

Hoare Logic

Various

April 19, 2009

Abstract

These theories contain a Hoare logic for a simple imperative programming language with while-loops, including a verification condition generator.

Special infrastructure for modelling and reasoning about pointer programs is provided, together with many examples, including Schorr-Waite. See [1, 2] for an excellent exposition.

Contents

0.0.1	Derivation of the proof rules and, most importantly, the VCG tactic	17
0.0.2	References	19
0.0.3	Field access and update	19
0.1	The heap	19
0.1.1	Paths in the heap	19
0.1.2	Lists on the heap	20
0.1.3	Functional abstraction	21
0.2	Verifications	22
0.2.1	List reversal	22
0.2.2	Searching in a list	23
0.2.3	Merging two lists	24
0.2.4	Storage allocation	26
0.2.5	References	27
0.3	The heap	27
0.3.1	Paths in the heap	27
0.3.2	Non-repeating paths	28
0.3.3	Lists on the heap	28
0.3.4	Functional abstraction	29
0.3.5	Field access and update	30
0.4	Verifications	31
0.4.1	List reversal	31
0.4.2	Searching in a list	33
0.4.3	Splicing two lists	34
0.4.4	Merging two lists	35
0.4.5	Cyclic list reversal	39
0.4.6	Storage allocation	40
0.4.7	Field access and update	41
0.5	Verifications	42
0.5.1	List reversal	42
0.6	Machinery for the Schorr-Waite proof	42
0.7	The Schorr-Waite algorithm	46
0.7.1	Paths in the heap	54
0.7.2	Lists on the heap	55

```

theory Hoare
imports Main
uses (hoare-tac.ML)
begin

types
  'a bexp = 'a set
  'a assn = 'a set

datatype
  'a com = Basic 'a  $\Rightarrow$  'a
    | Seq 'a com 'a com ((-;/ -) [61,60] 60)
    | Cond 'a bexp 'a com 'a com ((1IF -/ THEN - / ELSE -/ FI) [0,0,0] 61)
    | While 'a bexp 'a assn 'a com ((1WHILE -/ INV {-} // DO - / OD) [0,0,0]
61)

syntax
  @assign :: id  $\Rightarrow$  'b  $\Rightarrow$  'a com ((2- :=/ -) [70,65] 61)
  @annskip :: 'a com (SKIP)

translations
  SKIP == Basic id

types 'a sem = 'a  $\Rightarrow$  'a  $\Rightarrow$  bool

consts iter :: nat  $\Rightarrow$  'a bexp  $\Rightarrow$  'a sem  $\Rightarrow$  'a sem
primrec
  iter 0 b S = (%s s'. s  $\sim$ : b & (s=s'))
  iter (Suc n) b S = (%s s'. s : b & (? s''. S s s'' & iter n b S s'' s'))

consts Sem :: 'a com  $\Rightarrow$  'a sem
primrec
  Sem(Basic f) s s' = (s' = f s)
  Sem(c1;c2) s s' = (? s''. Sem c1 s s'' & Sem c2 s'' s')
  Sem(IF b THEN c1 ELSE c2 FI) s s' = ((s : b  $\longrightarrow$  Sem c1 s s') &
    (s  $\sim$ : b  $\longrightarrow$  Sem c2 s s'))
  Sem(While b x c) s s' = (? n. iter n b (Sem c) s s')

constdefs Valid :: 'a bexp  $\Rightarrow$  'a com  $\Rightarrow$  'a bexp  $\Rightarrow$  bool
  Valid p c q == !s s'. Sem c s s'  $\longrightarrow$  s : p  $\longrightarrow$  s' : q

syntax
  @hoare-vars :: [idts, 'a assn, 'a com, 'a assn]  $\Rightarrow$  bool
    (VARs -// {-} // - // {-} [0,0,55,0] 50)
syntax ( output)
  @hoare :: ['a assn, 'a com, 'a assn]  $\Rightarrow$  bool

```

$(\{-\} // - // \{-\} [0,55,0] 50)$

ML⟨⟨

local

```

fun abs((a,T),body) =
  let val a = absfree(a, dummyT, body)
  in if T = Bound 0 then a else Const(Syntax.constrainAbsC,dummyT) $ a $ T
end
in

fun mk-abstuple [x] body = abs (x, body)
  | mk-abstuple (x::xs) body =
    Syntax.const split $ abs (x, mk-abstuple xs body);

fun mk-fbody a e [x as (b,-)] = if a=b then e else Syntax.free b
  | mk-fbody a e ((b,-)::xs) =
    Syntax.const Pair $ (if a=b then e else Syntax.free b) $ mk-fbody a e xs;

fun mk-fexp a e xs = mk-abstuple xs (mk-fbody a e xs)
end
⟩⟩

```

ML⟨⟨

```

fun bexp-tr (Const (TRUE, -)) xs = Syntax.const TRUE
  | bexp-tr b xs = Syntax.const Collect $ mk-abstuple xs b;

fun assn-tr r xs = Syntax.const Collect $ mk-abstuple xs r;
⟩⟩

```

ML⟨⟨

```

fun com-tr (Const(@assign,-) $ Free (a,-) $ e) xs =
  Syntax.const Basic $ mk-fexp a e xs
  | com-tr (Const (Basic,-) $ f) xs = Syntax.const Basic $ f
  | com-tr (Const (Seq,-) $ c1 $ c2) xs =
    Syntax.const Seq $ com-tr c1 xs $ com-tr c2 xs
  | com-tr (Const (Cond,-) $ b $ c1 $ c2) xs =
    Syntax.const Cond $ bexp-tr b xs $ com-tr c1 xs $ com-tr c2 xs
  | com-tr (Const (While,-) $ b $ I $ c) xs =
    Syntax.const While $ bexp-tr b xs $ assn-tr I xs $ com-tr c xs
  | com-tr t - = t (* if t is just a Free/Var *)
⟩⟩

```

ML⟨
local

fun *var-tr* (*Free* (*a*, -)) = (*a*, *Bound 0*) (* *Bound 0* = dummy term *)
 | *var-tr* (*Const* (-*constrain*, -) \$ (*Free* (*a*, -)) \$ *T*) = (*a*, *T*);

fun *vars-tr* (*Const* (-*idts*, -) \$ *idt* \$ *vars*) = *var-tr idt* :: *vars-tr vars*
 | *vars-tr t* = [*var-tr t*]

in
fun *hoare-vars-tr* [*vars*, *pre*, *prg*, *post*] =
 let val xs = *vars-tr vars*
 in Syntax.const Valid \$
 assn-tr pre xs \$ *com-tr prg xs* \$ *assn-tr post xs*
 end
 | *hoare-vars-tr ts* = *raise TERM* (*hoare-vars-tr*, *ts*);
end
 ⟩

parse-translation ⟨ [(@*hoare-vars*, *hoare-vars-tr*)] ⟩

ML⟨

fun *dest-abstuple* (*Const* (*split*, -) \$ (*Abs* (*v*, -, *body*))) =
 subst-bound (*Syntax.free v*, *dest-abstuple body*)
 | *dest-abstuple* (*Abs* (*v*, -, *body*)) = *subst-bound* (*Syntax.free v*, *body*)
 | *dest-abstuple trm* = *trm*;

fun *abs2list* (*Const* (*split*, -) \$ (*Abs* (*x*, *T*, *t*))) = *Free* (*x*, *T*) :: *abs2list t*
 | *abs2list* (*Abs* (*x*, *T*, *t*)) = [*Free* (*x*, *T*)]
 | *abs2list* - = [];

fun *mk-ts* (*Const* (*split*, -) \$ (*Abs* (*x*, -, *t*))) = *mk-ts t*
 | *mk-ts* (*Abs* (*x*, -, *t*)) = *mk-ts t*
 | *mk-ts* (*Const* (*Pair*, -) \$ *a* \$ *b*) = *a* :: (*mk-ts b*)
 | *mk-ts t* = [*t*];

fun *mk-vts* (*Const* (*split*, -) \$ (*Abs* (*x*, -, *t*))) =
 (*Syntax.free x*) :: (*abs2list t*, *mk-ts t*)
 | *mk-vts* (*Abs* (*x*, -, *t*)) = ([*Syntax.free x*], [*t*])
 | *mk-vts t* = *raise Match*;

fun *find-ch* [] *i xs* = (*false*, (*Syntax.free not-ch*, *Syntax.free not-ch*))
 | *find-ch* ((*v*, *t*) :: *vts*) *i xs* = *if* *t* = (*Bound i*) *then find-ch vts* (*i-1*) *xs*
 else (*true*, (*v*, *subst-bounds* (*xs*, *t*)));

```

fun is-f (Const (split,-) $ (Abs(x,-,t))) = true
  | is-f (Abs(x,-,t)) = true
  | is-f t = false;
>>

