lookvar x (cdr ns) (cdr vs))))

(define (lookbody f p)
  (if (equal? (car (car p)) f)
      (car (cdr (cdr (car p)))))
  (lookbody f (cdr p)))

(define (lookname f p)
  (if (equal? (car (car p)) f)
      (car (cdr (car p)))
      (lookname f (cdr p)))))

(define (apply op v1 v2)
  (if (equal? op 5) ; equal
      (if (equal? v1 v2) 'T 'F)
      (cons v1 v2)))
\end{verbatim}

Parameter binding times:
- apply: \(op : B\) \(v1 : D\) \(v2 : D\)
- lookname: \(f : B\) \(p : B\)
- lookbody: \(f : B\) \(p : B\)
- lookvar: \(x : B\) \(ns : B\) \(vs : D\)
- eeaval: \(e : B\) \(ns : B\) \(vs : D\) \(1 : B\) \(p : B\)
- run: \(p : B\) \(l : B\) \(input : D\)

Specialisation points:
None.

Parameter binding times:
- apply: \(op : B\) \(v1 : D\) \(v2 : D\)
- lookname: \(f : B\) \(p : B\)
- lookbody: \(f : B\) \(p : B\)
- lookvar: \(x : B\) \(ns : B\) \(vs : D\)
- eeaval: \(e : B\) \(ns : B\) \(vs : D\) \(1 : D\) \(p : B\)
- run: \(p : B\) \(l : D\) \(input : D\)

Specialisation points:
- Call 13 in eeaval to eeaval
C.1.2 int-while-dynscope

;;; Simple Scheme interpreter with dynamic scoping
(define (run data program)
  (evalexp (lookup-body 'main program) ; 1, 2
    (lookup-paramnames 'main program) ; 3
    data program))

(define (function? funname program)
  (and (pair? program) (or (eq? funname (caadar program))
    (function? funname (cdr program)))))

(define (lookup-paramnames funname program)
  (if (eq? funname (caadar program)) (cdadar program)
    (lookup-paramnames funname (cdr program)))))

(define (lookup-body funname program)
  (if (eq? funname (caadar program)) (caddar program)
    (lookup-body funname (cdr program)))))

(define (variable? varname names)
  (and (pair? names)
    (or (eq? varname (car names)) (variable? varname (cdr names)))))

(define (lookup-value varname names values)
  (if (eq? varname (car names)) (car values)
    (lookup-value varname (cdr names) (cdr values))))

(define (evalexp exp names values program)
  (cond
    ((list? exp)
      (case (car exp)
        ((QUOTE) (cadr exp))
        ((LET LET*)
          (let* ((value
            (evalexp (car (cdaadr exp)) names values program))) ; 1
            (evalexp (caddr exp) ; 2
              (cons (caaadr exp) names)
              (cons value values)
              program)))
        (ELSE)
          (if (function? (car exp) program) ; 6
            (evalexp (lookup-body (car exp) program) ; 7, 8
              (append ;; DYNAMIC SCOPING
                (lookup-paramnames (car exp) program) ; 9
                (lookup-value (car exp) (cdr exp) names values program) ; 10
                (append ;; DYNAMIC SCOPING
                  (argvals (cdr exp) names values program) ; 11
                  ;; DYNAMIC SCOPING
                  program))
            (if (eq? (cadr exp) names values program) ; 3
              (evalexp (caddr exp) names values program) ; 4
              (evalexp (cadddr exp) names values program))) ; 5
          (else
            (if (function? (car exp) program)
              (evalexp (lookup-body (car exp) program) ; 7, 8
                (append ;; DYNAMIC SCOPING
                  (lookup-paramnames (car exp) program) ; 9
                  (lookup-value (car exp) (cdr exp) names values program) ; 10
                  (append ;; DYNAMIC SCOPING
                    (argvals (cdr exp) names values program) ; 11
                    ;; DYNAMIC SCOPING
                    program))
              (if (eq? (cadr exp) names values program) ; 3
                (evalexp (caddr exp) names values program) ; 4
                (evalexp (cadddr exp) names values program))) ; 5
            (else
              ;; it must be a constant
              (apply (eval (car exp) (scheme-report-environment 5))
                (argvals (cdr exp) names values program))))) ; 12
          (else ;; it must be a base function
            (apply (eval (car exp) (scheme-report-environment 5))
              (argvals (cdr exp) names values program))))) ; 12
          (else ;; it must be a constant
            exp)))

(define (argvals exps names values program)
  (if (null? exps) '()
    (cons (evalexp (car exps) names values program)
      (argvals (cdr exps) names values program)))))

Parameter binding times:
argvals: exps : B names : S values : D program : B
evalexp: exp : B names : S values : D program : B
lookup-value: varname : B names : S values : D
variable?: varname : B names : S
lookup-body: funname : B program : B
lookup-paramnames: funname : B program : B
function?: funname : B program : B
run: data : D program : B
Specialisation points:
Call 1 in variable? to variable?
Call 1 in lookup-value to lookup-value
Call 7 in evalexp to evalexp

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
C.1.3 int-while-statscope

;;; Simple Scheme interpreter with static lexical scoping
(define (run data program)
  (evalexp (lookup-body 'main program) ; 1, 2
    (lookup-paramnames 'main program) ; 3
    data program))

(define (function? funname program)
  (and (pair? program) (or (eq? funname (caadar program))
    (function? funname (cdr program)))))

(define (lookup-paramnames funname program)
  (if (eq? funname (caadar program)) (cdadar program)
    (lookup-paramnames funname (cdr program)))))

(define (lookup-body funname program)
  (if (eq? funname (caadar program)) (caddar program)
    (lookup-body funname (cdr program)))))

(define (variable? varname names)
  (and (pair? names)
    (or (eq? varname (car names)) (variable? varname (cdr names)))))

(define (lookup-value varname names values)
  (if (eq? varname (car names)) (car values)
    (lookup-value varname (cdr names) (cdr values))))

(define (evalexp exp names values program)
  (cond
    ((list? exp)
      (case (car exp)
        ((QUOTE) (cadr exp))
        ((LET LET*)
          (let* ((value
            (evalexp (car (cdaadr exp)) names values program))) ; 1
            (evalexp (caddr exp) ; 2
              (cons (caaadr exp) names
                (cons value values)
                program))))
        ((IF)
          (if (evalexp (cadr exp) names values program) ; 3
            (evalexp (caddr exp) names values program) ; 4
            (evalexp (cadddr exp) names values program))) ; 5
        (else
          (if (function? (car exp) program) ; 6
            (evalexp (lookup-body (car exp) ; 7, 8
              (lookup-paramnames (car exp) program) ; 9
              (argvals (cdr exp) names values program) ; 10
              program))
            ;; else it must be a base function
            (apply (eval (car exp) (scheme-report-environment 5))
              (argvals (cdr exp) names values program))))) ; 11
    ((variable? exp names) ; 12
      (lookup-value exp names values))
    (else ;; it must be a constant
      exp)))))

(define (argvals exps names values program)
  (if (null? exps) '()
    (cons (evalexp (car exps) names values program)
      (argvals (cdr exps) names values program))))

Parameter binding times:
argvals: exps : B names : B values : D program : B
evalexp: exp : B names : B values : D program : B
lookup-value: exp : B names : B values : D program : B
variable?: varname : B names : B
lookup-paramnames: funname : B program : B
function?: funname : B program : B
run: data : D program : B

Specialisation points:
Call 7 in evalexp to evalexp

C.1.4 lambdaint

;;; Reducer for the lambda calculus
;;; Representation:
;;; R [[n]] = (1 0 ... 0) n zeros

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
Termination Analysis and Specialization-Point Insertion in Partial Evaluation

::: \[ R [[\text{\textit{a}} \text{\textit{e}}]] = (2 R [[\text{\textit{a}}]]) R [[\text{\textit{e}}]] \]

::: \[ R [[\text{\textit{e}} \text{\textit{e}}]] = (3 R [[\text{\textit{e}}]]) R [[\text{\textit{e}}]] \]

\[(\text{define} \ (\text{lambdaint} \ \text{e}) \ (\text{red} \ \text{e}))\]

\[(\text{define} \ (\text{red} \ \text{e}) \ ; \text{reduce lambda expression} \ \text{e})\]

\[
(\text{if} \ (\text{isvar?} \ \text{e})
  \begin{align*}
  & (\text{let*} \ ((f \ (\text{red} \ (\text{app->e1} \ \text{e}))))
  & (a \ (\text{red} \ (\text{app->e2} \ \text{e}))))
  & (\text{if} \ (\text{islam?} \ f)
  & (\text{red} \ (\text{subst} \ (\text{lam->var} \ f) \ a \ (\text{lam->body} \ f)))
  & (\text{mkapp} \ f \ a)))
  
  (\text{define} \ (\text{subst} \ \text{x} \ \text{a} \ \text{e})
  \begin{align*}
  & (\text{if} \ (\text{isvar?} \ \text{e}) \ (\text{equal?} \ \text{x} \ \text{e}) \ \text{a} \ \text{e})
  & (\text{if} \ (\text{islam?} \ \text{e})
  & (\text{if} \ (\text{equal?} \ \text{x} \ (\text{lam->var} \ \text{e}))
  & (\text{mklam} \ (\text{lam->var} \ \text{e}) \ (\text{subst} \ \text{x} \ \text{a} \ (\text{lam->body} \ \text{e})))
  & (\text{mkapp} \ (\text{subst} \ \text{x} \ \text{a} \ (\text{app->e1} \ \text{e})) \ (\text{subst} \ \text{x} \ \text{a} \ (\text{app->e2} \ \text{e})))))))
  
  (\text{define} \ (\text{isvar?} \ \text{e}) \ (\text{equal?} \ (\text{car} \ \text{e}) \ 1))
  
  (\text{define} \ (\text{islam?} \ \text{e}) \ (\text{equal?} \ (\text{car} \ \text{e}) \ 2))
  
  (\text{define} \ (\text{mklam} \ \text{n} \ \text{e}) \ (\text{cons} \ 2 \ (\text{cons} \ \text{n} \ (\text{cons} \ \text{e} \ '()))))
  
  (\text{define} \ (\text{lam->var} \ \text{e} \ (\text{cad dr} \ \text{e})))
  
  (\text{define} \ (\text{lam->body} \ \text{e} \ (\text{cad dr} \ \text{e})))
  
