

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

Termination Analysis and Specialization-Point Insertion in Off-line Partial Evaluation

ARNE JOHN GLENSTRUP
IT University of Copenhagen
and
NEIL D. JONES
DIKU, University of Copenhagen

ACM Transactions on Programming Languages and Systems, Vol. 27, No. 6, November 2005, Pages 101–168.

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, *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 *nestimeql* shows one shortcoming of the size approximation function \mathcal{E}^1 : for the function

```
immatcopy x = if x = [] then []
              else cons (car x) (immatcopy (cdr x))
```

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

The remaining examples are handled without resulting in overconservative re-

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2005 ACM 0164-0925/2005/0500-0101 \$5.00

<i>Program</i>	<i>Goal BT</i>	<i>Result</i>	<i>Optimal result</i>	<i>Conservative</i>	<i>Program description</i>
int-loop	s s d s d d	SP	SP	no no	interpret a LOOP program: $\text{int}(prog, bound, input)$
int-while-d	s d	SP, G	SP, G	no	interpret a WHILE program with dynamic scoping
int-while-s	s d	SP	SP	no	interpret a WHILE program, cf. Figure 2
lambdaint	s	SP, G	SP, G	no	interpret a λ expression
parsexp	s			no	parse a numerical expression
turing	s d	SP	SP	no	run a program on a Turing machine: $\text{turing}(prog, tape)$
ack	s d	SP	SP	no	compute Ackermann's function: $\text{ack}(m, n)$
binom	s d d s	SP	SP	no no	compute the binomial function using unary numbers
gcd-1	s d	SP	SP	no	compute greatest common divisor
gcd-2	s d	SP	SP	no	compute greatest common divisor, swapping arguments
graphcol-1	d s s d	SP SP	SP SP	no no	colour a graph with colours from a set (using The Trick)
graphcol-2	d s s d	SP SP	SP SP	no no	colour a graph with colours from a set (tail recursive, using The Trick)
graphcol-3	d s s d	SP SP	SP SP	no no	colour a graph with colours from a set (tail recursive, without The Trick)
match	s d d s	SP	SP	no no	match naïvely sublist pattern p in a symbol list s
power	d s s d	SP SP	SP SP	no no	compute the power function: x^a
reach	d d s d s s s d s	SP SP SP	SP SP SP	no no no	compute a path from node x to node y in graph G
rematch	s d d s	SP, G SP	SP, G SP	no no	match regular expression r in a string s
strmatch	s d d s	SP, G	SP, G	no no	match naïvely string pattern p in a string s
typeof	s s			no	infer a type in the typed lambda calculus
add	s d d s	SP, G	SP, G	no no	add two unary numbers
addlists	s d			no	add two lists elementwise
anchored	s d d s	SP, G	SP, G	no no	function where y is anchored in x
append	s d			no	append two lists: $\text{append}(list1, list2)$
assocrw	s	SP, G		yes	make an associative expression right-recursive
badd	s d	SP	SP	no	add numbers using a quasiterminating addition function
contrived-1	s d			no	exercise the analyser with a small contrived program
contrived-2	s d	SP	SP	no	insert a specialisation point in a static function
decrease	s			no	recursive calls with a decreasing argument
deeprev	s			no	recursively reverse all list elements in a data structure
disjconj	s			no	check for disjunctive and conjunctive terms
duplicate	s			no	build a copy of a list
equal	s	SP	SP	no	recursive calls with a constant argument
evenodd	s			no	check whether a unary number is even or odd
exponential	s s	SP	SP	no	functions generating exponentially many SCGs

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.