```

```

ML<<
fun assn-tr' (Const (Collect,-) $ T) = dest-abstuple T
  | assn-tr' (Const (@{const-name Set.Int}, -) $ (Const (Collect,-) $ T1) $
    (Const (Collect,-) $ T2)) =
    Syntax.const Set.Int $ dest-abstuple T1 $ dest-abstuple T2
  | assn-tr' t = t;

fun bexp-tr' (Const (Collect,-) $ T) = dest-abstuple T
  | bexp-tr' t = t;
>>

```

```

ML<<
fun mk-assign f =
  let val (vs, ts) = mk-vts f;
      val (ch, which) = find-ch (vs~~ts) ((length vs)-1) (rev vs)
  in if ch then Syntax.const @assign $ fst(which) $ snd(which)
    else Syntax.const @skip end;

fun com-tr' (Const (Basic,-) $ f) = if is-f f then mk-assign f
  else Syntax.const Basic $ f
  | com-tr' (Const (Seq,-) $ c1 $ c2) = Syntax.const Seq $
    com-tr' c1 $ com-tr' c2
  | com-tr' (Const (Cond,-) $ b $ c1 $ c2) = Syntax.const Cond $
    bexp-tr' b $ com-tr' c1 $ com-tr' c2
  | com-tr' (Const (While,-) $ b $ I $ c) = Syntax.const While $
    bexp-tr' b $ assn-tr' I $ com-tr' c
  | com-tr' t = t;

fun spec-tr' [p, c, q] =
  Syntax.const @hoare $ assn-tr' p $ com-tr' c $ assn-tr' q
>>

```

print-translation << [(Valid, spec-tr')] >>

lemma SkipRule: $p \subseteq q \implies \text{Valid } p \text{ (Basic id) } q$
by (auto simp: Valid-def)

lemma BasicRule: $p \subseteq \{s. f \ s \in q\} \implies \text{Valid } p \text{ (Basic f) } q$
by (auto simp: Valid-def)

lemma SeqRule: $\text{Valid } P \ c1 \ Q \implies \text{Valid } Q \ c2 \ R \implies \text{Valid } P \ (c1;c2) \ R$

by (*auto simp: Valid-def*)

lemma *CondRule*:

$p \subseteq \{s. (s \in b \longrightarrow s \in w) \wedge (s \notin b \longrightarrow s \in w')\}$
 $\implies \text{Valid } w \ c1 \ q \implies \text{Valid } w' \ c2 \ q \implies \text{Valid } p \ (\text{Cond } b \ c1 \ c2) \ q$
by (*auto simp: Valid-def*)

lemma *iter-aux*: $! s \ s'. \text{Sem } c \ s \ s' \dashrightarrow s : I \ \& \ s : b \dashrightarrow s' : I \implies$
 $(\bigwedge s \ s'. s : I \implies \text{iter } n \ b \ (\text{Sem } c) \ s \ s' \implies s' : I \ \& \ s' \sim b)$

apply (*induct n*)

apply *clarsimp*

apply (*simp (no-asm-use)*)

apply *blast*

done

lemma *WhileRule*:

$p \subseteq i \implies \text{Valid } (i \cap b) \ c \ i \implies i \cap (-b) \subseteq q \implies \text{Valid } p \ (\text{While } b \ i \ c) \ q$

apply (*clarsimp simp: Valid-def*)

apply (*drule iter-aux*)

prefer 2 **apply** *assumption*

apply *blast*

apply *blast*

done

lemma *Compl-Collect*: $\neg(\text{Collect } b) = \{x. \sim(b \ x)\}$

by *blast*

lemmas *AbortRule* = *SkipRule* — dummy version

use *hoare-tac.ML*

method-setup *vcg* = $\langle\langle$

Scan.succeed (*fn* *ctxt* => *SIMPLE-METHOD'* (*hoare-tac* *ctxt* (*K all-tac*))) $\rangle\rangle$

verification condition generator

method-setup *vcg-simp* = $\langle\langle$

Scan.succeed (*fn* *ctxt* =>

SIMPLE-METHOD' (*hoare-tac* *ctxt* (*asm-full-simp-tac* (*local-simpset-of* *ctxt*))))

$\rangle\rangle$

verification condition generator plus simplification

end

theory *Arith2*

imports *Main*

begin

constdefs

$cd :: [nat, nat, nat] \Rightarrow bool$
 $cd\ x\ m\ n == x\ dvd\ m \ \&\ x\ dvd\ n$

$gcd :: [nat, nat] \Rightarrow nat$
 $gcd\ m\ n == @x.(cd\ x\ m\ n) \ \&\ (!y.(cd\ y\ m\ n) \dashrightarrow y \leq x)$

consts $fac :: nat \Rightarrow nat$

primrec

$fac\ 0 = Suc\ 0$
 $fac(Suc\ n) = (Suc\ n) * fac(n)$

cd

lemma $cd\text{-}nnn: 0 < n \Rightarrow cd\ n\ n\ n$
apply (*simp add: cd-def*)
done

lemma $cd\text{-}le: [| cd\ x\ m\ n; 0 < m; 0 < n |] \Rightarrow x \leq m \ \&\ x \leq n$
apply (*unfold cd-def*)
apply (*blast intro: dvd-imp-le*)
done

lemma $cd\text{-}swap: cd\ x\ m\ n = cd\ x\ n\ m$
apply (*unfold cd-def*)
apply *blast*
done

lemma $cd\text{-}diff\text{-}l: n \leq m \Rightarrow cd\ x\ m\ n = cd\ x\ (m-n)\ n$
apply (*unfold cd-def*)
apply (*fastsimp dest: dvd-diffD*)
done

lemma $cd\text{-}diff\text{-}r: m \leq n \Rightarrow cd\ x\ m\ n = cd\ x\ m\ (n-m)$
apply (*unfold cd-def*)
apply (*fastsimp dest: dvd-diffD*)
done

gcd

lemma $gcd\text{-}nnn: 0 < n \Rightarrow n = gcd\ n\ n$
apply (*unfold gcd-def*)
apply (*frule cd-nnn*)
apply (*rule some-equality [symmetric]*)
apply (*blast dest: cd-le*)
apply (*blast intro: le-anti-sym dest: cd-le*)
done

lemma $gcd\text{-}swap: gcd\ m\ n = gcd\ n\ m$

```

apply (simp add: gcd-def cd-swap)
done

lemma gcd-diff-l:  $n \leq m \implies \text{gcd } m \ n = \text{gcd } (m-n) \ n$ 
apply (unfold gcd-def)
apply (subgoal-tac  $n \leq m \implies !x. \text{cd } x \ m \ n = \text{cd } x \ (m-n) \ n$ )
apply simp
apply (rule allI)
apply (erule cd-diff-l)
done

lemma gcd-diff-r:  $m \leq n \implies \text{gcd } m \ n = \text{gcd } m \ (n-m)$ 
apply (unfold gcd-def)
apply (subgoal-tac  $m \leq n \implies !x. \text{cd } x \ m \ n = \text{cd } x \ m \ (n-m)$ )
apply simp
apply (rule allI)
apply (erule cd-diff-r)
done

pow

lemma sq-pow-div2 [simp]:
   $m \bmod 2 = 0 \implies ((n::\text{nat}) * n)^{(m \text{ div } 2)} = n^m$ 
apply (simp add: power2-eq-square [symmetric] power-mult [symmetric] mult-div-cancel)
done

end

theory Examples imports Hoare Arith2 begin

lemma multiply-by-add: VARS  $m \ s \ a \ b$ 
   $\{a=A \ \& \ b=B\}$ 
   $m := 0; \ s := 0;$ 
  WHILE  $m \neq a$ 
  INV  $\{s=m*b \ \& \ a=A \ \& \ b=B\}$ 
  DO  $s := s+b; \ m := m+(1::\text{nat})$  OD
   $\{s = A*B\}$ 
by vcg-simp

lemma VARS  $M \ N \ P :: \text{int}$ 
   $\{m=M \ \& \ n=N\}$ 
  IF  $M < 0$  THEN  $M := -M; \ N := -N$  ELSE SKIP FI;
   $P := 0;$ 

```

```

  WHILE  $0 < M$ 
  INV  $\{0 \leq M \ \& \ (EX \ p. \ p = (if \ m < 0 \ then \ -m \ else \ m) \ \& \ p * N = m * n \ \& \ P = (p - M) * N)\}$ 
  DO  $P := P + N; \ M := M - 1$  OD
   $\{P = m * n\}$ 
apply vcg-simp
  apply (simp add:int-distrib)
apply clarsimp
apply(rule conjI)
  apply clarsimp
apply clarsimp
done