  (\text{define} \ (\text{mkapp} \ \text{e1} \ \text{e2}) \ (\text{cons} \ 3 \ (\text{cons} \ \text{e1} \ (\text{cons} \ \text{e2} \ '())))))
  
  (\text{define} \ (\text{app->e1} \ \text{e}) \ (\text{cad dr} \ \text{e})))
  
  (\text{define} \ (\text{app->e2} \ \text{e}) \ (\text{cad dr} \ \text{e})))

\text{Parameter binding times:}

\text{app->e2: e : } S

\text{app->e1: e : S}

\text{mkapp: e1 : S e2 : S}

\text{lam->body: e : S}

\text{lam->var: e : S}

\text{mklam: n : S e : S}

\text{islam?: e : S}

\text{isvar?: e : S}

\text{subst: x : S a : S e : S}

\text{red: e : S}

\text{lambdaint: e : B}

\text{Specialisation points:}

\text{Call 6 in subst to subst}

\text{Call 8 in red to red}

\text{Call 9 in subst to subst}

\text{Call 11 in subst to subst}

\text{Call 3 in red to red}

\text{C.1.5 parsexp}

\text{::: Parse a list of atoms as an expression. Return remaining list.}

\text{::: e.g. } {'1' '2' '3' '4' '5' '6' '7' '8'

\text{define} \ (\text{parsexp} \ \text{xs}) \ (\text{expr} \ \text{xs})))

\text{define} \ (\text{expr} \ \text{xs})

\text{define} \ (\text{term} \ \text{xs})

\text{define} \ (\text{factor} \ \text{xs})

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
App-8    ·    A. Glenstrup and N. Jones

(define (member? x xs)
  (if (pair? xs)
      (if (equal? x (car xs))
          #t
          (member? x (cdr xs)))
    #f))

(define (atom xs)
  (if (pair? xs) (cdr xs) xs))

Parameter binding times:
  atom: xs : B
  member?: x : B xs : B
term: xs : B
expr: xs : B
parsexp: xs : B

Specialisation points:
None.

C.1.6   turing

;;; Turing machine interpreter

;;; instrs ::= '(instr . instrs)
;;; instr ::= '(Halt) ; Stop interpretation
;;;         | '(Write . x) ; Write x onto the tape at current pos
;;;         | '(Left) ; Move pos left, extend tape if needed
;;;         | '(Right) ; Move pos right, extend tape if needed
;;;         | '(Goto . i) ; Continue at instruction i
;;;         | '(IfGoto x . i) ; If current pos contains x, goto i

(define (run prog tapeinput) (turing prog '() tapeinput prog))
(define (turing instrs revltape rtape prog)
  (if (pair? instrs)
      (if (equal? 'Halt (caar instrs))
          rtape
        (if (equal? 'Write (caar instrs))
            (turing (cdr instrs) ; 1
              revltape (cons (cdar instrs) (cdr rtape)) prog)
        (if (equal? 'Left (caar instrs))
            (if (pair? revltape)
                (turing (cdr instrs) ; 2
                  (cdr revltape)
                  (cons (car revltape) rtape) prog)
              (turing (cdr instrs) ; 3
                () (cons 'Blank rtape) prog))
        (if (equal? 'Right (caar instrs))
            (if (pair? rtape)
                (turing (cdr instrs) ; 4
                  (cons (car rtape) revltape)
                  (cdr rtape) prog)
              (turing (cdr instrs) ; 5
                (cons 'Blank revltape)
                () prog))
        (if (equal? 'Goto (caar instrs))
            (turing (lookup (cdar instrs) prog) revltape rtape prog) ; 6, 7
        (if (equal? 'IfGoto (caar instrs))
            (if (equal? (car rtape) (cadar instrs))
                (turing ; 8
                  (lookup (cddar instrs) prog) revltape rtape prog)
              (turing (cdr instrs) revltape rtape prog) ; 9
                (turing (cdr instrs) revltape rtape prog) ; 10
                rtape))))
  )))

(define (lookup i instrs)
  (if (= i 1) instrs (lookup (- i 1) (cdr instrs))))

Parameter binding times:
  lookup: 1 : B instrs : B
turing: instrs : B revltape : D rtape : D prog : B
run: prog : B tapeinput : B

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
Specialisation points:
Call 8 in turing to turing
Call 6 in turing to turing

C.2 Algorithms

C.2.1 ack
;;; Ackermann’s function, numbers represented by list length
(define (goal m n) (ack m n))
(define (ack m n)
  (if (equal? '() m)
      (cons 1 n)
      (if (equal? '() n)
          (ack (cdr m) '(1))
          (ack (cdr m) (ack m (cdr n))))))

Parameter binding times:
ack: m : B n : D
goal: m : B n : B

Specialisation points:
Call 3 in ack to ack

C.2.2 binom
;;; Binomial function, numbers represented by list length
(define (goal n k) (binom n k))
(define (binom n k)
  (if (equal? '() n)
      '(1)
      (if (equal? '() k)
          '(1)
          (+ (binom (cdr n) (cdr k)) (binom (cdr n) k)))))

Parameter binding times:
binom: n : B k : D
goal: n : B k : B

Specialisation points:
None.

Parameter binding times:
binom: n : D k : B
goal: n : B k : B

Specialisation points:
Call 2 in binom to binom

C.2.3 gcd-1
;;; Greatest common divisor, numbers represented by list length
(define (goal x y) (gcd x y))
(define (gcd x y)
  (if (or (equal? x '()) (equal? y '()))
      'error
      (if (equal? x y)
          x
          (if (gt x y)
              (gcd (monus x y) y)
              (gcd x (monus y x))))))

(define (gt x y)
  (if (equal? x '())
      #f
      (if (equal? y '())
          #t
          (gt (cdr x) (cdr y)))))

(define (monus x y)
  (if (equal? (lgth y) 1)
      (cdr x)
      (monus (cdr x) (cdr y)))))

(define (lgth x) (if (equal? x '()) 0 (+ 1 (lgth (cdr x))))))

Parameter binding times:
lgth: x : D
monus: x : D y : D
gt: x : D y : D
gcd: x : D y : D
goal: x : B y : B

Specialisation points:
Call 1 in gt to gt
Call 2 in monus to monus
Call 4 in gcd to gcd
Call 1 in lgth to lgth
Call 2 in gcd to gcd
C.2.4  \textit{gcd-2}

;;; Greatest common divisor, numbers represented by list length
;;; \texttt{x} and \texttt{y} are swapped in the recursive call
(define (goal \texttt{x} \texttt{y}) (gcd \texttt{x} \texttt{y}))
(define (gcd \texttt{x} \texttt{y})
  (if (or (equal? \texttt{x} '()) (equal? \texttt{y} '()))
    'error
    (if (equal? \texttt{x} \texttt{y})
      \texttt{x}
      (if (gt \texttt{x} \texttt{y})
        (gcd \texttt{y} (monus \texttt{x} \texttt{y}))
        (gcd (monus \texttt{y} \texttt{x}) \texttt{x}))))
(define (gt \texttt{x} \texttt{y})
  (if (equal? \texttt{x} '())
    #f
    (if (equal? \texttt{y} '())
      #t
      (gt (cdr \texttt{x}) (cdr \texttt{y})))))
(define (monus \texttt{x} \texttt{y})
  (if (equal? (lgth \texttt{y}) 1)
    (cdr \texttt{x})
    (monus (cdr \texttt{x}) (cdr \texttt{y}))))
(define (lgth \texttt{x})
  (if (equal? \texttt{x} '())
    0
    (+ 1 (lgth (cdr \texttt{x})))))

Parameter binding times:
\texttt{lgth: x : D}
\texttt{monus: x : D y : D}
\texttt{gt: x : D y : D}
\texttt{gcd: x : D y : D}
\texttt{goal: x : B y : B}

Specialisation points:
Call 2 in \texttt{monus} to \texttt{monus}
Call 1 in \texttt{lgth} to \texttt{lgth}
Call 1 in \texttt{gt} to \texttt{gt}
Call 4 in \texttt{gcd} to \texttt{gcd}
Call 2 in \texttt{gcd} to \texttt{gcd}

C.2.5  \textit{graphcolour-1}

;;; Colour graph \textit{G} with colours \texttt{cs} so that neighbors have different colours
;;; The graph is represented as a list of nodes with adjacency lists
;;; Example:
;;; \texttt{'((a . (b c d)) (b . (a c e)) (c . (a b d e f)) (d . (a c f))
;;; (e . (b c f)) (f . (c d e)))}
(define (graphcolour \texttt{G} \texttt{cs})
  (let* ((\texttt{ns} \texttt{G})) ; to speed up: (\texttt{ns} (sortnodesbyarity \texttt{G}))
    (reverse
     (colorrest \texttt{cs} \texttt{cs}
       (cons (colornode \texttt{cs} (car \texttt{ns}) '()) '())
       (cdr \texttt{ns})))))

;;; Colour a node by appending a colour list to the node. The head of
;;; the list is the chosen colour, the tail are the yet untried
;;; colours. If impossible, return nil.
;;; Example of coloured node: \texttt{('(red blue yellow) . (a . (b c d)))}
(define (colornode \texttt{cs} \texttt{node} \texttt{colorednodes})
  (if (pair? \texttt{cs})
    (if (possible (car \texttt{cs}) (cdr \texttt{node}) \texttt{colorednodes})
      (cons \texttt{cs} \texttt{node})
      (colornode (cdr \texttt{cs}) \texttt{node} \texttt{colorednodes}))
    '()))

;;; Can we use color with these adjacent nodes and current coloured nodes?
;;; (define (possible color adjs colorednodes)
  (if (pair? adjs)
    (if (equal? color (colorof (car adjs) \texttt{colorednodes}))
      (possible (color adj) \texttt{colorednodes})
      #f
    )
  )

;;; Return colour of node. If no colour yet, return nil.
;;; (define (colorof node \texttt{colorednodes})
  (if (pair? node \texttt{colorednodes})
    (if (equal? (cadar \texttt{node}) \texttt{node} \texttt{colorednodes})
      (colorof node (cdr \texttt{node} \texttt{colorednodes}))
      '())
    )

;;; Colour the first node of rest with colours from \texttt{ncs}, and
;;; colour remaining nodes. If impossible, return nil.
;;; (define (colorest \texttt{cs} \texttt{ncs} \texttt{colorednodes} \texttt{rest})
  (if (pair? rest)
    (let* ((\texttt{(colorednode \texttt{colorednode \texttt{ncs} \texttt{car \texttt{rest} \texttt{colorednodes}})}}
      (if (pair? \texttt{colorednode})
      )