<i>Program</i>	<i>Goal BT</i>	<i>Result</i>	<i>Optimal result</i>	<i>Conservative</i>	<i>Program description</i>
fold	d s s d	SP	SP	no no	fold a fixed operator over a list: fold(<i>elt</i> , <i>list</i>)
game	s s s			no	play a small game with two players
increase	s	SP, G	SP, G	no	recursive calls with increasing argument
intlookup	d s	SP	SP	no	look up variable values like in an interpreter
letexp	s s	SP, G	SP, G	no	example using the let construction
list	s			no	check for a list data structure
lte	s d d s	SP	SP	no no	check whether <i>x</i> is a substructure of <i>y</i>
map	s			no	map a fixed function along a list
member	d s s d	SP	SP	no no	check for list membership: member(<i>element</i> , <i>list</i>)
mergelists	s d	SP	SP	no	merge two sorted lists
mul	s d	SP	SP	no	multiply two unary numbers
naiverev	s			no	naïve reverse: append reversed tail to head element
nestdec	s			no	nested call returns cdr of its argument
nesteq1	s	SP	SP	no	nested call returns a copy of its argument
nestimeq1	s	SP, G	SP	yes	nested call to a function that constructs a copy of its argument only from constants
nestinc	s	SP, G	SP, G	no	nested call returns a cons cell containing its argument
nolexicord	s s s s s s s s s s s d			no no	a terminating function with no lexicographical ordering, cf. Example 9.2.1
ordered	s			no	check whether a list is ordered
overlap	s d	SP	SP	no	check for non-empty set intersection
permute	s	SP, G	SP	yes	compute all the permutations of a list
revapp	s d			no	reverse <i>list1</i> and append to <i>list2</i> : revapp(<i>list1</i> , <i>list2</i>)
select	s			no	pick out an element and cons it onto the remaining list
shuffle	s	SP, G		yes	shuffle a list
sp1	s d	SP	SP	no	mutual recursion requiring specialisation points
subsets	s			no	compute all subsets of a set
thetrick	s d d s	SP SP, G	SP SP, G	no no	example using the trick for dynamic if conditionals
vangelder	s d	SP	SP	no	quasiterminating example invented by Van Gelder
mergesort	s	SP, G		yes	sort list by splitting, recursive sorting, and merging
minsort	s	SP, G		yes	sort list: extract min elt, cons it onto the sorted rest
quicksort	s	SP, G		yes	sort list by splitting by size, sorting and appending

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

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

sults, notably including several interpreters.

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 *int-loop*

```

;;; 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)))
    (eval ef (cons nf '()) (cons input '()) l p)))

(define (eval e ns vs l p)
  (if (equal? (car e) 1) ; constants
      (cdr e)
      (if (equal? (car e) 2) ; variable
          (lookvar (cdr e) ns vs) ; 1
          (if (equal? (car e) 3) ; basefcn
              (let* ((v1 (eval (car (cdr (cdr e))) ns vs l p)) ; 2
                     (v2 (eval (car (cdr (cdr (cdr e)))) ns vs l p))) ; 3
                (apply (car (cdr e)) v1 v2)) ; 4
              (if (equal? (car e) 4) ; if
                  (if (equal? (eval (car (cdr e)) ns vs l p) 'T) ; 5
                      (eval (car (cdr (cdr e))) ns vs l p) ; 6
                      (eval (car (cdr (cdr (cdr e)))) ns vs l p)) ; 7
                  (if (equal? (car e) 5) ; ==
                      (if (equal? (eval (car (cdr e)) ns vs l p) ; 8
                                  (eval (car (cdr (cdr e))) ns vs l p)) ; 9
                          'T
                          'F)
                      ; call
                      (let* ((ef (lookbody (car (cdr e)) p)) ; 10
                             (nf (lookname (car (cdr e)) p)) ; 11
                             (v (eval (car (cdr (cdr e))) ns vs l p))) ; 12
                        (if (equal? l '()) '() ; 13
                            (eval 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
      (cons v1 v2)))

```

```

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
eval:    e : B ns : B vs : D l : 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
eval:    e : B ns : B vs : D l : D p : B
run:     p : B l : D input : D
Specialisation points:
Call 13 in eval to eval