```

```

lemma Euclid-GCD: VARS  $a \ b$ 
 $\{0 < A \ \& \ 0 < B\}$ 
 $a := A; \ b := B;$ 
  WHILE  $a \neq b$ 
  INV  $\{0 < a \ \& \ 0 < b \ \& \ gcd \ A \ B = gcd \ a \ b\}$ 
  DO IF  $a < b$  THEN  $b := b - a$  ELSE  $a := a - b$  FI OD
   $\{a = gcd \ A \ B\}$ 
apply vcg

```

```

  apply auto
  apply(simp add: gcd-diff-r less-imp-le)
  apply(simp add: linorder-not-less gcd-diff-l)
apply(erule gcd-nnn)
done

```

```

lemmas distrib =
  diff-mult-distrib diff-mult-distrib2 add-mult-distrib add-mult-distrib2

```

```

lemma gcd-scm: VARS  $a \ b \ x \ y$ 
 $\{0 < A \ \& \ 0 < B \ \& \ a = A \ \& \ b = B \ \& \ x = B \ \& \ y = A\}$ 
  WHILE  $a \sim b$ 
  INV  $\{0 < a \ \& \ 0 < b \ \& \ gcd \ A \ B = gcd \ a \ b \ \& \ 2 * A * B = a * x + b * y\}$ 
  DO IF  $a < b$  THEN ( $b := b - a; \ x := x + y$ ) ELSE ( $a := a - b; \ y := y + x$ ) FI OD
   $\{a = gcd \ A \ B \ \& \ 2 * A * B = a * (x + y)\}$ 
apply vcg
  apply simp
  apply(simp add: distrib gcd-diff-r linorder-not-less gcd-diff-l)
apply(simp add: distrib gcd-nnn)
done

```

```

lemma power-by-mult: VARS  $a\ b\ c$ 
  { $a=A \ \& \ b=B$ }
   $c := (1::nat)$ ;
  WHILE  $b \sim = 0$ 
  INV { $A \wedge B = c * a^b$ }
  DO WHILE  $b \bmod 2 = 0$ 
    INV { $A \wedge B = c * a^b$ }
    DO  $a := a*a; b := b \operatorname{div} 2$  OD;
     $c := c*a; b := b - 1$ 
  OD
  { $c = A^B$ }
apply vcg-simp
apply(case-tac  $b$ )
apply(simp add: mod-less)
apply simp
done

```

```

lemma factorial: VARS  $a\ b$ 
  { $a=A$ }
   $b := 1$ ;
  WHILE  $a \sim = 0$ 
  INV { $\operatorname{fac} A = b * \operatorname{fac} a$ }
  DO  $b := b*a; a := a - 1$  OD
  { $b = \operatorname{fac} A$ }
apply vcg-simp
apply(clarsimp split: nat-diff-split)
done

```

```

lemma [simp]:  $1 \leq i \implies \operatorname{fac} (i - \operatorname{Suc} 0) * i = \operatorname{fac} i$ 
by(induct i, simp-all)

```

```

lemma VARS  $i\ f$ 
  {True}
   $i := (1::nat); f := 1$ ;
  WHILE  $i \leq n$  INV { $f = \operatorname{fac}(i - 1) \ \& \ 1 \leq i \ \& \ i \leq n+1$ }
  DO  $f := f*i; i := i+1$  OD
  { $f = \operatorname{fac} n$ }
apply vcg-simp
apply(subgoal-tac i = Suc n)
apply simp
apply arith
done

```

```

lemma sqrt: VARs r x
{True}
x := X; r := (0::nat);
WHILE (r+1)*(r+1) <= x
INV {r*r <= x & x=X}
DO r := r+1 OD
{r*r <= X & X < (r+1)*(r+1)}
apply vcg-simp
done

```

```

lemma sqrt-without-multiplication: VARs u w r x
{True}
x := X; u := 1; w := 1; r := (0::nat);
WHILE w <= x
INV {u = r+r+1 & w = (r+1)*(r+1) & r*r <= x & x=X}
DO r := r + 1; w := w + u + 2; u := u + 2 OD
{r*r <= X & X < (r+1)*(r+1)}
apply vcg-simp
done

```

```

lemma imperative-reverse: VARs y x
{x=X}
y:=[];
WHILE x ~ = []
INV {rev(x)@y = rev(X)}
DO y := (hd x # y); x := tl x OD
{y=rev(X)}
apply vcg-simp
apply(simp add: neq-Nil-conv)
apply auto
done

```

```

lemma imperative-append: VARs x y
{x=X & y=Y}
x := rev(x);
WHILE x ~ = []
INV {rev(x)@y = X@Y}
DO y := (hd x # y);
    x := tl x
OD
{y = X@Y}
apply vcg-simp
apply(simp add: neq-Nil-conv)

```

apply *auto*
done

lemma *zero-search*: *VARs A i*
 $\{True\}$
 $i := 0;$
 $WHILE\ i < length\ A \ \&\ A!i \sim = key$
 $INV\ \{!j. j < i \dashv\rightarrow A!j \sim = key\}$
 $DO\ i := i+1\ OD$
 $\{(i < length\ A \dashv\rightarrow A!i = key) \ \&$
 $(i = length\ A \dashv\rightarrow (!j. j < length\ A \dashv\rightarrow A!j \sim = key))\}$
apply *vcg-simp*
apply(*blast elim!:* *less-SucE*)
done

lemma *lem*: $m - Suc\ 0 < n \implies m < Suc\ n$
by *arith*

lemma *Partition*:
 $[\![\ leq == \%A\ i. !k. k < i \dashv\rightarrow A!k \leq pivot;$
 $\quad geq == \%A\ i. !k. i < k \ \&\ k < length\ A \dashv\rightarrow pivot \leq A!k\]\!] \implies$
 $VARs\ A\ u\ l$
 $\{0 < length(A::('a::order)list)\}$
 $l := 0; u := length\ A - Suc\ 0;$
 $WHILE\ l \leq u$
 $INV\ \{leq\ A\ l \ \&\ geq\ A\ u \ \&\ u < length\ A \ \&\ l \leq length\ A\}$
 $DO\ WHILE\ l < length\ A \ \&\ A!l \leq pivot$
 $INV\ \{leq\ A\ l \ \&\ geq\ A\ u \ \&\ u < length\ A \ \&\ l \leq length\ A\}$
 $DO\ l := l+1\ OD;$
 $WHILE\ 0 < u \ \&\ pivot \leq A!u$
 $INV\ \{leq\ A\ l \ \&\ geq\ A\ u \ \&\ u < length\ A \ \&\ l \leq length\ A\}$
 $DO\ u := u - 1\ OD;$
 $IF\ l \leq u\ THEN\ A := A[l := A!u, u := A!l]\ ELSE\ SKIP\ FI$
 OD
 $\{leq\ A\ u \ \&\ (!k. u < k \ \&\ k < l \dashv\rightarrow A!k = pivot) \ \&\ geq\ A\ l\}$

apply (*simp*)
apply (*erule thin-rl*)+
apply *vcg-simp*
apply (*force simp: neq-Nil-conv*)
apply (*blast elim!:* *less-SucE intro: Suc-leI*)
apply (*blast elim!:* *less-SucE intro: less-imp-diff-less dest: lem*)
apply (*force simp: nth-list-update*)

done

end

theory *HoareAbort*
imports *Main*
uses (*hoare-tac.ML*)
begin

types

$'a \text{ bexp} = 'a \text{ set}$
 $'a \text{ assn} = 'a \text{ set}$

datatype

$'a \text{ com} = \text{Basic } 'a \Rightarrow 'a$
| *Abort*
| *Seq* $'a \text{ com } 'a \text{ com} \quad ((-;/ -) \quad [61,60] \ 60)$
| *Cond* $'a \text{ bexp } 'a \text{ com } 'a \text{ com} \quad ((1\text{IF } -/ \text{ THEN } -/ \text{ ELSE } -/ \text{ FI}) \ [0,0,0] \ 61)$
| *While* $'a \text{ bexp } 'a \text{ assn } 'a \text{ com} \quad ((1\text{WHILE } -/ \text{ INV } \{-\} // \text{DO } -/ \text{ OD}) \ [0,0,0] \ 61)$

syntax

$@\text{assign} :: id \Rightarrow 'b \Rightarrow 'a \text{ com} \quad ((2- :=/ -) \ [70,65] \ 61)$
 $@\text{annskip} :: 'a \text{ com} \quad (\text{SKIP})$