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
(let* ((colored (colorrest cs cs
(cons colorednode colorednodes)
(cdr rest))))
  (if (pair? colored)
      colored
    ; if remaining nodes are not colourable, and there
    (if (pair? (car colorednode)) ; are colours left,
      (colorrest-thetrick
       cs cs (cdr (car colorednode))) ; try next colour
       '()))
    '()))
  colorednodes))

(define (colorrest-thetrick cs1 cs ncs colorednodes rest)
  (if (equal? cs1 ncs)
      (colorrest cs cs1 colorednodes rest)
    (colorrest-thetrick (cdr cs1) cs ncs colorednodes rest)))

(define (reverse xs) (revapp xs '()))
(define (revapp xs rest)
  (if (pair? xs)
      (revapp (cdr xs) (cons (car xs) rest))
    rest))

Parameter binding times:
revapp: xs : D rest : D
reverse: xs : D
colorrest-thetrick: cs1 : D cs : D ncs : D colorednodes : D rest : B
colorrest: cs : D ncs : D colorednodes : D rest : B
colorof: node : B colorednodes : D
possible: color : D adjn : B colorednodes : D
colornode: cs : D node : B colorednodes : D
graphcolour: g : B cs : D

Specialisation points:
Call 1 in colorof to colorof
Call 2 in colorrest-thetrick to colorrest-thetrick
Call 1 in revapp to revapp
Call 3 in revapp to revapp

------------------------------------

Parameter binding times:
revapp: xs : D rest : D
reverse: xs : D
colorrest-thetrick: cs1 : B cs : B ncs : D colorednodes : D rest : B
colorrest: cs : B ncs : B colorednodes : D rest : D
colorof: node : D colorednodes : D
possible: color : B adjn : B colorednodes : D
colornode: cs : B node : D colorednodes : D
graphcolour: g : D cs : B

Specialisation points:
Call 1 in colorof to colorof
Call 2 in colorrest to colorrest
Call 2 in possible to possible
Call 1 in revapp to revapp
Call 3 in colorrest to colorrest-thetrick

----------

C.2.6 graphcolour-2

;;; Colour graph G with colours cs so that neighbors have different
;;; colours (slightly tail recursive version)
;;; The graph is represented as a list of nodes with adjacency lists
;;; Example:
;;; '((a . (b c d)) (b . (a c e)) (c . (a b d e f)) (d . (a c f))
;;;  (e . (b c f)) (f . (c d e)))
;;; define (graphcolour G cs)
;;; (let* ((ns G) ; to speed up: (ns (sortnodesbyarity G))
;;;; (reverse
;;; (colorrest cs cs
;;; (cons (colornode cs (car ns)) '()) '())
;;; '()))
;;; Colour a node by appending a colour list to the node. The head of
;;; the list is the chosen colour, the tail are the yet untried
;;; colours. If impossible, return nil.
;;; Example of coloured node: '(((red blue yellow) . (a . (b c d))))
;;; (define (colornode cs node colorednodes)
;;; (if (pair? cs)
;;; (if (possible (car cs) (cdr node) colorednodes)
;;; (cons cs node)
;;; (colornode (cdr cs) node colorednodes))
;;; '())))

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
Can we use color with these adjacent nodes and current coloured nodes?

(define (possible color adjs colorednodes)
  (if (pair? adjs)
      (if (equal? color (colorof (car adjs) colorednodes))
        #f
        (possiblecolor (cdr adjs) colorednodes))
    #t))

;; Return colour of node. If no colour yet, return nil.
(define (colorof node colorednodes)
  (if (pair? colorednodes)
      (if (equal? (cadar colorednodes) node)
        (caaar colorednodes)
        (colorof node (cdr colorednodes)))
    '()))

;; Colour the first node of rest with colours from ncs, and
;; colour remaining nodes. If impossible, return nil.
(define (colorrest cs ncs colorednodes rest)
  (if (pair? rest)
      (colornoderest cs ncs (car rest) colorednodes rest)
      colorednodes))

;; Like colornode, only continue with colouring the rest
(define (colornoderest cs ncs node colorednodes rest)
  (if (pair? ncs)
      (if (possible (car ncs) (cdr node) colorednodes)
          (let* ((colored (colorrest cs cs (cons (cons ncs node) colorednodes) (cdr rest))))
            (if (pair? colored)
                colored
                ;if remaining nodes are not colourable, and
                ;if (pair? ncs) ; there are some colours left,
                ;(colorrest-the-trick
                cs cs (cdr ncs) ; try next colour
                colorednodes rest)
            ('))))
          (colornoderest cs (cdr ncs) node colorednodes rest))
    '()))

(define (colorrest-the-trick cs1 cs ncs colorednodes rest)
  (if (equal? cs1 ncs)
      (colorrest cs cs1 colorednodes rest)
      (colorrest-the-trick (cdr cs1) cs ncs colorednodes rest))))

(define (reverse xs) (revapp xs '()))
(define (revapp xs rest)
  (if (pair? xs)
      (revapp (cdr xs) (cons (car xs) rest))
      rest))

Parameter binding times:
  revapp: xs : D rest : D
  reverse: xs : D
  colornoderest: cs : D ncs : D node : B colorednodes : D rest : B
  colorrest: cs : D ncs : D colorednodes : D rest : B
  colorof: node : B colorednodes : D
  possible: color : D adjs : B colorednodes : D
  colornode: cs : D node : B colorednodes : B
  graphcolour: g : B cs : D

Specialisation points:
  Call 4 in colornoderest to colornoderest
  Call 1 in colorof to colorof
  Call 2 in colorrest-the-trick to colorrest-the-trick
  Call 1 in colornode to colornode
  Call 1 in revapp to revapp
  Call 1 in colorrest to colornoderest

Parameter binding times:
  revapp: xs : D rest : D
  reverse: xs : D
  colorrest-the-trick: cs1 : B cs : B ncs : B colorednodes : D rest : D
  colornoderest: cs : B ncs : B node : D colorednodes : D rest : D
  colorrest: cs : B ncs : B colorednodes : D rest : D
  colorof: node : D colorednodes : D
  possible: color : B adjs : D colorednodes : D
  colornode: cs : B node : D colorednodes : B
  graphcolour: g : D cs : B