```

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) (cdr exp))
      ((LET LET*)
       (let* ((value
               (evalexp (car (cdaadr exp)) names values program))) ; 1
              (evalexp (caddr exp)
                       (cons (caaaadr exp) names) ; 2
                       (cons value values)
                       program))))
      ((IF)
       (if (evalexp (cadr exp) names values program) ; 3
           (evalexp (caddr exp) names values program) ; 4
           (evalexp (caddr exp) names values program))) ; 5
      (else
       (if (function? (car exp) program) ; 6
           (evalexp (lookup-body (car exp) program) ; 7, 8
                   (append ;; DYNAMIC SCOPING
                          (lookup-paramnames (car exp) program) ; 9
                          names) ;; DYNAMIC SCOPING
                   (append ;; DYNAMIC SCOPING
                          (argvals (cdr exp) names values program) ; 10
                          values) ;; DYNAMIC SCOPING
                   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)) ; 13
   (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

```

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 (caaaadr exp) names)
                       (cons value values)
                       program))))
      ((IF)
       (if (evalexp (cadr exp) names values program) ; 3
           (evalexp (caddr exp) names values program) ; 4
           (evalexp (caddr exp) names values program))) ; 5
      (else
       (if (function? (car exp) program) ; 6
           (evalexp (lookup-body (car exp) program) ; 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)) ; 13
   (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: varname : B names : B values : D
variable?:    varname : B names : B
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 7 in evalexp to evalexp

```

C.1.4 *lambdaint*

```

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

```

```

;;; R [[\n.e]] = (2 R [[n]] R [[e]])
;;; R [[e e']] = (3 R [[e]] R [[e']])
(define (lambdaint e) (red e))
(define (red e) ; reduce lambda expression e
  (if (isvar? e)
      e
      (if (islam? e)
          e
          (let* ((f (red (app->e1 e))) (a (red (app->e2 e))))
              (if (islam? f)
                  (red (subst (lam->var f) a (lam->body f)))
                  (mkapp f a)))))))

(define (subst x a e)
  (if (isvar? e)
      (if (equal? x e) a e)
      (if (islam? e)
          (if (equal? x (lam->var e))
              e
              (mklam (lam->var e) (subst x a (lam->body e))))
          (mkapp (subst x a (app->e1 e)) (subst x a (app->e2 e))))))

(define (isvar? e) (equal? (car e) 1))
(define (islam? e) (equal? (car e) 2))

(define (mklam n e) (cons 2 (cons n (cons e '()))))
(define (lam->var e) (cadr e))
(define (lam->body e) (caddr e))
(define (mkapp e1 e2) (cons 3 (cons e1 (cons e2 '()))))
(define (app->e1 e) (cadr e))
(define (app->e2 e) (caddr e))

```

Parameter binding times:

```

app->e2: e : S
app->e1: e : S
mkapp:  e1 : S e2 : S
lam->body: e : S
lam->var: e : S
mklam:  n : S e : S
islam?: e : S
isvar?: e : S
subst:  x : S a : S e : S
red:    e : S
lambdaint: e : B

```

Specialisation points:

```

Call 6 in subst to subst
Call 8 in red to red
Call 9 in subst to subst
Call 5 in red to red
Call 11 in subst to subst
Call 3 in red to red

```

C.1.5 *parsexp*

```

;;; Parse a list of atoms as an expression. Return remaining list.
;;; e.g. '( "5" "*" "(" "3" "+" "2" "*" "4" ")" )
(define (parsexp xs) (expr xs))
(define (expr xs)
  (let* ((rs1 (term xs))
        (if (equal? '() rs1)
            rs1
            (if (member? (car rs1) '("+ "-"))
                (let* ((rs2 (expr (cdr rs1)))
                      (if (equal? '() rs2) rs1 rs2))
                    rs1))))))

(define (term xs)
  (let* ((rs1 (factor xs))
        (if (equal? rs1 '())
            rs1
            (if (member? (car rs1) '(" "/"))
                (let* ((rs2 (term (cdr rs1)))
                      (if (equal? rs2 (cdr rs1)) rs1 rs2))
                    rs1))))))

(define (factor xs)
  (if (equal? "(" (car xs))
      (let* ((rs1 (expr (cdr xs)))
            (if (and (not (equal? rs1 '()))
                    (not (equal? rs1 (cdr xs)))
                    (equal? ")" (car rs1)))
                (cdr rs1)
            (cdr rs1))