translations

$\text{SKIP} == \text{Basic } id$

types $'a \text{ sem} = 'a \text{ option} \Rightarrow 'a \text{ option} \Rightarrow \text{bool}$

consts $\text{iter} :: \text{nat} \Rightarrow 'a \text{ bexp} \Rightarrow 'a \text{ sem} \Rightarrow 'a \text{ sem}$

primrec

$\text{iter } 0 \ b \ S = (\lambda s \ s'. \ s \notin \text{Some } 'b \wedge s = s')$
 $\text{iter } (\text{Suc } n) \ b \ S =$
 $(\lambda s \ s'. \ s \in \text{Some } 'b \wedge (\exists s''. \ S \ s \ s'' \wedge \text{iter } n \ b \ S \ s'' \ s'))$

consts $\text{Sem} :: 'a \text{ com} \Rightarrow 'a \text{ sem}$

primrec

$\text{Sem}(\text{Basic } f) \ s \ s' = (\text{case } s \text{ of } \text{None} \Rightarrow s' = \text{None} \mid \text{Some } t \Rightarrow s' = \text{Some}(f \ t))$
 $\text{Sem } \text{Abort} \ s \ s' = (s' = \text{None})$
 $\text{Sem}(c1; c2) \ s \ s' = (\exists s''. \ \text{Sem } c1 \ s \ s'' \wedge \text{Sem } c2 \ s'' \ s')$
 $\text{Sem}(\text{IF } b \ \text{THEN } c1 \ \text{ELSE } c2 \ \text{FI}) \ s \ s' =$
 $(\text{case } s \text{ of } \text{None} \Rightarrow s' = \text{None}$
| $\text{Some } t \Rightarrow ((t \in b \longrightarrow \text{Sem } c1 \ s \ s') \wedge (t \notin b \longrightarrow \text{Sem } c2 \ s \ s')))$
 $\text{Sem}(\text{While } b \ x \ c) \ s \ s' =$
 $(\text{if } s = \text{None} \text{ then } s' = \text{None} \text{ else } \exists n. \ \text{iter } n \ b \ (\text{Sem } c) \ s \ s')$

constdefs $\text{Valid} :: 'a \text{ bexp} \Rightarrow 'a \text{ com} \Rightarrow 'a \text{ bexp} \Rightarrow \text{bool}$

$Valid\ p\ c\ q == \forall s\ s'. Sem\ c\ s\ s' \longrightarrow s : Some\ 'p \longrightarrow s' : Some\ 'q$

syntax

$@hoare\ vars :: [idts, 'a\ assn, 'a\ com, 'a\ assn] => bool$
 $(VARs\ -//\ \{-\}\ /\ -\ /\ \{-\}\ [0,0,55,0]\ 50)$

syntax (output)

$@hoare :: ['a\ assn, 'a\ com, 'a\ assn] => bool$
 $(\{-\}\ /\ -\ /\ \{-\}\ [0,55,0]\ 50)$

ML

local

fun free a = Free(a, dummyT)

fun abs((a,T),body) =

let val a = absfree(a, dummyT, body)

in if T = Bound 0 then a else Const(Syntax.constrainAbsC, dummyT) \$ a \$ T

end

in

fun mk-abstuple [x] body = abs (x, body)

| mk-abstuple (x::xs) body =

Syntax.const split \$ abs (x, mk-abstuple xs body);

fun mk-fbody a e [x as (b,-)] = if a=b then e else free b

| mk-fbody a e ((b,-)::xs) =

Syntax.const Pair \$ (if a=b then e else free b) \$ mk-fbody a e xs;

fun mk-fexp a e xs = mk-abstuple xs (mk-fbody a e xs)

end

>>

ML

fun bexp-tr (Const (TRUE, -)) xs = Syntax.const TRUE

| bexp-tr b xs = Syntax.const Collect \$ mk-abstuple xs b;

fun assn-tr r xs = Syntax.const Collect \$ mk-abstuple xs r;

>>

ML

fun com-tr (Const (@assign, -) \$ Free (a, -) \$ e) xs =

Syntax.const Basic \$ mk-fexp a e xs

| com-tr (Const (Basic, -) \$ f) xs = Syntax.const Basic \$ f

| com-tr (Const (Seq, -) \$ c1 \$ c2) xs =

Syntax.const Seq \$ com-tr c1 xs \$ com-tr c2 xs


```

| com-tr (Const (Cond,-) $ b $ c1 $ c2) xs =
  Syntax.const Cond $ bexp-tr b xs $ com-tr c1 xs $ com-tr c2 xs
| com-tr (Const (While,-) $ b $ I $ c) xs =
  Syntax.const While $ bexp-tr b xs $ assn-tr I xs $ com-tr c xs
| com-tr t - = t (* if t is just a Free/Var *)
>>

```

ML⟨⟨
local

```

fun var-tr(Free(a,-)) = (a,Bound 0) (* Bound 0 = dummy term *)
| var-tr(Const (-constrain, -) $ (Free (a,-)) $ T) = (a,T);

```

```

fun vars-tr (Const (-idts, -) $ idt $ vars) = var-tr idt :: vars-tr vars
| vars-tr t = [var-tr t]

```

```

in
fun hoare-vars-tr [vars, pre, prg, post] =
  let val xs = vars-tr vars
  in Syntax.const Valid $
    assn-tr pre xs $ com-tr prg xs $ assn-tr post xs
  end
| hoare-vars-tr ts = raise TERM (hoare-vars-tr, ts);
end
>>

```

parse-translation ⟨⟨ [(@hoare-vars, hoare-vars-tr)] ⟩⟩

ML⟨⟨

```

fun dest-abstuple (Const (split,-) $ (Abs(v,-, body))) =
  subst-bound (Syntax.free v, dest-abstuple body)
| dest-abstuple (Abs(v,-, body)) = subst-bound (Syntax.free v, body)
| dest-abstuple trm = trm;

fun abs2list (Const (split,-) $ (Abs(x,T,t))) = Free (x, T)::abs2list t
| abs2list (Abs(x,T,t)) = [Free (x, T)]
| abs2list - = [];

fun mk-ts (Const (split,-) $ (Abs(x,-,t))) = mk-ts t
| mk-ts (Abs(x,-,t)) = mk-ts t
| mk-ts (Const (Pair,-) $ a $ b) = a::(mk-ts b)
| mk-ts t = [t];

fun mk-vts (Const (split,-) $ (Abs(x,-,t))) =

```

```

      ((Syntax.free x)::(abs2list t), mk-ts t)
| mk-vts (Abs(x,-,t)) = ([Syntax.free x], [t])
| mk-vts t = raise Match;

fun find-ch [] i xs = (false, (Syntax.free not-ch, Syntax.free not-ch ))
| find-ch ((v,t)::vts) i xs = if t=(Bound i) then find-ch vts (i-1) xs
      else (true, (v, subst-bounds (xs,t)));

fun is-f (Const (split,-) $ (Abs(x,-,t))) = true
| is-f (Abs(x,-,t)) = true
| is-f t = false;
>>

```

```

ML<<
fun assn-tr' (Const (Collect,-) $ T) = dest-abstuple T
| assn-tr' (Const (@{const-name Int},-) $ (Const (Collect,-) $ T1) $
      (Const (Collect,-) $ T2)) =
      Syntax.const Set.Int $ dest-abstuple T1 $ dest-abstuple T2
| assn-tr' t = t;

fun bexp-tr' (Const (Collect,-) $ T) = dest-abstuple T
| bexp-tr' t = t;
>>

```

```

ML<<
fun mk-assign f =
  let val (vs, ts) = mk-vts f;
      val (ch, which) = find-ch (vs~~ts) ((length vs)-1) (rev vs)
  in if ch then Syntax.const @assign $ fst(which) $ snd(which)
      else Syntax.const @skip end;

fun com-tr' (Const (Basic,-) $ f) = if is-f f then mk-assign f
      else Syntax.const Basic $ f
| com-tr' (Const (Seq,-) $ c1 $ c2) = Syntax.const Seq $
      com-tr' c1 $ com-tr' c2
| com-tr' (Const (Cond,-) $ b $ c1 $ c2) = Syntax.const Cond $
      bexp-tr' b $ com-tr' c1 $ com-tr' c2
| com-tr' (Const (While,-) $ b $ I $ c) = Syntax.const While $
      bexp-tr' b $ assn-tr' I $ com-tr' c
| com-tr' t = t;

fun spec-tr' [p, c, q] =
  Syntax.const @hoare $ assn-tr' p $ com-tr' c $ assn-tr' q
>>