Specialisation points:
C.2.7  \textit{graphcolour-3} \\
\textbullet{} colours graph $G$ with colours $cs$ so that neighbors have different \\
\textbullet{} the graph is represented as a list of nodes with adjacency lists \\
\textbullet{} Example: \\
\textbullet{} '((a . (b c d)) (b . (a c e)) (c . (a b d e f)) (d . (a c f)) \\
\textbullet{} (e . (b c f)) (f . (c d e))) \\
(\text{define} \ (\text{graphcolour} \ G \ cs) \\
\ (\text{let*} \ ((\text{ns} \ G)) \ ; \ \text{to speed up:} \ (\text{ns} \ \text{sortnodesbyarity} \ G)) \\
\ (\text{reverse} \\
\ (\text{colorrest} \ cs \ cs \\
\ (\text{cons} \ (\text{colornode} \ cs \ (\text{car} \ \text{ns}) \ '()) \ '()) \\
\ (\text{cdr} \ \text{ns}))))))) \\
\textbullet{} Colour a node by appending a colour list to the node. The head of \\
\textbullet{} the list is the chosen colour, the tail are the yet untried \\
\textbullet{} colours. If impossible, return nil. \\
\textbullet{} Example of coloured node: '((red blue yellow) . (a . (b c d))) \\
(\text{define} \ (\text{colornode} \ cs \ node \ colorednodes) \\
\ (\text{if} \ (\text{pair?} \ cs) \\
\ (\text{if} \ (\text{possible} \ (\text{car} \ cs) \ (\text{cdr} \ node) \ colorednodes) \\
\ (\text{cons} \ cs \ node) \\
\ (\text{colornode} \ (\text{cdr} \ cs) \ node \ colorednodes)) \\
\ '()))) \\
\textbullet{} Can we use color with these adjacent nodes and current coloured nodes? \\
(\text{define} \ (\text{possiblecolor} \ adjacs \ colorednodes) \\
\ (\text{if} \ (\text{pair?} \ adjacs) \\
\ (\text{if} \ (\text{equal?} \ \text{color} \ (\text{colorof} \ \text{car} \ adjacs) \ colorednodes)) \\
\ #f \\
\ (\text{possiblecolor} \ (\text{cdr} \ adjacs) \ colorednodes)) \\
\ #t)) \\
\textbullet{} Return colour of node. If no colour yet, return nil. \\
(\text{define} \ (\text{colorof} \ node \ colorednodes) \\
\ (\text{if} \ (\text{pair?} \ colorednodes) \\
\ (\text{if} \ (\text{equal?} \ (\text{cadar} \ colorednodes) \ node) \\
\ (\text{caaar} \ colorednodes) \\
\ (\text{colorof} \ (\text{cdr} \ colorednodes))) \\
\ '()))) \\
\textbullet{} Colour the first node of rest with colours from ncs, and \\
\textbullet{} colour remaining nodes. If impossible, return nil. \\
(\text{define} \ (\text{colornoderest} \ cs \ ncs \ colorednodes \ rest) \\
\ (\text{if} \ (\text{pair?} \ rest) \\
\ (\text{colornoderest} \ cs \ ncs \ (\text{car} \ rest) \ colorednodes \ rest) \\
\ colorednodes)) \\
\textbullet{} Like colornode, only continue with colouring the rest \\
(\text{define} \ (\text{colornoderest} \ cs \ ncs \ node \ colorednodes \ rest) \\
\ (\text{if} \ (\text{pair?} \ ncs) \\
\ (\text{if} \ (\text{possible} \ (\text{car} \ ncs) \ (\text{cdr} \ node) \ colorednodes) \\
\ (\text{let*} \ ((\text{colored} \ \text{colornoderest} \ cs \ ncs \ (\text{car} \ rest) \ colorednodes \ rest) \\
\ (\text{trynextcolour} \ colorednodes \ rest)) \\
\ '()))) \\
\ (\text{if} \ (\text{pair?} \ colored) \\
\ colored \\
\ (\text{if} \ (\text{colorof} \ \text{node} \ colorednodes) \\
\ (\text{trynextcolour} \ colorednodes \ rest)) \\
\ '()))) \\
\textbullet{} Can we use color with these adjacent nodes and current coloured nodes? \\
(\text{define} \ (\text{possiblecolor} \ adjacs \ colorednodes) \\
\ (\text{if} \ (\text{pair?} \ adjacs) \\
\ (\text{if} \ (\text{equal?} \ \text{color} \ (\text{colorof} \ \text{car} \ adjacs) \ colorednodes)) \\
\ #f \\
\ (\text{possiblecolor} \ (\text{cdr} \ adjacs) \ colorednodes)) \\
\ #t)) \\
\textbullet{} Return colour of node. If no colour yet, return nil. 
(\text{define} \ (\text{colorof} \ node \ colorednodes) \\
\ (\text{if} \ (\text{pair?} \ colorednodes) \\
\ (\text{if} \ (\text{equal?} \ (\text{cadar} \ colorednodes) \ node) \\
\ (\text{caaar} \ colorednodes) \\
\ (\text{colorof} \ (\text{cdr} \ colorednodes))) \\
\ '()))) \\
\textbullet{} Colour the first node of rest with colours from ncs, and \\
\textbullet{} colour remaining nodes. If impossible, return nil. 
(\text{define} \ (\text{colornoderest} \ cs \ ncs \ colorednodes \ rest) \\
\ (\text{if} \ (\text{pair?} \ rest) \\
\ (\text{colornoderest} \ cs \ ncs \ (\text{car} \ rest) \ colorednodes \ rest) \\
\ colorednodes)) \\
\textbullet{} Like colornode, only continue with colouring the rest 
(\text{define} \ (\text{colornoderest} \ cs \ ncs \ node \ colorednodes \ rest) \\
\ (\text{if} \ (\text{pair?} \ ncs) \\
\ (\text{if} \ (\text{possible} \ (\text{car} \ ncs) \ (\text{cdr} \ node) \ colorednodes) \\
\ (\text{let*} \ ((\text{colored} \ \text{colornoderest} \ cs \ ncs \ (\text{car} \ rest) \ colorednodes \ rest) \\
\ (\text{trynextcolour} \ colorednodes \ rest)) \\
\ '()))) \\
\ (\text{if} \ (\text{pair?} \ colored) \\
\ colored \\
\ (\text{if} \ (\text{colorof} \ \text{node} \ colorednodes) \\
\ (\text{trynextcolour} \ colorednodes \ rest)) \\
\ '()))) \\
\textbullet{} Can we use color with these adjacent nodes and current coloured nodes? 
(\text{define} \ (\text{possiblecolor} \ adjacs \ colorednodes) \\
\ (\text{if} \ (\text{pair?} \ adjacs) \\
\ (\text{if} \ (\text{equal?} \ \text{color} \ (\text{colorof} \ \text{car} \ adjacs) \ colorednodes)) \\
\ #f \\
\ (\text{possiblecolor} \ (\text{cdr} \ adjacs) \ colorednodes)) \\
\ #t)) \\
\textbullet{} Return colour of node. If no colour yet, return nil. 
(\text{define} \ (\text{colorof} \ node \ colorednodes) \\
\ (\text{if} \ (\text{pair?} \ colorednodes) \\
\ (\text{if} \ (\text{equal?} \ (\text{cadar} \ colorednodes) \ node) \\
\ (\text{caaar} \ colorednodes) \\
\ (\text{colorof} \ (\text{cdr} \ colorednodes))) \\
\ '()))) \\
\textbullet{} Colour the first node of rest with colours from ncs, and \\
\textbullet{} colour remaining nodes. If impossible, return nil. 
(\text{define} \ (\text{colornoderest} \ cs \ ncs \ colorednodes \ rest) \\
\ (\text{if} \ (\text{pair?} \ rest) \\
\ (\text{colornoderest} \ cs \ ncs \ (\text{car} \ rest) \ colorednodes \ rest) \\
\ colorednodes)) \\
\textbullet{} Like colornode, only continue with colouring the rest 
(\text{define} \ (\text{colornoderest} \ cs \ ncs \ node \ colorednodes \ rest) \\
\ (\text{if} \ (\text{pair?} \ ncs) \\
\ (\text{if} \ (\text{possible} \ (\text{car} \ ncs) \ (\text{cdr} \ node) \ colorednodes) \\
\ (\text{let*} \ ((\text{colored} \ \text{colornoderest} \ cs \ ncs \ (\text{car} \ rest) \ colorednodes \ rest) \\
\ (\text{trynextcolour} \ colorednodes \ rest)) \\
\ '()))) \\
\ (\text{if} \ (\text{pair?} \ colored) \\
\ colored \\
\ (\text{if} \ (\text{colorof} \ \text{node} \ colorednodes) \\
\ (\text{trynextcolour} \ colorednodes \ rest)) \\
\ '()))) 

Parameter binding times: 
\textit{revapp}: \ \text{xs} : \text{D} \ \text{rest} : \text{D} \\
\textit{reverse}: \ \text{xs} : \text{D} 

\textit{ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.}
Specialisation points:
Call 1 in colorof to colorof
Call 4 in colornoderest to colornoderest
Call 2 in colornode to colornode
Call 1 in revapp to revapp
Call 1 in colorrest to colornoderest

Parameter binding times:
revapp: xs : D rest : D
to reverse: xs : D
colornoderest: cs : B ncs : B node : D colorednodes : D rest : D
to colorrest: cs : B ncs : B colorednodes : D rest : D
to colorof: node : D colorednodes : D
possible: color : B adjs : D colorednodes : D
to possible: color : B colorednodes : D
colornode: cs : B node : D colorednodes : B
to graphcolour: g : D cs : D

Specialisation points:
Call 1 in colorof to colorof
Call 2 in colornode to colornode
Call 1 in revapp to revapp
Call 1 in colorrest to colornoderest

---

C.2.8 match

;; Simple pattern matcher
(define (match p s) (loop p s p s))
(define (loop p s pp ss)
  (if (equal? p '()) #t
      (if (equal? s '()) #f
        (if (equal? (car p) (car s))
            (loop (cdr p) (cdr s) pp ss)
            (loop pp (cdr ss) pp (cdr ss))))))

Parameter binding times:
loop: p : B s : D pp : B ss : D
match: p : B s : D

Specialisation points:
Call 2 in loop to loop

---

C.2.9 power

;; Power function: x to the nth power
;; (numbers represented by list length)
(define (goal x n) (power x n))
(define (power x n)
  (if (equal? n '()) '(1) (mult x (power x (cdr n)))))
(define (mult x y)
  (if (equal? y '()) #f
      (add x (mult x (cdr y)))))
(define (add x y)
  (if (equal? y '()) x (cons 1 (add x (cdr y)))))

Parameter binding times:
add: x : B y : D
to mult: x : B y : D
power: x : B n : D
goal: x : B n : B

Specialisation points:
Call 2 in mult to mult
Call 1 in add to add
Call 1 in power to power

---

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
Specialisation points:
  Call 1 in add to add
  Call 2 in mult to mult

C.2.10  reach

;; How can node v be reached from node u in a directed graph.
;; Graph example: '((a . b) (a . d) (b . d) (c . a))
(define (goal u v edges) (reach u v edges))
(define (reach u v edges)
  (if (member? (cons u v) edges)
      (cons (cons u v) '())
      (via u v edges edges)))

(define (via u v rest edges)
  (if (equal? rest '())
      ()
      (via u v (cdr rest) edges)))

(define (member? x xs)
  (if (equal? xs '()) #f
      (if (equal? x (car xs)) #t (member? x (cdr xs)))))

Parameter binding times:
  member?: x : D xs : B
  via: u : B v : D rest : B edges : B
  reach: u : B v : D edges : B
  goal: u : B v : D edges : B
Specialisation points:
  Call 2 in reach to via

Parameter binding times:
  member?: x : B xs : B
  via: u : B v : D rest : B edges : B
  reach: u : B v : D edges : B
  goal: u : B v : D edges : B
Specialisation points:
  Call 2 in reach to via

Parameter binding times:
  member?: x : D xs : B
  via: u : D v : B rest : B edges : B
  reach: u : D v : B edges : B
  goal: u : D v : B edges : B
Specialisation points:
  Call 2 in reach to via

C.2.11   rematch

;; Regular expression pattern matcher
;;; pat ::= "." | character | \"character
;;; | pat*" | (pat) | pat ... pat
;;; When parsed, this is represented by:
;;; pat ::= ('dot) | ('char c) | ('star pat) | ('seq pat ... pat)
(define (rematch patstr str)
  (let* ((matchrest (domatch (parsepat patstr) (string->list str))))
    (if (pair? matchrest)
        (cons (list->string (reverse (car matchrest)))
              (list->string (cdr matchrest)))
        matchrest)))

(define (parsepat patstr) (parsep (string->list patstr) '() '()))

Parameter binding times:
  (parsep patchars seq stack)
  (if (pair? patchpars)
      (parsep-dot patchpars seq stack)
      (parsep-star patchpars seq stack))

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
(define (parsep-dot patchars seq stack)
  (parsep (cdr patchars) (cons (cons 'dot '()) seq) stack))

(define (parsep-star patchars seq stack)
  (if (pair? seq)
      (parsep
        (cdr patchars)
        (cons (cons 'star (cons (car seq) '()))
                (cadr seq))
        stack)
      (parsep
        (cdr patchars)
        (cons (cons 'char (cons #\ '())) '())
        stack)))

(define (parsep-openb patchars seq stack)
  (parsep (cdr patchars) '() (cons seq stack)))

(define (parsep-closeb patchars seq stack)
  (if (pair? stack)
      (parsep
        (cdr patchars)
        (cons (cons 'seq (reverse seq))
                (car stack))
        (cdr stack))
      (error "unmatched ')' in pattern")))

(define (parsep-char patchars seq stack)
  (if (pair? patchars)
      (parsep (cdr patchars)
              (cons (cons 'char (cons (car patchars) '()))
                        seq)
              stack)
      (parsep patchars
              (cons (cons 'char (cons #\ '()))
                        seq)
              stack)))

; domatch* cs must match on as much of cs as possible,
; Assume cs = cs1 ++ cs2, where cs1 has been matched. Then
; ((reverse cs1) . cs2) is returned