```

```

      xs))
    (atom xs)))
(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
 factor: 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)
                       revltape (cons (cdar instrs) (cdr rtape)) prog) ; 1
              (if (equal? 'Left (caar instrs))
                  (if (pair? revltape)
                      (turing (cdr instrs)
                              (cdr revltape)
                              (cons (car revltape) rtape) prog) ; 2
                      (turing (cdr instrs)
                              '()
                              (cons 'Blank rtape) prog)) ; 3
                  (if (equal? 'Right (caar instrs))
                      (if (pair? rtape)
                          (turing (cdr instrs)
                                  (cons (car rtape) revltape)
                                  (cdr rtape) prog) ; 4
                          (turing (cdr instrs)
                                  (cons 'Blank revltape)
                                  '() prog)) ; 5
                      (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
                                      (lookup (cddar instrs) prog) revltape rtape prog) ; 8
                                      (turing (cdr instrs) revltape rtape prog)) ; 9
                              #f) ; 10
                              )
                          )
                      )
                  )
              )
          )
      )
      rtape
      )))
(define (lookup i instrs)
  (if (= i 1) instrs (lookup (- i 1) (cdr instrs))))
Parameter binding times:
lookup: i : B instrs : B
turing: instrs : B revltape : D rtape : D prog : B
run: prog : B tapeinput : B

```

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 *gcd-2*

```

;;; Greatest common divisor, numbers represented by list length
;;; x and y are swapped in the recursive call
(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 y (monus x y)) (gcd (monus y x) 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 2 in monus to monus
Call 1 in lgth to lgth
Call 1 in gt to gt
Call 4 in gcd to gcd
Call 2 in gcd to gcd

```

C.2.5 *graphcolour-1*

```

;;; Colour graph G with colours cs so that neighbors have different colours
;;; 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) '()) '())
                          (cdr 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))
          '()))

    ;; 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
              (possible color (cdr adjs) colorednodes))
          #t))

    ;; Return colour of node. If no colour yet, return nil.
    (define (colorof node colorednodes)
      (if (pair? colorednodes)
          (if (equal? (caddr 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)
          (let* ((colorednode (colornode ncs (car rest) colorednodes)))
              (if (pair? colorednode)
                  (colorrest cs ncs colorednodes (cdr rest))
                  nil))
          nil))

```

```

      (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 rest)
                '()))
            '()))
      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 adjs : 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 2 in colornode to colornode
Call 1 in revapp to revapp
Call 3 in colorrest to colorrest-thetrick
-----

```

```

Parameter binding times:
revapp:          xs : D rest : D
reverse:         xs : D
colorrest-thetrick: cs1 : B cs : B ncs : 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 : 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) '()) '())
                    (cdr 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))
          '()))
  )

```

```

;;; 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
          (possible color (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-thetrick
                       cs cs (cdr ncs) ; try next colour
                       colorednodes rest)
                      '()))
              (colornoderest cs (cdr ncs) node colorednodes rest))
          '()))
      (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
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-thetrick to colorrest-thetrick
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
reverse:     xs : D
colorrest-thetrick: 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:

```

```

Call 1 in colorof to colorof
Call 2 in possible to possible
Call 1 in revapp to revapp
Call 1 in colorrest to colornoderest

```

C.2.7 *graphcolour-3*

```

;;; 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) '()) '())
                          (cdr 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))
          '()))

    ;; Can we use color with these adjacent nodes and current coloured nodes?
    (define (possible color adjcs colorednodes)
      (if (pair? adjcs)
          (if (equal? color (colorof (car adjcs) colorednodes))
              #f
              (possible color (cdr adjcs) 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
                           cs (cdr ncs) ; try next colour
                           colorednodes rest)
                          '()))
                  (colornoderest cs (cdr ncs) node 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 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
reverse:     xs : 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:
Call 1 in colorof to colorof
Call 2 in possible to possible
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
-----

```

```

Parameter binding times:
loop:  p : D s : B pp : D ss : B
match: p : D s : B
Specialisation points:
None.