```

```

print-translation << [(Valid, spec-tr')] >>

```

lemma *SkipRule*: $p \subseteq q \implies \text{Valid } p \text{ (Basic id) } q$
by (*auto simp: Valid-def*)

lemma *BasicRule*: $p \subseteq \{s. f s \in q\} \implies \text{Valid } p \text{ (Basic f) } q$
by (*auto simp: Valid-def*)

lemma *SeqRule*: $\text{Valid } P \text{ } c1 \text{ } Q \implies \text{Valid } Q \text{ } c2 \text{ } R \implies \text{Valid } P \text{ (} c1; c2 \text{)} R$
by (*auto simp: Valid-def*)

lemma *CondRule*:
 $p \subseteq \{s. (s \in b \implies s \in w) \wedge (s \notin b \implies s \in w')\}$
 $\implies \text{Valid } w \text{ } c1 \text{ } q \implies \text{Valid } w' \text{ } c2 \text{ } q \implies \text{Valid } p \text{ (Cond } b \text{ } c1 \text{ } c2 \text{)} q$
by (*fastsimp simp: Valid-def image-def*)

lemma *iter-aux*:
 $! s s'. \text{Sem } c \text{ } s \text{ } s' \implies s \in \text{Some } ' (I \cap b) \implies s' \in \text{Some } ' I \implies$
 $(\bigwedge s s'. s \in \text{Some } ' I \implies \text{iter } n \text{ } b \text{ (Sem } c \text{)} s \text{ } s' \implies s' \in \text{Some } ' (I \cap -b))$
apply (*unfold image-def*)
apply (*induct n*)
apply *clarsimp*
apply (*simp (no-asm-use)*)
apply *blast*
done

lemma *WhileRule*:
 $p \subseteq i \implies \text{Valid } (i \cap b) \text{ } c \text{ } i \implies i \cap (-b) \subseteq q \implies \text{Valid } p \text{ (While } b \text{ } i \text{ } c \text{)} q$
apply (*simp add: Valid-def*)
apply (*simp (no-asm) add: image-def*)
apply *clarify*
apply (*drule iter-aux*)
prefer 2 **apply** *assumption*
apply *blast*
apply *blast*
done

lemma *AbortRule*: $p \subseteq \{s. \text{False}\} \implies \text{Valid } p \text{ Abort } q$
by (*auto simp: Valid-def*)

0.0.1 Derivation of the proof rules and, most importantly, the VCG tactic

lemma *Compl-Collect*: $\neg(\text{Collect } b) = \{x. \sim(b \text{ } x)\}$
by *blast*

use *hoare-tac.ML*

```

method-setup vcg = ⟨⟨
  Scan.succeed (fn ctxt => SIMPLE-METHOD' (hoare-tac ctxt (K all-tac))) ⟩⟩
  verification condition generator

method-setup vcg-simp = ⟨⟨
  Scan.succeed (fn ctxt =>
    SIMPLE-METHOD' (hoare-tac ctxt (asm-full-simp-tac (local-simpset-of ctxt))))
  ⟩⟩
  verification condition generator plus simplification

```

```

syntax
  guarded-com :: bool ⇒ 'a com ⇒ 'a com ((2- →/ -) 71)
  array-update :: 'a list ⇒ nat ⇒ 'a ⇒ 'a com ((2-[-] :=/ -) [70,65] 61)
translations
  P → c == IF P THEN c ELSE Abort FI
  a[i] := v => (i < CONST length a) → (a := list-update a i v)

```

Note: there is no special syntax for guarded array access. Thus you must write $j < \text{length } a \rightarrow a[i] := a!j$.

end

theory ExamplesAbort **imports** HoareAbort **begin**

```

lemma VARS x y z::nat
  {y = z & z ≠ 0} z ≠ 0 → x := y div z {x = 1}
by vcg-simp

```

```

lemma
  VARS a i j
  {k ≤ length a & i < k & j < k} j < length a → a[i] := a!j {True}
apply vcg-simp
done

```

```

lemma VARS (a::int list) i
  {True}
  i := 0;
  WHILE i < length a
  INV {i ≤ length a}
  DO a[i] := 7; i := i+1 OD
  {True}
apply vcg-simp
done

```

end

theory *Pointers0* **imports** *Hoare* **begin**

0.0.2 References

axclass *ref* < *type*
consts *Null* :: '*a*::*ref*

0.0.3 Field access and update

syntax

@fassign :: '*a*::*ref* => *id* => '*v* => '*s* *com*
 ((2-[^].- :=/ -) [70,1000,65] 61)
@faccess :: '*a*::*ref* => ('*a*::*ref* => '*v*) => '*v*
 (-[^].- [65,1000] 65)

translations

p[^].*f* := *e* => *f* := *fun-upd f p e*
p[^].*f* => *f p*

An example due to Suzuki:

lemma *VARs* *v n*
 {*distinct*[*w,x,y,z*]}
w[^].*v* := (1::*int*); *w*[^].*n* := *x*;
x[^].*v* := 2; *x*[^].*n* := *y*;
y[^].*v* := 3; *y*[^].*n* := *z*;
z[^].*v* := 4; *x*[^].*n* := *z*
 {*w*[^].*n*[^].*n*[^].*v* = 4}
by *vcg-simp*

0.1 The heap

0.1.1 Paths in the heap

consts

Path :: ('*a*::*ref* => '*a*) => '*a* => '*a* *list* => '*a* => *bool*

primrec

Path h x [] = (*x* = *y*)
*Path h x (a#*as*)* *y* = (*x* ≠ *Null* ∧ *x* = *a* ∧ *Path h (h a) as* *y*)

lemma [*iff*]: *Path h Null xs y* = (*xs* = [] ∧ *y* = *Null*)

apply(*case-tac xs*)

apply *fastsimp*

apply *fastsimp*

done

lemma [*simp*]: *a* ≠ *Null* ⇒ *Path h a as z* =
 (*as* = [] ∧ *z* = *a* ∨ (∃ *bs*. *as* = *a*#*bs* ∧ *Path h (h a) bs z*))
apply(*case-tac as*)

```

apply fastsimp
apply fastsimp
done

```

```

lemma [simp]:  $\bigwedge x. \text{Path } f \ x \ (as@bs) \ z = (\exists y. \text{Path } f \ x \ as \ y \wedge \text{Path } f \ y \ bs \ z)$ 
by(induct as, simp+)

```

```

lemma [simp]:  $\bigwedge x. u \notin \text{set } as \implies \text{Path } (f(u := v)) \ x \ as \ y = \text{Path } f \ x \ as \ y$ 
by(induct as, simp, simp add: eq-sym-conv)

```

0.1.2 Lists on the heap

Relational abstraction

```

constdefs
  List :: ('a::ref  $\Rightarrow$  'a)  $\Rightarrow$  'a  $\Rightarrow$  'a list  $\Rightarrow$  bool
  List h x as == Path h x as Null

```

```

lemma [simp]: List h x [] = (x = Null)
by(simp add: List-def)

```

```

lemma [simp]: List h x (a#as) = (x  $\neq$  Null  $\wedge$  x = a  $\wedge$  List h (h a) as)
by(simp add: List-def)

```

```

lemma [simp]: List h Null as = (as = [])
by(case-tac as, simp-all)

```

```

lemma List-Ref[simp]:
  a  $\neq$  Null  $\implies$  List h a as = ( $\exists bs. as = a\#bs \wedge List \ h \ (h \ a) \ bs$ )
by(case-tac as, simp-all, fast)

```

```

theorem notin-List-update[simp]:
   $\bigwedge x. a \notin \text{set } as \implies \text{List } (h(a := y)) \ x \ as = \text{List } h \ x \ as$ 
apply(induct as)
apply simp
apply(clarsimp simp add: fun-upd-apply)
done

```

```

declare fun-upd-apply[simp del]fun-upd-same[simp] fun-upd-other[simp]

```

```

lemma List-unique:  $\bigwedge x \ bs. List \ h \ x \ as \implies List \ h \ x \ bs \implies as = bs$ 
by(induct as, simp, clarsimp)

```

```

lemma List-unique1: List h p as  $\implies \exists ! as. List \ h \ p \ as$ 
by(blast intro: List-unique)

```

```

lemma List-app:  $\bigwedge x. List \ h \ x \ (as@bs) = (\exists y. Path \ h \ x \ as \ y \wedge List \ h \ y \ bs)$ 
by(induct as, simp, clarsimp)

```

```

lemma List-hd-not-in-tl[simp]: List h (h a) as  $\implies a \notin \text{set } as$ 
apply (clarsimp simp add:in-set-conv-decomp)
apply(frule List-app[THEN iffD1])
apply(fastsimp dest: List-unique)
done