(define (domatch pat cs)
  (if (pair? pat)
      (if (equal? (car pat) 'dot) (domatch-dot cs)
          (if (equal? (car pat) 'char) (domatch-char cs (cadr pat))
              (if (equal? (car pat) 'star) (domatch-star cs (cadr pat) '())
                  (if (equal? (car pat) 'seq) (domatch-seq cs '() (cadr pat))
                      (error "unknown pattern data " pat))))
        (cons () cs)))

(define (domatch-dot cs)
  (if (pair? cs) (cons (cons (car cs) '()) (cadr cs)) 'nomatch))

(define (domatch-char cs c)
  (if (pair? cs)
      (if (equal? (car cs) c)
          (cons (cons (car cs) '()) (cadr cs)) 'nomatch)
      'nomatch))

(define (domatch-star cs pat init)
  ; init holds the chars already star-matched
  (if (pair? cs)
      (let* ((first (domatch pat cs))
             (lpat (first domatch pat cs))
             (lcs (map (lambda (x) (cadr x)) first))
             (lcs2 (foreach (lambda (x) (cadr x)) lcs))
             (lfirst (first first))
             (init (last init))
             (linit (last init)))
        (if (pair? first)
            (domatch-star (cdr first) pat (append (car first) init))
            (error "bad star match")))
      'nomatch))
(define (domatch-seq cs rest pats)  
  ; domatch-seq matches first pattern on cs = match ++ cs' and the  
  ; remaining patterns on cs' ++ rest  
  (if (pair? pats)  
    (let* ((first (domatch (car pats) cs)))  
      (if (pair? first)  
        (let* ((next  
             (domatch-seq  
               (append (cdr first) rest) '() (cdr pats))))  
          (if (pair? next)  
            (cons (append (car next) (car first)) (cdr next))  
              (if (pair? (car first))  
                (domatch-seq  
                  (reverse (cdar first))  
                  (cons (caar first) (append (cdr first) rest))  
                  pats)  
                'nomatch))  
          (if (pair? (car first))  
            (domatch-seq  
              (reverse (cdar first))  
              (cons (caar first) (append (cdr first) rest))  
              pats)  
          'nomatch))  
        (cons '() (append cs rest)))))

Parameter binding times:
  domatch-seq: cs : D rest : D pats : D  
  domatch-star: cs : D pat : D init : D  
  domatch-char: cs : D c : D  
  domatch-dot: cs : D  
  domatch: pat : D cs : D  
  parsep-char: patchars : D seq : S stack : S  
  parsep-closeb: patchars : D seq : S stack : S  
  parsep-openb: patchars : D seq : S stack : S  
  parsep-star: patchars : D seq : S stack : S  
  parsep-dot: patchars : D seq : S stack : S  
  parsep: patchars : D seq : S stack : S  
  rematch: patstr : D str : B  

Specialisation points:
  Call 2 in domatch-star to domatch-star  
  Call 3 in domatch-seq to domatch-seq  
  Call 6 in parsep to parsep-char

----------------------------

Parameter binding times:
  domatch-seq: cs : D rest : D pats : D  
  domatch-star: cs : D pat : D init : D  
  domatch-char: cs : D c : D  
  domatch-dot: cs : D  
  domatch: pat : D cs : D  
  parsep-char: patchars : D seq : D stack : D  
  parsep-closeb: patchars : D seq : D stack : D  
  parsep-openb: patchars : D seq : D stack : D  
  parsep-star: patchars : D seq : D stack : D  
  parsep-dot: patchars : D seq : D stack : D  
  parsep: patchars : D seq : D stack : D  
  rematch: patstr : D str : B  

Specialisation points:
  Call 2 in domatch-star to domatch-star  
  Call 2 in domatch-seq to domatch-seq  
  Call 1 in parsep to parsep-dot  
  Call 3 in domatch-seq to domatch-seq  
  Call 3 in domatch to domatch-star  
  Call 2 in parsep to parsep-star  
  Call 2 in parsep-char to parsep  
  Call 1 in domatch-seq to domatch  
  Call 3 in parsep to parsep-openb  
  Call 4 in parsep to parsep-closeb  
  Call 1 in parsep-char to parsep

C.2.12 strmatch

;;; Naive pattern string matcher
;;; strmatch returns a list of indices indicating the
;;; positions in str where patstr occurs
(define (strmatch patstr str)  
  (domatch (string->list patstr) (string->list str) 0))
(define (domatch pats cs m)
App-18  ·  A. Glenstrup and N. Jones

(if (pair? cs)
  (if (prefix precs cs)
      ; 1
      (cons n (domatch patcs (cdr cs) (+ n 1)))
    (domatch patcs (cdr cs) (+ n 1)))
  (if (equal? patcs cs) (cons n '()) '()))

declare (prefix precs cs)
  (or (null? precs)
    (and (pair? cs) (and (equal? (car precs) (car cs))
      (prefix (cdr precs) (cdr cs))))))

Parameter binding times:
prefix: precs : B cs : D
domatch: patcs : B cs : D n : S
strmatch: patstr : B str : D
Specialisation points:
  Call 3 in domatch to domatch
  Call 2 in domatch to domatch

----------------------------
Parameter binding times:
prefix: precs : D cs : B
domatch: patcs : D cs : B n : B
strmatch: patstr : D str : B
Specialisation points:
None.

C.2.13  typeinf

;; Type inference for the Typed Lambda Calculus
;; e ::= ('var . x) variable x
;; | ('apply . (e1 . e2)) apply abstraction e1 to expression e2
;; | ('lambda . (x . e1)) make lambda abstraction
;; t ::= ('tyvar . a)
;; | ('arrow . (t1 . t2))
;; -----------------------------
;; (define inittenv 1) ; tenv simply holds the next fresh type variables
;; (define (typeinf inittenv e) ; infer the type of e
  (let* ((atenv (freshtvar inittenv)))
    (car (etype '() (cdr atenv) e (car atenv)))))

;; (define (freshtvar tenv) (cons (cons 'Tvar tenv) (+ tenv 1)))
;; (define (vtype venv x) ; return the type of x as found in environment
  (if (equal? x (caar venv))
    (cdar venv)
    (vtype (cdr venv) x)))

;; (define (tsubst a t t1) ; substitute t for occ's of type var a in t1
  (if (equal? 'Tvar (car t1))
    (if (equal? a (cdr t1)) t t1)
    (if (equal? 'Arr (car t1))
      (cons 'Arr (cons (tsubst a (cadr t1) (cddr t1))
        (tsubst a t (cddr t1))))
      (error 'tsubst-t1)))

;; (define (unify venv t1 t2) ; unify types t1 and t2, returning
  (if (equal? (tsubst venv t1 t2) t2)
    (error 'unify-t2))

;; (define (unify venv t1 t2) ; unify types t1 and t2, returning
  (if (equal? 'Tvar (car t1))
    (cons (tsubst venv (cadr t1) (cddr t1))
      (if (equal? 'Tvar (car t2))
        (cons (tsubst venv (cadr t2) (cddr t2))
          (let* ((venv1tx1 (unify venv (cadr t1) (cddr t1)))
            (venv2t2)
            (unify venv1tx1 (cadr t2) (cddr t2))))
        (error 'unify-t2)))
      (error 'unify-t1))))

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
(define (etype venv tenv e t) ; infer type of e, unified with type t
  (if (equal? 'Var (car e)) ; using variable environment
      (let* ((venv1 t1 (unify venv (vtype venv (cdr e)) t)))
        (cons (cdr venv1 t1) (cons (car venv1 t1) tenv)))
    (if (equal? 'App (car e))
      (let* ((atenv1 (freshtvar tenv))
              (t2venv2 tenv2)
              (etype venv (cdr atenv1) (cadr e) (car atenv1)))
        (let* ((t2venv2 tenv2)
                (etype (cadr t2venv2 tenv2)
                        (cddr t2venv2 tenv2)
                        (cddr e)
                        (cons 'Arr (cons (car t2venv2 tenv2) t1))))
          (t1 (car tvenv2 tenv2 tenv2 tenv2))
          ; t1 = ('Arr ( ? . t'))
          (cons (cddr t1 tenv2) (cddr e)
                (car atenv1))
          (tvenv3 tvenv3)
          (etype venv (cdr e) (car atenv1))
          (tvenv3 tvenv3)
          (etype (cadr tvenv3 tvenv3)
                  (cddr tvenv3 tvenv3)
                  (cddr e)
                  (cons 'Arr (cons (car tvenv3 tvenv3))
                        tvenv3))
          (tvenv3 tvenv3)
          (etype (cdr tvenv3 tvenv3)
                  (cddr tvenv3 tvenv3))
          (tvenv3 tvenv3)
          (etype (cdr tvenv3 tvenv3)
                  (cddr tvenv3 tvenv3))
          (error 'Error-in-lambda-expression e))))
    (error 'Error-in-lambda-expression e)))))

Parameter binding times:
ETYPE: venv : B tenv : B e : B t : B
Unify: venv : B tenv : B t1 : B t2 : B
Subst: venv : B a : B t : B
VType: venv : B x : B
Freshtvar: tenv : B
Typeinf: inittenv : B e : B
Specialisation points:
None.

C.3 Basic operations

C.3.1 add
;; Add two numbers unarily represented as '(s s s ... s)
(define (goal x y) (add x y))
(define (add x y) (if (equal? y '()) x (add (cons 1 x) (cdr y))))

Parameter binding times:
add: x : S y : D
Goal: x : B y : B
Specialisation points:
  Call 1 in add to add

Parameter binding times:
add: x : D y : B
Goal: x : B y : B
Specialisation points:
None.

C.3.2 addlists
;;; Add two lists elementwise
(define (goal xs ys) (addlist xs ys))
(define (addlist xs ys)
  (if (pair? xs)
      (cons (+ (car xs) (car ys)) (addlist (cdr xs) (cdr ys)))
      '())))

Parameter binding times:
addlist: xs : B ys : D
goal: xs : B ys : D
Specialisation points:
None.
C.3.3 anchored

;; Parameter $y$ anchored in parameter $x$
(define (goal $x$ $y$) (anchored $x$ $y$))
(define (anchored $x$ $y$)
  (if (= (equal? $x$ ‘()) $y$ (anchored (cdr $x$) (cons 1 $y$)))))

Parameter binding times:
anchored: $x$ : B $y$ : D
goal: $x$ : B $y$ : B
Specialisation points: None.