```

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 '()) '() (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
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 2 in power to power
-----

```

```

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

```

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 '())
      '()
      (if (equal? u (caar rest))
          (let* ((path (reach (cdar rest) v edges))
                 (if (equal? path '())
                     (via u v (cdr rest) edges)
                     (cons (car rest) path))))
              (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 : B 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 : B v : B edges : B

Specialisation points:
 Call 2 in reach to via

Parameter binding times:
 member?: x : D xs : B
 via: u : D v : D rest : B edges : B
 reach: u : D v : D edges : B
 goal: u : B 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)
  ; if str matches patstr, match returns a pair consisting of
  ; the prefix of str which matches the pattern and
  ; the remaining part of str, else match returns #f
  (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) '() '()))
  (define (parsep patchars seq stack)
    (if (pair? patchars)
        (if (equal? #\. (car patchars))
            (parsep-dot patchars seq stack)
            (if (equal? #\* (car patchars))
                (parsep-star patchars seq stack)
```

```

      (if (equal? #\<( (car patchars))
              (parsep-openb patchars seq stack)
          (if (equal? #\<( (car patchars))
                  (parsep-closeb patchars seq stack)
              (if (equal? #\<( (car patchars))
                      (parsep-char (cdr patchars) seq stack)
                  (parsep-char patchars seq stack))))))
          ; else (pair? patchars)
          (if (pair? stack)
              (error "unmatched '( ' in pattern")
              (cons 'seq (reverse seq))))))

(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) '()))
             (cdr 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 '() (cdr pat))
                      (error "unknown pattern data " pat))))))
      (cons '() cs))

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

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

(define (domatch-star cs pat init)
  ; init holds the chars already star-matched
  (if (pair? cs)
      (let* ((first (domatch pat cs)))
          (if (pair? first)
              (domatch-star (cdr first) pat (append (car first) init))
              first))))

```

```

      (cons init cs)))
    (cons init cs)))
(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))
                        ; first match was too long, try matching fewer chars
                        (if (pair? (car first))
                            (domatch-seq
                             (reverse (cdr first))
                             (cons (caar first) (append (cdr first) rest))
                             pats)
                            'nomatch))) ; even shortest possible first match
                                ; (empty string) doesn't lead to match
                        'nomatch)) ; first match failed
                    (cons '() (append cs rest)))) ; no patterns left to
                    ; match: success!
      '()
      (cons '() (append cs rest))))

```

Parameter binding times:

```

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

```

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
parsepat:     patstr : 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 patcs cs n)

```

```

(if (pair? cs)
  (if (prefix patcs cs) ; 1
      (cons n (domatch patcs (cdr cs) (+ n 1))) ; 2
      (domatch patcs (cdr cs) (+ n 1))) ; 3
    (if (equal? patcs cs) (cons n '()) '()))))
(define (prefix prec cs)
  (or (null? prec)
      (and (pair? cs) (and (equal? (car prec) (car cs))
                           (prefix (cdr prec) (cdr cs))))))

```

Parameter binding times:
 prefix: prec : 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: prec : 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))
        (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 t (cadr t1)) (tsubst a t (caddr t1))))
              (error 'tsubst-t1))))

    (define (subst venv a t) ; substitute t for occurrences of type
      (if (pair? venv) ; var a in the variable environment
          (cons (cons (caar venv) (tsubst a t (cdar venv)))
                (subst (cdr venv) a t))
          '()))

    (define (unify venv t1 t2) ; unify types t1 and t2, returning
      ; (<new venv> . <unified type>)
      (if (equal? 'Tvar (car t1))
          (cons (subst venv (cdr t1) t2) t2)
          (if (equal? 'Arr (car t1))
              (if (equal? 'Tvar (car t2))
                  (cons (subst venv (cdr t2) t1) t1)
                  (if (equal? 'Arr (car t2))
                      (let* ((venv1tx1 (unify venv (cadr t1) (caddr t1)))
                            (venv2tx2 (unify (car venv1tx1) (cadr t2) (caddr t2))))
                        (cons (car venv2tx2)
                              (cons 'Arr (cons (cdr venv1tx1) (cdr venv2tx2))))))
                      (error 'unify-t2)))
              (error 'unify-t1))))