```

```

lemma List-distinct[simp]:  $\bigwedge x. \text{List } h \ x \ as \implies \text{distinct } as$ 
apply(induct as, simp)
apply(fastsimp dest:List-hd-not-in-tl)
done

```

0.1.3 Functional abstraction

```

constdefs
  islist :: ('a::ref  $\Rightarrow$  'a)  $\Rightarrow$  'a  $\Rightarrow$  bool
  islist h p ==  $\exists as. \text{List } h \ p \ as$ 
  list :: ('a::ref  $\Rightarrow$  'a)  $\Rightarrow$  'a  $\Rightarrow$  'a list
  list h p == SOME as. List h p as

lemma List-conv-islist-list: List h p as = (islist h p  $\wedge as = \text{list } h \ p$ )
apply(simp add:islist-def list-def)
apply(rule iffI)
apply(rule conjI)
apply blast
apply(subst some1-equality)
  apply(erule List-unique1)
  apply assumption
apply(rule refl)
apply simp
apply(rule someI-ex)
apply fast
done

```

```

lemma [simp]: islist h Null
by(simp add:islist-def)

```

```

lemma [simp]: a  $\neq \text{Null} \implies \text{islist } h \ a = \text{islist } h \ (h \ a)$ 
by(simp add:islist-def)

```

```

lemma [simp]: list h Null = []
by(simp add:list-def)

```

```

lemma list-Ref-conv[simp]:
   $\llbracket a \neq \text{Null}; \text{islist } h \ (h \ a) \rrbracket \implies \text{list } h \ a = a \ \# \ \text{list } h \ (h \ a)$ 
apply(insert List-Ref[of - h])
apply(fastsimp simp:List-conv-islist-list)
done

```

```

lemma [simp]: islist h (h a)  $\implies a \notin \text{set}(\text{list } h \ (h \ a))$ 

```

```

apply(insert List-hd-not-in-tl[of h])
apply(simp add:List-conv-islist-list)
done

```

```

lemma list-upd-conv[simp]:
  islist h p  $\implies y \notin \text{set}(\text{list } h \text{ } p) \implies \text{list } (h(y := q)) \text{ } p = \text{list } h \text{ } p$ 
apply(drule notin-List-update[of - - h q p])
apply(simp add:List-conv-islist-list)
done

```

```

lemma islist-upd[simp]:
  islist h p  $\implies y \notin \text{set}(\text{list } h \text{ } p) \implies \text{islist } (h(y := q)) \text{ } p$ 
apply(frule notin-List-update[of - - h q p])
apply(simp add:List-conv-islist-list)
done

```

0.2 Verifications

0.2.1 List reversal

A short but unreadable proof:

```

lemma VARs tl p q r
  {List tl p Ps  $\wedge$  List tl q Qs  $\wedge$  set Ps  $\cap$  set Qs = {}}
  WHILE p  $\neq$  Null
  INV { $\exists ps \text{ } qs. \text{ } List \text{ } tl \text{ } p \text{ } ps \wedge List \text{ } tl \text{ } q \text{ } qs \wedge set \text{ } ps \cap set \text{ } qs = \{ \} \wedge$ 
       $rev \text{ } ps @ qs = rev \text{ } Ps @ Qs$ }
  DO r := p; p := p^.tl; r^.tl := q; q := r OD
  {List tl q (rev Ps @ Qs)}
apply vcg-simp
apply fastsimp
apply(fastsimp intro:notin-List-update[THEN iffD2])

apply fastsimp
done

```

A longer readable version:

```

lemma VARs tl p q r
  {List tl p Ps  $\wedge$  List tl q Qs  $\wedge$  set Ps  $\cap$  set Qs = {}}
  WHILE p  $\neq$  Null
  INV { $\exists ps \text{ } qs. \text{ } List \text{ } tl \text{ } p \text{ } ps \wedge List \text{ } tl \text{ } q \text{ } qs \wedge set \text{ } ps \cap set \text{ } qs = \{ \} \wedge$ 
       $rev \text{ } ps @ qs = rev \text{ } Ps @ Qs$ }
  DO r := p; p := p^.tl; r^.tl := q; q := r OD
  {List tl q (rev Ps @ Qs)}
proof vcg
  fix tl p q r
  assume List tl p Ps  $\wedge$  List tl q Qs  $\wedge$  set Ps  $\cap$  set Qs = {}}
  thus  $\exists ps \text{ } qs. \text{ } List \text{ } tl \text{ } p \text{ } ps \wedge List \text{ } tl \text{ } q \text{ } qs \wedge set \text{ } ps \cap set \text{ } qs = \{ \} \wedge$ 
       $rev \text{ } ps @ qs = rev \text{ } Ps @ Qs$  by fastsimp
next

```



```

fix  $tl\ p\ q\ r$ 
assume  $(\exists ps\ qs. List\ tl\ p\ ps \wedge List\ tl\ q\ qs \wedge set\ ps \cap set\ qs = \{\} \wedge$ 
 $rev\ ps\ @\ qs = rev\ Ps\ @\ Qs) \wedge p \neq Null$ 
 $(is\ (\exists ps\ qs. ?I\ ps\ qs) \wedge \neg)$ 
then obtain  $ps\ qs$  where  $I: ?I\ ps\ qs \wedge p \neq Null$  by fast
then obtain  $ps'$  where  $ps = p \# ps'$  by fastsimp
hence  $List\ (tl(p := q))\ (p^\wedge.tl)\ ps' \wedge$ 
 $List\ (tl(p := q))\ p\ (p \# qs) \wedge$ 
 $set\ ps' \cap set\ (p \# qs) = \{\} \wedge$ 
 $rev\ ps' @ (p \# qs) = rev\ Ps @ Qs$ 
using  $I$  by fastsimp
thus  $\exists ps'\ qs'. List\ (tl(p := q))\ (p^\wedge.tl)\ ps' \wedge$ 
 $List\ (tl(p := q))\ p\ qs' \wedge$ 
 $set\ ps' \cap set\ qs' = \{\} \wedge$ 
 $rev\ ps' @ qs' = rev\ Ps @ Qs$  by fast

next
fix  $tl\ p\ q\ r$ 
assume  $(\exists ps\ qs. List\ tl\ p\ ps \wedge List\ tl\ q\ qs \wedge set\ ps \cap set\ qs = \{\} \wedge$ 
 $rev\ ps\ @\ qs = rev\ Ps\ @\ Qs) \wedge \neg p \neq Null$ 
thus  $List\ tl\ q\ (rev\ Ps @ Qs)$  by fastsimp
qed

```

Finally, the functional version. A bit more verbose, but automatic!

```

lemma VARs  $tl\ p\ q\ r$ 
 $\{islist\ tl\ p \wedge islist\ tl\ q \wedge$ 
 $Ps = list\ tl\ p \wedge Qs = list\ tl\ q \wedge set\ Ps \cap set\ Qs = \{\}\}$ 
 $WHILE\ p \neq Null$ 
 $INV\ \{islist\ tl\ p \wedge islist\ tl\ q \wedge$ 
 $set(list\ tl\ p) \cap set(list\ tl\ q) = \{\} \wedge$ 
 $rev(list\ tl\ p) @ (list\ tl\ q) = rev\ Ps @ Qs\}$ 
 $DO\ r := p; p := p^\wedge.tl; r^\wedge.tl := q; q := r\ OD$ 
 $\{islist\ tl\ q \wedge list\ tl\ q = rev\ Ps @ Qs\}$ 
apply vcg-simp
apply clarsimp
apply clarsimp
apply clarsimp
done

```

0.2.2 Searching in a list

What follows is a sequence of successively more intelligent proofs that a simple loop finds an element in a linked list.