Parameter binding times:
anchored: $x$ : D $y$ : S
goal: $x$ : B $y$ : B
Specialisation points:
Call 1 in anchored to anchored

C.3.4 append

(define (goal $x$ $y$) (append $x$ $y$))
(define (append $xs$ $ys$)
  (if (= (equal? $xs$ ‘()) $ys$ (cons (car $xs$) (append (cdr $xs$) $ys$)))))

Parameter binding times:
append: $xs$ : B $ys$ : D
goal: $x$ : B $y$ : D
Specialisation points: None.

C.3.5 assocrw

;;;; Rewrite expression with associative operator ‘op’
;;;; a -> a1 b -> b1 c -> c1
;;;; ------------------------ ------------
;;;; ‘(op (op a b) c)’ -> ‘(op a1 (op b1 c1))’
;;;; ‘(op a b)’ -> ‘(op a1 b1)’
;;;; ‘(op a b)’ -> ‘(op a1 b)’
;;;; a != ‘op a’ -> a1
;;;; ‘(op a b)’ -> ‘(op a1 b)’
(define (assocrw exp) (rewrite exp))
(define (rewrite exp)
  (if (and (pair? exp)
            (= ‘op (car exp)))
      (let* ((opab (cadr exp)))
        (if (and (pair? opab)
                 (= ‘op (car opab)))
          (let* ((a1 (rewrite (cadr opab)))
                   (b1 (rewrite (caddr opab)))
                   (c1 (rewrite (caddr exp)))))
            (rewrite (cons (car exp) ; op
                           (cons a1
                                (cons (car opab) ; op
                                      (cons b1 (cons c1 (cdddr opab))))
                                (cdddr exp)))))))
   (cons (car exp) ; op
         (cons (rewrite (cadr exp)) ; a
               (cons (rewrite (caddr exp)) ; b
                     (cdddr exp))))))

Parameter binding times:
rewrite: exp : B
assocrw: exp : B
Specialisation points:
Call 4 in rewrite to rewrite
Call 3 in rewrite to rewrite
Call 5 in rewrite to rewrite
Call 2 in rewrite to rewrite

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
Call 6 in rewrite to rewrite
Call 1 in rewrite to rewrite

C.3.6  badd
(define (goal x y) (badd x y))
(define (badd x y)
  (if (equal? y '()) x (badd '(1) (badd x (cdr y))))))

Parameter binding times:
badd: x : B y : D
goal: x : B y : D

Specialisation points:
Call 1 in badd to badd
Call 2 in badd to badd

C.3.7  contrived-1
;; A contrived example from Arne Glenstrup's Master's Thesis
;; Numbers represented by list length
(define (contrived-1 a b) (f a (cons 1 (cons 1 a)) a b))
(define (f x y z d)
  (if (and (> z 'zero) (> d 'zero))
     (f (cons 1 x) z (cdr z) (cdr d))
     (if (> y 'zero) (g x (cdr y) d) x)))
(define (g u v w)
  (if (> u 'zero)
     (f (cons 1 u) (if (equal? (h v 'zero) 'zero) v (dec v)) (cdr v) (cdr w))
     u))
(define (h r s)
  (if (> s 'zero) (h (cdr r) 42)
     (if (> r 'zero) (h r (cdr s)) r)))
(define (dec n) (cdr n))

Parameter binding times:
dec: n : B
h: r : B s : B
g: u : B v : B w : D
f: x : B y : B z : B d : D
contrived-1: a : B b : D

Specialisation points:
None.

C.3.8  contrived-2
;; A contrived example from Arne Glenstrup's Master's Thesis
;; Numbers represented by list length
(define (contrived-2 a b) (f a (cons 1 (cons 1 a)) a b))
(define (f x y x d)
  (if (and (> x 'zero) (> d 'zero))
     (f (cons 1 x) z (cdr z) (cdr d))
     (if (> y 'zero) (g x (cdr y) d) x)))
(define (g u v w)
  (if (> u 'zero)
     (f (cons 1 u) (if (equal? (h v 'zero) 'zero) v (dec v)) (cdr v) (cdr w))
     u))
(define (h r s)
  (if (> s 'zero) (h (cdr r) 42)
     (if (> r 'zero) (h 17 (cdr s)) r)))
(define (dec n) (cdr n))

Parameter binding times:
dec: n : B
h: r : B s : B
g: u : B v : B w : D
f: x : B y : B z : B d : D
contrived-2: a : B b : D

Specialisation points:
Call 1 in h to h
C.3.9 decrease

(define (goal x) (decrease x))
(define (decrease x) (if (equal? x '()) 42 (decrease (cdr x))))

Parameter binding times:
- decrease: x : B
- goal: x : B
- Specialisation points: None.

C.3.10 deeprev

;;; Recursively reverse all list elements in a data structure
;;; Example: (deeprev '((1 2 3) 4 5 6 ((8 9 10 11) . 12))
(define (goal x) (deeprev x))
(define (deeprev x)
  (if (pair? x)
    (deeprevapp x '())
    x))

(define (deeprevapp xs rest)
  (if (pair? xs)
    (deeprevapp (cdr xs) (cons (deeprev (car xs)) rest))
    (if (equal? xs '())
      rest
      (revconsapp rest xs))))

Parameter binding times:
- revconsapp: xs : B r : B
- deeprevapp: xs : B rest : B
- deeprev: x : B
- goal: x : B
- Specialisation points: None.

C.3.11 disjconj

;;; Predicates for disjunctive and conjunctive terms p
(define (disjconj p) (disj? p))
(define (disj? p)
  (if (pair? p)
    (if (equal? 'Or (car p))
      (and (conj? (cadr p)) (disj? (cddr p)))
      (conj? p))
    (conj? p)))

(define (conj? p)
  (if (pair? p)
    (if (equal? 'And (car p))
      (and (disj? (cadr p)) (conj? (cddr p)))
      (bool? p))
    (bool? p)))

(define (bool? p) (or (equal? 'F p) (equal? 'T p)))

Parameter binding times:
- bool?: p : B
- conj?: p : B
- disj?: p : B
- disjconj: p : B
- Specialisation points: None.

C.3.12 duplicate

;;; Compute a list where each element is duplicated
(define (goal x) (duplicate x))
(define (duplicate xs)
  (if (equal? xs '())
    ()
    (cons (car xs) (cons (car xs) (duplicate (cdr xs))))))

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
Parameter binding times:
  duplicate: x : B
  goal: x : B
Specialisation points:
  None.

C.3.13  equal

(defun (equal x) (equal x))
(defun (equal x) (if (equal? x '()) 42 (equal x)))

Parameter binding times:
  equal: x : B
  goal: x : B
Specialisation points:
  Call 1 in equal to equal

C.3.14  evenodd

;;; Predicate: is x, unarily represented as '(: s s ... s), even/odd?
(defun (evenodd x) (even? x))
(defun (even? x) (if (null? x) #t (odd? (cdr x))))
(defun (odd? x) (if (pair? x) (even? (cdr x)) #f))

Parameter binding times:
  odd?: x : B
  even?: x : B
  evenodd: x : B
Specialisation points:
  None.

C.3.15  exponential

;;; These functions produce 2^n size-change graphs
(defun (f1 x1 y1)
  (if x1
      (f1 y1 x1)
      (f1 x1 y1)))
(defun (f2 x1 y1 x2 y2)
  (if x1
      (f2 y1 x1 x2 y2)
      (f2 x2 y2 x1 y1)))
(defun (f3 x1 y1 x2 y2 x3 y3)
  (if x1
      (f3 y1 x1 x2 y2 x3 y3)
      (f3 x3 y3 x1 y1 x2 y2)))
(defun (f4 x1 y1 x2 y2 x3 y3 x4 y4)
  (if x1
      (f4 y1 x1 x2 y2 x3 y3 x4 y4)
      (f4 x4 y4 x1 y1 x2 y2 x3 y3)))
(defun (f5 x1 y1 x2 y2 x3 y3 x4 x4 y4 x5 y6)
  (if x1
      (f5 y1 x1 x2 y2 x3 y3 x4 x4 y4 x5 y6)
      (f5 x6 y6 x1 y1 x2 y2 x3 y3 x4 x4 y4)))

Parameter binding times:
  f5: x1 : B y1 : B x2 : B y2 : B x3 : B y3 : B x4 : B y4 : B x5 : B y6 : B
  f4: x1 : B y1 : B x2 : B y2 : B x3 : B y3 : B x4 : B y4 : B
  f3: x1 : B y1 : B x2 : B y2 : B x3 : B y3 : B
  f2: x1 : B y1 : B x2 : B y2 : B
  f1: x1 : B y1 : B
Specialisation points:
  Call 2 in f1 to f1
  Call 1 in f2 to f2
  Call 1 in f3 to f3
  Call 1 in f4 to f4
  Call 1 in f5 to f5
  Call 1 in f1 to f1
  Call 2 in f2 to f2
  Call 1 in f3 to f3
  Call 2 in f4 to f4
  Call 2 in f5 to f5

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
C.3.16 fold

;; The fold operators, using a fixed operator, op
(define (fold a xs) (cons (foldl a xs) (cons (foldr a xs) '())))
(define (foldl a xs)
  (if (pair? xs)
      (foldl (op a (car xs)) (cdr xs))
      a))
(define (foldr a xs)
  (if (pair? xs)
      (op (car xs) (foldr a (cdr xs)))
      a))
(define (op x1 x2) (+ x1 x2))

Parameter binding times:
op: x1 : D x2 : D
foldr: a : D xs : B
foldl: a : D xs : B
fold: a : D xs : B
Specialisation points:
None.

C.3.17 game

;; The game function from Manuvir Das’ PhD Thesis (p. 137)
(define (goal p1 p2 moves) (game p1 p2 moves))
(define (game p1 p2 moves)
  (if (equal? moves '())
      (cons p1 p2)
      (if (equal? (car moves) 'swap)
          (game p2 p1 (cdr moves))
          (if (equal? (car moves) 'capture)
              (game (cons (car p2) p1) (cdr p2) (cdr moves))
              'error)))))

Parameter binding times:
game: p1 : B p2 : B moves : B
goal: p1 : B p2 : B moves : B
Specialisation points:
None.