```

```

(define (etype venv tenv e t) ; infer type of e, unified with type t
  (if (equal? 'Var (car e)) ; using variable environment
      ; and tyvar generator
      (let* ((venv1t1 (unify venv (vtype venv (cdr e)) t)))
            (cons (cdr venv1t1) (cons (car venv1t1) tenv)))
      (if (equal? 'App (car e))
          (let* ((atenv1 (freshtvar tenv))
                 (t2venv2tenv2
                  (etype venv (cdr atenv1) (cadr e) (car atenv1)))
                 (t1venv3tenv3
                  (etype (cadr t2venv2tenv2)
                         (caddr t2venv2tenv2)
                         (caddr e)
                         (cons 'Arr (cons (car t2venv2tenv2) t))))
                 (t1 (car t1venv3tenv3)))
                ; t1 == ('Arr . (? . t'))
                (cons (caddr t1) (cdr t1venv3tenv3)))
          (if (equal? 'Lam (car e))
              (let* ((atenv1 (freshtvar tenv))
                     (t1venv2tenv2
                      (etype (cons (cons (cadr e) (car atenv1)) venv)
                              (cdr atenv1) (caddr e) (car atenv1)))
                     (venv3t3
                      (unify (cadr t1venv2tenv2)
                             (cons 'Arr
                                    (cons (vtype (caddr t1venv2tenv2) (cadr e))
                                            (car t1venv2tenv2))))
                      t)))
                    (cons (cdr venv3t3) (cons (cadr venv3t3) (caddr t1venv2tenv2))))
              (error 'Error-in-lambda-expression e))))))

```

```

Parameter binding times:
etype:   venv : B tenv : B e : B t : B
unify:   venv : B t1 : B t2 : B
subst:   venv : B a : B t : B
tsubst:  a : B t : B t1 : 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))
;;; -----
;;; a != 'op a -> a1  b -> b1          a != 'op ...
;;; 'op a b -> 'op a1 b1          a -> a
```

```
(define (assocrw exp) (rewrite exp))
(define (rewrite exp)
  (if (and (pair? exp)
          (equal? 'op (car exp)))
      (let* ((opab (cadr exp))
             (if (and (pair? opab)
                     (equal? '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 (caddr opab))))
                             (caddr exp))))))
                 (cons (car exp) ; op
                       (cons
                        (rewrite (cadr exp)) ; a
                        (cons
                         (rewrite (caddr exp)) ; b
                         (caddr exp))))))
      exp))
```

```
Parameter binding times:
  rewrite: exp : S
  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
```

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 (> w 'zero)
      (f (cons 1 u)
          (if (equal? (h v 'zero) 'zero) v (dec v))
          (cdr v)
          (cdr w))
      u))
(define (h r s)
  (if (> r 'zero)
      (h (cdr r) 42)
      (if (> s '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 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 (> w 'zero)
      (f (cons 1 u)
          (if (equal? (h v 'zero) 'zero) v (dec v))
          (cdr v)
          (cdr w))
      u))
(define (h r s)
  (if (> r 'zero)
      (h (cdr r) 42)
      (if (> s '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))
;; ==> ((3 2 1) 4 5 6 ((11 10 9) 8) . 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))))
(define (revconsapp xs r)
  (if (pair? xs) (revconsapp (cdr xs) (cons (car xs) r)) r))
```

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? (caddr p)))
          (conj? p))
      (conj? p)))
(define (conj? p)
  (if (pair? p)
      (if (equal? 'And (car p))
          (and (disj? (cadr p)) (conj? (caddr 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))))))
```

```

Parameter binding times:
  duplicate: xs : B
  goal:      x  : B
Specialisation points:
None.