We start with a proof based on the *List* predicate. This means it only works for acyclic lists.

```

lemma VARs  $tl\ p$ 
 $\{List\ tl\ p\ Ps \wedge X \in set\ Ps\}$ 
 $WHILE\ p \neq Null \wedge p \neq X$ 
 $INV\ \{p \neq Null \wedge (\exists ps. List\ tl\ p\ ps \wedge X \in set\ ps)\}$ 
 $DO\ p := p^\wedge.tl\ OD$ 

```

```

    {p = X}
  apply vcg-simp
    apply(case-tac p = Null)
      apply clarsimp
      apply fastsimp
      apply clarsimp
      apply fastsimp
      apply clarsimp
    done

```

Using *Path* instead of *List* generalizes the correctness statement to cyclic lists as well:

```

lemma VARS tl p
  {Path tl p Ps X}
  WHILE p ≠ Null ∧ p ≠ X
  INV {∃ ps. Path tl p ps X}
  DO p := p^.tl OD
  {p = X}
  apply vcg-simp
  apply blast
  apply fastsimp
  apply clarsimp
done

```

Now it dawns on us that we do not need the list witness at all — it suffices to talk about reachability, i.e. we can use relations directly.

```

lemma VARS tl p
  {(p,X) ∈ {(x,y). y = tl x & x ≠ Null}^*}
  WHILE p ≠ Null ∧ p ≠ X
  INV {(p,X) ∈ {(x,y). y = tl x & x ≠ Null}^*}
  DO p := p^.tl OD
  {p = X}
  apply vcg-simp
  apply clarsimp
  apply(erule converse-rtranclE)
  apply simp
  apply(simp)
  apply(fastsimp elim:converse-rtranclE)
done

```

0.2.3 Merging two lists

This is still a bit rough, especially the proof.

```

consts merge :: 'a list * 'a list * ('a ⇒ 'a ⇒ bool) ⇒ 'a list

```

```

recdef merge measure(%(xs,ys,f). size xs + size ys)
merge(x#xs,y#ys,f) = (if f x y then x # merge(xs,y#ys,f)
                      else y # merge(x#xs,ys,f))
merge(x#xs,[],f) = x # merge(xs,[],f)

```

$merge([], y \# ys, f) = y \# merge([], ys, f)$
 $merge([], [], f) = []$

lemma *imp-disjCL*: $(P|Q \longrightarrow R) = ((P \longrightarrow R) \wedge (\sim P \longrightarrow Q \longrightarrow R))$
by *blast*

declare *disj-not1* [*simp del*] *imp-disjL* [*simp del*] *imp-disjCL* [*simp*]

lemma *VARs hd tl p q r s*
 $\{List\ tl\ p\ Ps \wedge List\ tl\ q\ Qs \wedge set\ Ps \cap set\ Qs = \{\} \wedge$
 $(p \neq Null \vee q \neq Null)\}$
IF if $q = Null$ then *True* else $p \sim = Null \ \& \ p.^{hd} \leq q.^{hd}$
THEN $r := p; p := p.^{tl}$ *ELSE* $r := q; q := q.^{tl}$ *FI*;
 $s := r$;
WHILE $p \neq Null \vee q \neq Null$
INV $\{EX\ rs\ ps\ qs. Path\ tl\ r\ rs\ s \wedge List\ tl\ p\ ps \wedge List\ tl\ q\ qs \wedge$
 $distinct(s \# ps @ qs @ rs) \wedge s \neq Null \wedge$
 $merge(Ps, Qs, \lambda x\ y. hd\ x \leq hd\ y) =$
 $rs @ s \# merge(ps, qs, \lambda x\ y. hd\ x \leq hd\ y) \wedge$
 $(tl\ s = p \vee tl\ s = q)\}$
DO IF if $q = Null$ then *True* else $p \neq Null \wedge p.^{hd} \leq q.^{hd}$
THEN $s.^{tl} := p; p := p.^{tl}$ *ELSE* $s.^{tl} := q; q := q.^{tl}$ *FI*;
 $s := s.^{tl}$
OD
 $\{List\ tl\ r\ (merge(Ps, Qs, \lambda x\ y. hd\ x \leq hd\ y))\}$
apply *vcg-simp*

apply (*fastsimp*)

apply *clarsimp*
apply (*rule conjI*)
apply *clarsimp*
apply (*simp add: eq-sym-conv*)
apply (*rule-tac x = rs @ [s] in exI*)
apply *simp*
apply (*rule-tac x = bs in exI*)
apply (*fastsimp simp: eq-sym-conv*)

apply *clarsimp*
apply (*rule conjI*)
apply *clarsimp*
apply (*rule-tac x = rs @ [s] in exI*)
apply *simp*
apply (*rule-tac x = bsa in exI*)
apply (*rule conjI*)
apply (*simp add: eq-sym-conv*)
apply (*rule exI*)
apply (*rule conjI*)
apply (*rule-tac x = bs in exI*)

```

apply(rule conjI)
apply(rule refl)
apply (simp add:eq-sym-conv)
apply (simp add:eq-sym-conv)

apply(rule conjI)
apply clarsimp
apply(rule-tac x = rs @ [s] in exI)
apply simp
apply(rule-tac x = bs in exI)
apply (simp add:eq-sym-conv)
apply clarsimp
apply(rule-tac x = rs @ [s] in exI)
apply (simp add:eq-sym-conv)
apply(rule exI)
apply(rule conjI)
apply(rule-tac x = bsa in exI)
apply(rule conjI)
apply(rule refl)
apply (simp add:eq-sym-conv)
apply(rule-tac x = bs in exI)
apply (simp add:eq-sym-conv)

apply(clarsimp simp add:List-app)
done

```

0.2.4 Storage allocation

```

constdefs new :: 'a set  $\Rightarrow$  'a::ref
new A == SOME a. a  $\notin$  A & a  $\neq$  Null

```

```

lemma new-notin:
 $\llbracket \sim \text{finite}(\text{UNIV}::('a::\text{ref})\text{set}); \text{finite}(A::'a\text{ set}); B \subseteq A \rrbracket \implies$ 
  new (A)  $\notin$  B & new A  $\neq$  Null
apply(unfold new-def)
apply(rule someI2-ex)
apply (fast dest:ex-new-if-finite[of insert Null A])
apply (fast)
done

```

```

lemma  $\sim \text{finite}(\text{UNIV}::('a::\text{ref})\text{set}) \implies$ 
  VARS xs elem next alloc p q
  {Xs = xs  $\wedge$  p = (Null::'a)}
  WHILE xs  $\neq$  []
  INV {islist next p  $\wedge$  set(list next p)  $\subseteq$  set alloc  $\wedge$ 
    map elem (rev(list next p)) @ xs = Xs}
  DO q := new(set alloc); alloc := q#alloc;
    q^.next := p; q^.elem := hd xs; xs := tl xs; p := q

```

```

    OD
    {islist next p ∧ map elem (rev(list next p)) = Xs}
  apply vcg-simp
  apply (clarsimp simp: subset-insert-iff neq-Nil-conv fun-upd-apply new-notin)
  apply fastsimp
done

end

```

theory Heap imports Main begin

0.2.5 References

```

datatype 'a ref = Null | Ref 'a

lemma not-Null-eq [iff]: (x ~ = Null) = (EX y. x = Ref y)
  by (induct x) auto

lemma not-Ref-eq [iff]: (ALL y. x ~ = Ref y) = (x = Null)
  by (induct x) auto

consts addr :: 'a ref ⇒ 'a
primrec addr(Ref a) = a

```

0.3 The heap

0.3.1 Paths in the heap

```

consts
  Path :: ('a ⇒ 'a ref) ⇒ 'a ref ⇒ 'a list ⇒ 'a ref ⇒ bool
primrec
  Path h x [] y = (x = y)
  Path h x (a#as) y = (x = Ref a ∧ Path h (h a) as y)

lemma [iff]: Path h Null xs y = (xs = [] ∧ y = Null)
apply(case-tac xs)
apply fastsimp
apply fastsimp
done

lemma [simp]: Path h (Ref a) as z =
  (as = [] ∧ z = Ref a ∨ (∃ bs. as = a#bs ∧ Path h (h a) bs z))
apply(case-tac as)
apply fastsimp
apply fastsimp
done

```

lemma [simp]: $\bigwedge x. \text{Path } f \ x \ (as @ bs) \ z = (\exists y. \text{Path } f \ x \ as \ y \wedge \text{Path } f \ y \ bs \ z)$
by(*induct as, simp+*)

lemma *Path-upd*[simp]:
 $\bigwedge x. u \notin \text{set } as \implies \text{Path } (f(u := v)) \ x \ as \ y = \text{Path } f \ x \ as \ y$
by(*induct as, simp, simp add: eq-sym-conv*)

lemma *Path-snoc*:
 $\text{Path } (f(a := q)) \ p \ as \ (\text{Ref } a) \implies \text{Path } (f(a := q)) \ p \ (as @ [a]) \ q$
by *simp*

0.3.2 Non-repeating paths

constdefs
 $\text{distPath} :: ('a \Rightarrow 'a \text{ ref}) \Rightarrow 'a \text{ ref} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ ref} \Rightarrow \text{bool}$
 $\text{distPath } h \ x \ as \ y \equiv \text{Path } h \ x \ as \ y \wedge \text{distinct } as$

The term $\text{distPath } h \ x \ as \ y$ expresses the fact that a non-repeating path as connects location x to location y by means of the h field. In the case where $x = y$, and there is a cycle from x to itself, as can be both $[]$ and the non-repeating list of nodes in the cycle.