C.3.18 increase

(define (goal x) (increase x))
(define (increase x) (if (equal? x '()) 42 (increase (cons 1 x))))

Parameter binding times:
increase: x : S
goal: x : B
Specialisation points:
Call 1 in increase to increase

C.3.19 intlookup

;; The function call case of an interpreter
;; Function number represented as list length
(define (run e p) (intlookup e p))
(define (intlookup e p)
  (if (equal? (fnum e) '())
      (car p)
      (intlookup (lookup e) p))
(define (lookup fnum p)
  (if (equal? fnum '(1) (car p) (lookup (cdr fnum) (cdr p)))))

Parameter binding times:
lookup: fnum : D p : B
intlookup: e : D p : B
run: e : D p : B
Specialisation points:
Call 1 in intlookup to intlookup
C.3.20 letexp

;; Testing the let construction
(define (goal x y) (letexp x y))
(define (letexp x y) (let* ((z (cons 1 x))) (letexp z y)))

Parameter binding times:
letexp: x : B y : B
goal: x : B y : B

Specialisation points:
Call 1 in letexp to letexp

C.3.21 list

;; The predicate for checking whether the argument is a list
(define (goal x) (list? x))
(define (list? xs) (if (pair? xs) (list? (cdr xs)) (null? xs)))

Parameter binding times:
list?: xs : B
goal: x : B

Specialisation points:
None.

C.3.22 lte

;; Less than or equal
(define (goal x y) (and (lte? x y) (even? x)))
(define (lte? x y)
  (if (null? x)
    #t
    (if (and (pair? x) (pair? y))
      (lte? (cdr x) (cdr y))
      #f)))
(define (even? x)
  (if (null? x)
    #t
    (if (null? (cdr x))
      #f
      (even? (cdr (cdr x))))))

Parameter binding times:
even?: x : B
lte?: x : B y : D
goal: x : B y : D

Specialisation points:
None.

C.3.23 map

;; The map function with fixed function f
(define (goal xs) (map xs))
(define (map xs)
  (if (equal? xs '())
    '()
    (cons (f (car xs)) (map (cdr xs)))))
(define (f x) (+ x x))

Parameter binding times:
f: x : B
map: xs : B
goal: xs : B

Specialisation points:
None.
### C.3.24  \textit{member}

\begin{verbatim}
;; The member function
(define (goal x xs) (member? x xs))
(define (member? x xs)
  (if (equal? xs '())
    #t
    (if (equal? x (car xs))
      #t
      (member? x (cdr xs))))
)
\end{verbatim}

Parameter binding times:
- \textit{member?}: \(x : B\) \(xs : B\)
- \textit{goal}: \(x : B\) \(xs : B\)

Specialisation points:
- None.

-----------------------------

Parameter binding times:
- \textit{member?}: \(x : B\) \(xs : D\)
- \textit{goal}: \(x : B\) \(xs : B\)

Specialisation points:
- Call 1 in \textit{member?} to \textit{member?}

### C.3.25  \textit{mergelists}

\begin{verbatim}
;;; Merge two lists
(define (goal xs ys) (merge xs ys))
(define (merge xs ys)
  (if (equal? xs '())
    ys
    (if (equal? ys '())
      xs
      (if (<= (car xs) (car ys))
        (cons (car xs) (merge (cdr xs) ys))
        (cons (car ys) (merge xs (cdr ys)))))))
\end{verbatim}

Parameter binding times:
- \textit{merge}: \(xs : B\) \(ys : D\)
- \textit{goal}: \(xs : B\) \(ys : D\)

Specialisation points:
- Call 2 in \textit{merge} to \textit{merge}

### C.3.26  \textit{mul}

\begin{verbatim}
;;; Unary multiplication and addition, e.g. (mul '(s s z) '(s s s z))
(define (goal x y) (mul x y))
(define (mul x y)
  (if (equal? x '())
    (add (mul (cdr x) y) y))
  (if (equal? y '())
    (add (mul x (cdr y)) (cons 's y)))
\end{verbatim}

Parameter binding times:
- \textit{add}: \(x : D\) \(y : D\)
- \textit{mul}: \(x : B\) \(y : D\)
- \textit{goal}: \(x : B\) \(y : D\)

Specialisation points:
- Call 1 in \textit{add} to \textit{add}

### C.3.27  \textit{naiverev}

\begin{verbatim}
;;; Naive reverse function
(define (goal xs) (naiverev xs))
(define (naiverev xs)
  (if (equal? xs '())
    xs
    (app (naiverev (cdr xs)) (cons (car xs) '()))))
\end{verbatim}

Parameter binding times:
- \textit{app}: \(xs : B\) \(ys : B\)
- \textit{naiverev}: \(xs : B\)
- \textit{goal}: \(xs : B\)

Specialisation points:
- None.

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
C.3.28  \textit{nestdec}

\begin{verbatim}
;; Parameter decrease by nested call in recursion
(define (goal x) (nestdec x))
(define (nestdec x) (if (equal? x '()) 17 (nestdec (dec x))))
\end{verbatim}

Parameter binding times:
\begin{itemize}
  \item dec: x : B
  \item nestdec: x : B
  \item goal: x : B
\end{itemize}

Specialisation points:
None.

C.3.29  \textit{nesteql}

\begin{verbatim}
;; Parameter equality by nested call in recursion
(define (goal x) (nesteql x))
(define (nesteql x) (if (equal? x '()) 17 (nesteql (eql x))))
\end{verbatim}

Parameter binding times:
\begin{itemize}
  \item eql: x : B
  \item nesteql: x : B
  \item goal: x : B
\end{itemize}

Specialisation points:
\begin{itemize}
  \item Call 1 in eql to eql
  \item Call 1 in nesteql to nesteql
\end{itemize}

C.3.30  \textit{nestimeql}

\begin{verbatim}
;; Using an immaterial "copy" as recursive argument
(define (goal x) (nestimeql x))
(define (nestimeql x)
  (if (equal? x '()) 42
   (nestimeql (immatcopy x))))
\end{verbatim}

Parameter binding times:
\begin{itemize}
  \item immatcopy: x : S
  \item nestimeql: x : S
  \item goal: x : B
\end{itemize}

Specialisation points:
\begin{itemize}
  \item Call 1 in immatcopy to immatcopy
  \item Call 1 in nestimeql to nestimeql
\end{itemize}

C.3.31  \textit{nestinc}

\begin{verbatim}
;; Parameter increase by nested call in recursion
(define (goal x) (nestinc x))
(define (nestinc x) (if (equal? x '()) 17 (nestinc (inc x))))
\end{verbatim}

Parameter binding times:
\begin{itemize}
  \item inc: x : S
  \item nestinc: x : S
  \item goal: x : B
\end{itemize}

Specialisation points:
\begin{itemize}
  \item Call 1 in inc to inc
  \item Call 1 in nestinc to nestinc
\end{itemize}

C.3.32  \textit{nolexicord}

\begin{verbatim}
;; Example not termination-provable by simple lexicographical ordering
(define (goal a1 b1 a2 b2 a3 b3) (nolexicord a1 b1 a2 b2 a3 b3))
\end{verbatim}

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
Parameter binding times:
nolexicord:  a1 : B  b1 : B  a2 : B  b2 : B  a3 : B  b3 : B
goal: a1 : B  b1 : B  a2 : B  b2 : B  a3 : B  b3 : B
Specialisation points:
None.

Parameter binding times:
nolexicord:  a1 : B  b1 : B  a2 : B  b2 : B  a3 : D  b3 : D
goal: a1 : B  b1 : B  a2 : B  b2 : B  a3 : B  b3 : D
Specialisation points:
None.

C.3.33 ordered
;; Predicate that checks whether a list is ordered
(define (goal? xs) (ordered? xs))
(define (ordered? xs)
  (if (pair? xs)
      (if (pair? (cdr xs))
          (if (<= (car xs) (cadr xs))
              (ordered? (cddr xs))
              #f)
        #t)
    #t))

Parameter binding times:
ordered?: xs : B
goal: xs : B
Specialisation points:
None.

C.3.34 overlap
;; Predicate for checking whether there is an overlap of two sets
(define (goal? xs ys) (overlap? xs ys))
(define (has-a-or-b? xs) (overlap? xs (cons 'a (cons 'b '()))))
(define (overlap? xs ys)
  (if (pair? xs)
      (if (member? (car xs) ys)
          #t
        (overlap? (cdr xs) ys))
    #f)

(define (member? x xs)
  (if (pair? xs)
      (if (equal? (car xs) x)
          #t
        (member? x (cdr xs)))
    #f))

Parameter binding times:
member?: x : B  xs : D
overlap?: xs : B  ys : D
goal: xs : B  ys : D
Specialisation points:
Call 1 in member? to member?

C.3.35 permute
;; Compute all the permutations of a list
(define (permute xs)
  (if (equal? xs '())
      '(())
    (select (car xs) (permute (revapp postfix))
      (select (car postfix) (permute (revapp postfix)))))

(define (revapp x xs)
  (for-each (lambda (y) (cons x y)) xs))

Parameter binding times:
permute?: xs : B  postfix: xs : D
permute: xs : B  postfix: xs : D

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
;; Map '(cons x' onto the list of lists xss and append the rest
(define (mapconsapp x xss rest)
  (if (equal? xss '())
    rest
    (cons (cons x (car xss)) (mapconsapp x (cdr xss) rest))))

;; Reverse xs and append the rest
(define (revapp xs rest)
  (if (equal? xs '())
    rest
    (revapp (cdr xs) (cons (car xs) rest))))

Parameter binding times:
revapp: xs : S rest : S
mapconsapp: x : S xss : S rest : S
select: x : S revprefix : S postfix : S
permute: xs : S
goal: x : B
Specialisation points:
Call 1 in mapconsapp to mapconsapp
Call 4 in select to select
Call 1 in revapp to revapp
Call 1 in permute to select

C.3.36 revapp

;;; Reverse list and append to rest
(define (goal x y) (revapp x y))
(define (revapp xs rest)
  (if (equal? xs '())
    rest
    (revapp (cdr xs) (cons (car xs) rest))))

Parameter binding times:
revapp: xs : B rest : B
goal: x : B y : D
Specialisation points: None.