```

C.3.13 *equal*

```

(define (goal x) (equal x))
(define (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 ... s), even/odd?
(define (evenodd x) (even? x))
(define (even? x) (if (null? x) #t (odd? (cdr x))))
(define (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 n*2^n size-change graphs

```

```

(define (f1 x1 y1)
  (if x1
      (f1 y1 x1)
      (f1 x1 y1)))

(define (f2 x1 y1 x2 y2)
  (if x1
      (f2 y1 x1 x2 y2)
      (f2 x2 y2 x1 y1)))

(define (f3 x1 y1 x2 y2 x3 y3)
  (if x1
      (f3 y1 x1 x2 y2 x3 y3)
      (f3 x3 y3 x1 y1 x2 y2)))

(define (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)))

(define (f5 x1 y1 x2 y2 x3 y3 x4 y4 x5 y5)
  (if x1
      (f5 y1 x1 x2 y2 x3 y3 x4 y4 x5 y5)
      (f5 x5 y5 x1 y1 x2 y2 x3 y3 x4 y4)))

```

```

Parameter binding times:
f5:  x1 : B y1 : B x2 : B y2 : B x3 : B y3 : B x4 : B y4 : B x5 : B y5 : 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 f3 to f3
Call 1 in f4 to f4
Call 1 in f2 to f2
Call 1 in f5 to f5
Call 1 in f1 to f1
Call 2 in f2 to f2
Call 2 in f3 to f3
Call 2 in f4 to f4
Call 2 in f5 to f5

```

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.

Parameter binding times:

```
op:    x1 : D x2 : D
foldr: a : B xs : D
foldl: a : D xs : D
fold:  a : B xs : D
```

Specialisation points:

```
Call 2 in foldr to foldr
Call 1 in foldl to foldl
```

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) (intlookup (lookup e p) p))
(define (lookup fnum p)
  (if (equal? fnum '()) (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 : S 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.

Parameter binding times:
 even?: x : D
 lte?: x : D y : B
 goal: x : D y : B
 Specialisation points:
 Call 1 in even? to even?

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 *member*

```
;; The member function
(define (goal x xs) (member? x xs))
(define (member? x xs)
  (if (equal? xs '())
      (if (equal? x (car xs))
          #t
          (member? x (cdr xs)))
      #f))
```

Parameter binding times:
 member?: x : D xs : B
 goal: x : B xs : B
 Specialisation points:
 None.

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

C.3.25 *mergelists*

```
;; 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)))))))
```

Parameter binding times:
 merge: xs : B ys : D
 goal: xs : B ys : D
 Specialisation points:
 Call 2 in merge to merge

C.3.26 *mul*

```
;; 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)))
(define (add x y) (if (equal? x '()) y (add (cdr x) (cons 's y))))
```

Parameter binding times:
 add: x : D y : D
 mul: x : B y : D
 goal: x : B y : D
 Specialisation points:
 Call 1 in add to add

C.3.27 *naiverev*

```
;; Naive reverse function
(define (goal xs) (naiverev xs))
(define (naiverev xs)
  (if (equal? xs '())
      xs
      (app (naiverev (cdr xs)) (cons (car xs) '()))))
(define (app xs ys)
  (if (equal? xs '()) ys (cons (car xs) (app (cdr xs) ys))))
```

Parameter binding times:
 app: xs : B ys : B
 naiverev: xs : B
 goal: xs : B
 Specialisation points:
 None.

C.3.28 *nestdec*

```
;; Parameter decrease by nested call in recursion
(define (goal x) (nestdec x))
(define (nestdec x) (if (equal? x '()) 17 (nestdec (dec x))))
(define (dec x) (if (equal? x '(1)) (cdr x) (dec (cdr x))))
```

Parameter binding times:

```
dec:    x : B
nestdec: x : B
goal:   x : B
```

Specialisation points:

None.

C.3.29 *nesteq1*

```
;; Parameter equality by nested call in recursion
(define (goal x) (nesteq1 x))
(define (nesteq1 x) (if (equal? x '()) 17 (nesteq1 (eql x))))
(define (eql x) (if (equal? x '()) x (eql x)))
```

Parameter binding times:

```
eql:    x : B
nesteq1: x : B
goal:   x : B
```

Specialisation points:

```
Call 1 in eql to eql
Call 1 in nesteq1 to nesteq1
```

C.3.30 *nestimeql*

```
;; Using an immaterial "copy" as recursive argument
(define (goal x) (nestimeql x))
(define (nestimeql x)
  (if (equal? x '()) 42 (nestimeql (immatcopy x))))
(define (immatcopy x)
  (if (equal? x '()) '() (cons '0 (immatcopy (cdr x)))))
```