C.3.37 select

;;; Compute a list of lists. Each list is computed by picking out an
;;; element of the original list and consing it onto the rest of the list
(define (select xs)
  (if (equal? xs '())
    '()
    (selects(car xs)'() (cdr xss))))
(define (selects x revprefix postfix)
  (cons (cons x (revapp revprefix postfix))
    (if (equal? postfix '())
      ()
      (selects (car postfix) (cons x revprefix) (cdr postfix))))))

;;; Reverse xs and append to rest
(define (revapp xs rest)
  (if (equal? xs '())
    rest
    (revapp (cdr xs) (cons (car xs) rest))))

Parameter binding times:
revapp: xs : B rest : B
selects: x : B revprefix : B postfix : B
Specialisation points: None.

C.3.38 shuffle

;;; Shuffle List
(define (goal xs) (shuffle xs))
(define (shuffle xs)
  (if (equal? xs '())
    '()
    (cons (car xs) (shuffle (reverse (cdr xs)))))
(define (reverse xs)
  (if (equal? xs '())
    xs
    (append (reverse (cdr xs)) (cons (car xs) '()))))
(define (append xs ys)
  (if (equal? xs '())
    ys
    (cons (car xs) (append (cdr xs) ys))))

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
A. Glenstrup and N. Jones

Parameter binding times:
append: xs : S ys : S
reverse: xs : S
shuffle: xs : S
goal: xs : B

Specialisation points:
Call 2 in reverse to reverse
Call 1 in append to append
Call 1 in shuffle to shuffle

C.3.39 sp1
;; Mutual recursion requiring specialisation points
(define (sp1 x y) (f x y))
(define (f x y) (if (equal? x '()) (g x y) (h x y)))
(define (g x y) (if (equal? x '()) (h x y) (r x y)))
(define (h x y) (if (equal? x '()) (h x y) (f x y)))
(define (r x y) x)

Parameter binding times:
r: x : B y : D
h: x : B y : D
g: x : B y : D
f: x : B y : D
sp1: x : B y : D

Specialisation points:
Call 1 in h to h
Call 2 in h to f

C.3.40 subsets
;; Compute all subsets
(define (goal xs) (subsets xs))
(define (subsets xs)
  (if (pair? xs)
    (let* ((subs (subsets (cdr xs))))
      (mapconsapp (car xs) subs subs))
    '(())))

;; map '(cons x' ont the list of lists xss, and append rest
(define (mapconsapp x xss rest)
  (if (pair? xss)
    (cons (cons x (car xss)) (mapconsapp x (cdr xss) rest))
    rest))

Parameter binding times:
mapconsapp: x : B xss : B rest : B
subsets: xs : B
goal: xs : B

Specialisation points:
None.

C.3.41 thetrick
;; The trick: pulling out the conditional into the context
(define (goal x y) (cons (f x y) (cons (g x y) '(())))
(define (f x y)
  (if (null? y) 42
    (f (if (null? x) x (cdr x)) ; 1
     (if (null? x) (cdr y) (cons 1 y)))
     (g x y)
(defined (g x y)
  (if (null? y) 42
    (if (null? x)
      (g x (cdr y)) ; 1
      (g (cdr x) (cons 1 y)))))

Parameter binding times:
g: x : B y : D
f: x : B y : D
goal: x : B y : D

Specialisation points:
Call 1 in g to g
Call 1 in f to f

Parameter binding times:
g: x : D y : B
f: x : D y : D

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
Termination Analysis and Specialization-Point Insertion in Partial Evaluation

Parameter binding times:
\[
\begin{align*}
\text{t:} & \quad x : B \quad y : B \\
\text{r:} & \quad x : B \quad y : B \\
\text{p:} & \quad x : B \quad y : B \\
\text{q:} & \quad x : B \quad y : B \\
\text{e:} & \quad a : B \quad b : B \\
\text{goal:} & \quad x : B \quad y : B 
\end{align*}
\]

Specialisation points:
\[
\begin{align*}
\text{Call 1 in g to g} \\
\text{Call 2 in g to g} \\
\text{Call 1 in f to f} \\
\text{Call 3 in p to p} \\
\text{Call 3 in q to q} \\
\text{Call 3 in r to r} \\
\text{Call 2 in p to r}
\end{align*}
\]

C.3.42 \textit{vangelder}

Following is an example due to Allen Van Gelder.

Note that in the following example there is a cycle involving \( q, p, r, t, \) and \( q \) again, such that nothing gets smaller along that cycle.

\[
\begin{align*}
\text{e(a,b)} & \\
\text{q(X,Y)} & \leftarrow e(X,Y) \\
\text{q(X,f(f(X)))} & \leftarrow p(X,f(f(X))), q(X,f(X)) \\
\text{q(X,f(f(X)))} & \leftarrow p(X,f(Y)) \\
\text{p(X,Y)} & \leftarrow e(X,Y), p(X,Y) \\
\text{p(X,f(Y))} & \leftarrow r(X,f(Y)), p(X,Y) \\
\text{r(X,Y)} & \leftarrow e(X,Y), r(X,Y) \\
\text{r(X,f(Y))} & \leftarrow q(X,Y), r(X,Y) \\
\text{r(f(X),f(X))} & \leftarrow t(f(X),f(X)) \\
\text{r(f(X),f(Y))} & \leftarrow q(f(X),f(Y)), t(X,Y) \\
\text{p(X,Y)} & \leftarrow e(X,Y), p(X,Y) \\
\text{p(X,f(Y))} & \leftarrow r(X,f(Y)), p(X,Y) \\
\text{r(X,Y)} & \leftarrow e(X,Y), r(X,Y) \\
\text{r(X,f(Y))} & \leftarrow q(X,Y), r(X,Y) \\
\text{r(f(X),f(X))} & \leftarrow t(f(X),f(X)) \\
\text{r(f(X),f(Y))} & \leftarrow q(f(X),f(Y)), t(X,Y) \\
\text{p(X,Y)} & \leftarrow e(X,Y), p(X,Y) \\
\text{p(X,f(Y))} & \leftarrow r(X,f(Y)), p(X,Y) \\
\end{align*}
\]

\begin{verbatim}
(define (goal x y) (q x y))
(define (e a b) (and (equal? a 'a) (equal? b 'b)))
(define (q x y)  
  (if (e x y) #t  
    (if (and (pair? y) (equal? (car y) 'f)  
      (pair? (cdr y))) (equal? (cadr y) 'f))  
    (if (and (p x y) (q x (cdr y))) #t  
      (p x (cdr y))))  
  #f))
(define (p x y)  
  (if (e x y) #t  
    (if (equal? 'f (car y))  
      (and (r x y) (p x (cdr y))))  
    #f))
(define (r x y)  
  (if (e x y) #t  
    (if (and (pair? y) (equal? (car y) 'f))  
      (if (and (q x (cdr y)) (r x (cdr y))) #t  
        (if (and (pair? x) (equal? (car x) 'f))  
          (t x y)  
          #f))  
      #f))
  #f))
(define (t x y)  
  (if (e x y) #t  
    (if (and (pair? x) (equal? (car x) 'f))  
      (pair? y) (equal? (car y) 'f))  
    (and (q x y) (t (cdr x) (cdr y))))  
  #f))
\end{verbatim}

Parameter binding times:
\[
\begin{align*}
\text{t:} & \quad x : B \quad y : D \\
\text{r:} & \quad x : B \quad y : B \\
\text{p:} & \quad x : B \quad y : D \\
\text{q:} & \quad x : B \quad y : D \\
\text{e:} & \quad a : B \quad b : B \\
\text{goal:} & \quad x : B \quad y : D 
\end{align*}
\]

C.4 Sorting functions

C.4.1 \textit{mergesort}

\begin{verbatim}
;; Mergesort
(define (goal xs) (mergesort xs))
(define (mergesort xs)
\end{verbatim}
(if (pair? xs)
  (if (pair? (cdr xs))
    (splitmerge xs '() '())
    xs)
  xs))

(define (splitmerge xs xs1 xs2)
  (if (pair? xs)
    (splitmerge (cdr xs) (cons (car xs) xs2) xs1)
    (merge (mergesort xs1) (mergesort xs2))))

(define (merge xs1 xs2)
  (if (pair? xs1)
    (if (pair? xs2)
      (if (<= (car xs1) (car xs2))
        (cons (car xs1) (merge (cdr xs1) xs2))
        (cons (car xs2) (merge xs1 (cdr xs2))))
      xs1)
    xs2))

Parameter binding times:
merge: xs1 : S xs2 : S
splitmerge: xs : S xs1 : S xs2 : S
mergesort: xs : S
goal: xs : B

Specialisation points:
Call 2 in merge to merge
Call 1 in merge to merge
Call 1 in splitmerge to splitmerge
Call 1 in mergesort to splitmerge

C.4.2 minsort
;; Minimum sort: remove minimum and cons it onto the rest, sorted.
(define (goal xs) (minsort xs))
(define (minsort xs)
  (if (pair? xs)
    (appmin (car xs) (cdr xs) xs)
    '()))

(define (appmin min rest xs)
  (if (pair? rest)
    (if (< (car rest) min)
      (appmin (car rest) (cdr rest) xs)
      (appmin min (cdr rest) xs))
    (cons min (minsort (remove min xs))))))

(define (remove x xs)
  (if (pair? xs)
    (if (equal? x (car xs))
      (cdr xs)
      (cons (car xs) (remove x (cdr xs))))
    '()))

Parameter binding times:
remove: x : S xs : S
appmin: min : S rest : S xs : S
mergesort: xs : S
goal: xs : B

Specialisation points:
Call 2 in appmin to appmin
Call 1 in appmin to appmin
Call 1 in remove to remove
Call 1 in minsort to appmin

C.4.3 quicksort
;; Quicksort
(define (goal xs) (quicksort xs))
(define (quicksort xs)
  (if (pair? xs)
    (if (pair? (cdr xs))
      (part (car xs) xs (cons (car xs) '() '())
              (part x (cdr xs) (cons (car xs) xs2))
    (part x (cdr xs) (cons (car xs) xs2))

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005.
(if (< x (car xs))
  (part x (cdr xs) xs1 (cons (car xs) xs2))
  (part x (cdr xs) xs1 xs2)))
(app (quicksort xs1) (quicksort xs2)))

(define (app xs ys)
  (if (pair? xs)
      (cons (car xs) (app (cdr xs) ys))
      ys))

Parameter binding times:
  app: xs : S ys : S
  part: x : S xs : S xs1 : S xs2 : S
  quicksort: xs : S
  goal: xs : B

Specialisation points:
  Call 3 in part to part
  Call 1 in app to app
  Call 1 in part to part
  Call 2 in part to part
  Call 1 in quicksort to part