Parameter binding times:

```
immatcopy: x : S
nestimeql: x : S
goal:      x : B
```

Specialisation points:

```
Call 1 in immatcopy to immatcopy
Call 1 in nestimeql to nestimeql
```

C.3.31 *nestinc*

```
;; Parameter increase by nested call in recursion
(define (goal x) (nestinc x))
(define (nestinc x) (if (equal? x '()) 17 (nestinc (inc x))))
(define (inc x) (if (equal? x '()) '(1) (cons 1 (inc (cdr x)))))
```

Parameter binding times:

```
inc:    x : S
nestinc: x : S
goal:   x : B
```

Specialisation points:

```
Call 1 in inc to inc
Call 1 in nestinc to nestinc
```

C.3.32 *nolexicord*

```
;; Example not termination-provable by simple lexicographical ordering
(define (goal a1 b1 a2 b2 a3 b3) (nolexicord a1 b1 a2 b2 a3 b3))
(define (nolexicord a1 b1 a2 b2 a3 b3)
  (if (equal? a1 '())
      42
      (if (equal? a1 b1)
          (nolexicord
            (cdr b1) (cdr a1) (cdr a2) (cdr b2) (cdr b3) (cdr a3))
          (nolexicord
            (cdr b1) (cdr a1) (cdr b2) (cdr a2) (cdr a3) (cdr b3)))))
```

```

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 (goal xs) (permute xs))
(define (permute xs)
  (if (equal? xs '())
      '()
      (select (car xs) '() (cdr xs))))
;; Select x as the first element and cons it onto
;; permutations of the remaining list represented by
;; the list of elements before x (reversed) and the
;; list of elements after x. Finally, recurse by moving
;; on to the next element in postfix
(define (select x revprefix postfix)
  (mapconsapp x (permute (revapp revprefix postfix))
              (if (equal? postfix '())
                  '()
                  (select (car postfix)
                          (cons x revprefix)
                          (cdr postfix))))))

```

```

;; 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:     xs : 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 : D
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 xs))))

(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
select:   xs : 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))))

```

```

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))
        '(cons (cons (car xs) (mapconsapp x (cdr xs) rest))
              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)) (cdr y) (cons 1 y)))) ; 1
(define (g x y)
  (if (null? y) 42
      (g x (cdr y) (cons 1 y)))) ; 2

```

```

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 : S
  f:    x : D  y : D

```

```

goal:   x : D y : B
Specialisation points:
Call 1 in g to g
Call 2 in g to g
Call 1 in f to f

```

C.3.42 *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.

```

```

;;; e(a,b).
;;; q(X,Y)      :- e(X,Y).
;;; q(X,f(f(X))) :- p(X,f(f(X))), q(X,f(X)).
;;; q(X,f(f(Y))) :- p(X,f(Y)).
;;;
;;; p(X,Y)      :- e(X,Y).
;;; p(X,f(Y))   :- r(X,f(Y)), p(X,Y).
;;;
;;; r(X,Y)      :- e(X,Y).
;;; r(X,f(Y))   :- q(X,Y), r(X,Y).
;;; r(f(X),f(X)) :- t(f(X),f(X)).
;;;
;;; t(X,Y)      :- e(X,Y).
;;; t(f(X),f(Y)) :- q(f(X),f(Y)), t(X,Y).

(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)))
(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)))

```

Parameter binding times:

```

t:   x : B y : D
r:   x : B y : D
p:   x : B y : D
q:   x : B y : D
e:   a : B b : D
goal: x : B y : D
Specialisation points:
Call 3 in p to p
Call 3 in q to q
Call 3 in r to r
Call 2 in p to r

```

C.4 Sorting functions

C.4.1 *mergesort*

```

;; Mergesort
(define (goal xs) (mergesort xs))
(define (mergesort xs)

```

```

(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
minsort:    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) '()) '())
          xs)
      xs))

(define (part x xs xs1 xs2)
  (if (pair? xs)
      (if (> x (car xs))
          (part x (cdr xs) (cons (car xs) xs1) xs2)
          (part x (cdr xs) (cons (car xs) xs1) xs2)

```

```
(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
```