lemma *neq-dP*: $p \neq q \implies \text{Path } h \ p \ Ps \ q \implies \text{distinct } Ps \implies$
 $EX \ a \ Qs. p = \text{Ref } a \ \& \ Ps = a \# Qs \ \& \ a \notin \text{set } Qs$
by (*case-tac Ps, auto*)

lemma *neq-dP-disp*: $\llbracket p \neq q; \text{distPath } h \ p \ Ps \ q \rrbracket \implies$
 $EX \ a \ Qs. p = \text{Ref } a \wedge Ps = a \# Qs \wedge a \notin \text{set } Qs$
apply (*simp only: distPath-def*)
by (*case-tac Ps, auto*)

0.3.3 Lists on the heap

Relational abstraction

constdefs
 $\text{List} :: ('a \Rightarrow 'a \text{ ref}) \Rightarrow 'a \text{ ref} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$
 $\text{List } h \ x \ as == \text{Path } h \ x \ as \ \text{Null}$

lemma [simp]: $\text{List } h \ x \ [] = (x = \text{Null})$
by(*simp add: List-def*)

lemma [simp]: $\text{List } h \ x \ (a \# as) = (x = \text{Ref } a \wedge \text{List } h \ (h \ a) \ as)$
by(*simp add: List-def*)

lemma [simp]: $\text{List } h \ \text{Null} \ as = (as = [])$
by(*case-tac as, simp-all*)

lemma *List-Ref*[simp]: $\text{List } h \ (\text{Ref } a) \ as = (\exists bs. as = a \# bs \wedge \text{List } h \ (h \ a) \ bs)$

by(*case-tac as, simp-all, fast*)

theorem *notin-List-update[simp]*:

$\bigwedge x. a \notin \text{set } as \implies \text{List } (h(a := y)) \ x \ as = \text{List } h \ x \ as$

apply(*induct as*)

apply *simp*

apply(*clarsimp simp add:fun-upd-apply*)

done

lemma *List-unique*: $\bigwedge x \ bs. \text{List } h \ x \ as \implies \text{List } h \ x \ bs \implies as = bs$

by(*induct as, simp, clarsimp*)

lemma *List-unique1*: $\text{List } h \ p \ as \implies \exists! as. \text{List } h \ p \ as$

by(*blast intro:List-unique*)

lemma *List-app*: $\bigwedge x. \text{List } h \ x \ (as@bs) = (\exists y. \text{Path } h \ x \ as \ y \wedge \text{List } h \ y \ bs)$

by(*induct as, simp, clarsimp*)

lemma *List-hd-not-in-tl[simp]*: $\text{List } h \ (h \ a) \ as \implies a \notin \text{set } as$

apply (*clarsimp simp add:in-set-conv-decomp*)

apply(*frule List-app[THEN iffD1]*)

apply(*fastsimp dest:List-unique*)

done

lemma *List-distinct[simp]*: $\bigwedge x. \text{List } h \ x \ as \implies \text{distinct } as$

apply(*induct as, simp*)

apply(*fastsimp dest:List-hd-not-in-tl*)

done

lemma *Path-is-List*:

$\llbracket \text{Path } h \ b \ Ps \ (\text{Ref } a); a \notin \text{set } Ps \rrbracket \implies \text{List } (h(a := \text{Null})) \ b \ (Ps @ [a])$

apply (*induct Ps arbitrary: b*)

apply (*auto simp add:fun-upd-apply*)

done

0.3.4 Functional abstraction

constdefs

islist :: $('a \Rightarrow 'a \text{ ref}) \Rightarrow 'a \text{ ref} \Rightarrow \text{bool}$

islist $h \ p == \exists as. \text{List } h \ p \ as$

list :: $('a \Rightarrow 'a \text{ ref}) \Rightarrow 'a \text{ ref} \Rightarrow 'a \text{ list}$

list $h \ p == \text{SOME } as. \text{List } h \ p \ as$

lemma *List-conv-islist-list*: $\text{List } h \ p \ as = (\text{islist } h \ p \wedge as = \text{list } h \ p)$

apply(*simp add:islist-def list-def*)

apply(*rule iffI*)

apply(*rule conjI*)

apply *blast*

apply(*subst some1-equality*)

```

    apply(erule List-unique1)
  apply assumption
  apply(rule refl)
  apply simp
  apply(rule someI-ex)
  apply fast
done

lemma [simp]: islist h Null
by(simp add:islist-def)

lemma [simp]: islist h (Ref a) = islist h (h a)
by(simp add:islist-def)

lemma [simp]: list h Null = []
by(simp add:list-def)

lemma list-Ref-conv[simp]:
  islist h (h a)  $\implies$  list h (Ref a) = a # list h (h a)
  apply(insert List-Ref[of h])
  apply(fastsimp simp:List-conv-islist-list)
done

lemma [simp]: islist h (h a)  $\implies$  a  $\notin$  set(list h (h a))
  apply(insert List-hd-not-in-tl[of h])
  apply(simp add:List-conv-islist-list)
done

lemma list-upd-conv[simp]:
  islist h p  $\implies$  y  $\notin$  set(list h p)  $\implies$  list (h(y := q)) p = list h p
  apply(drule notin-List-update[of - h q p])
  apply(simp add:List-conv-islist-list)
done

lemma islist-upd[simp]:
  islist h p  $\implies$  y  $\notin$  set(list h p)  $\implies$  islist (h(y := q)) p
  apply(frule notin-List-update[of - h q p])
  apply(simp add:List-conv-islist-list)
done

end

```

theory *HeapSyntax* **imports** *Hoare Heap* **begin**

0.3.5 Field access and update

syntax


```

@refupdate :: ('a ⇒ 'b) ⇒ 'a ref ⇒ 'b ⇒ ('a ⇒ 'b)
  (-/'((- → -)') [1000,0] 900)
@fassign :: 'a ref => id => 'v => 's com
  ((2-^.- := / -) [70,1000,65] 61)
@faccess :: 'a ref => ('a ref ⇒ 'v) => 'v
  (-^.- [65,1000] 65)
translations
  f(r → v) == f(addr r := v)
  p^.f := e => f := f(p → e)
  p^.f      => f(addr p)

declare fun-upd-apply[simp del] fun-upd-same[simp] fun-upd-other[simp]

```

An example due to Suzuki:

```

lemma VARS v n
  {w = Ref w0 & x = Ref x0 & y = Ref y0 & z = Ref z0 &
   distinct[w0,x0,y0,z0]}
  w^.v := (1::int); w^.n := x;
  x^.v := 2; x^.n := y;
  y^.v := 3; y^.n := z;
  z^.v := 4; x^.n := z
  {w^.n^.n^.v = 4}
by vcg-simp

end

```

theory *Pointer-Examples* **imports** *HeapSyntax* **begin**

axiomatization where *unproven*: *PROP A*

0.4 Verifications

0.4.1 List reversal

A short but unreadable proof:

```

lemma VARS tl p q r
  {List tl p Ps ∧ List tl q Qs ∧ set Ps ∩ set Qs = {}}
  WHILE p ≠ Null
  INV {∃ ps qs. List tl p ps ∧ List tl q qs ∧ set ps ∩ set qs = {} ∧
       rev ps @ qs = rev Ps @ Qs}
  DO r := p; p := p^.tl; r^.tl := q; q := r OD
  {List tl q (rev Ps @ Qs)}
apply vcg-simp
apply fastsimp
apply(fastsimp intro:notin-List-update[THEN iffD2])

```

apply *fastsimp*
done

And now with ghost variables *ps* and *qs*. Even “more automatic”.

lemma *VARs next p ps q qs r*
 $\{List\ next\ p\ Ps \wedge List\ next\ q\ Qs \wedge set\ Ps \cap set\ Qs = \{\}\ \wedge$
 $ps = Ps \wedge qs = Qs\}$
 $WHILE\ p \neq Null$
 $INV\ \{List\ next\ p\ ps \wedge List\ next\ q\ qs \wedge set\ ps \cap set\ qs = \{\}\ \wedge$
 $rev\ ps\ @\ qs = rev\ Ps\ @\ Qs\}$
 $DO\ r := p; p := p.^next; r.^next := q; q := r;$
 $qs := (hd\ ps) \# qs; ps := tl\ ps\ OD$
 $\{List\ next\ q\ (rev\ Ps\ @\ Qs)\}$
apply *vcg-simp*
apply *fastsimp*
apply *fastsimp*
done

A longer readable version:

lemma *VARs tl p q r*
 $\{List\ tl\ p\ Ps \wedge List\ tl\ q\ Qs \wedge set\ Ps \cap set\ Qs = \{\}\}$
 $WHILE\ p \neq Null$
 $INV\ \{\exists ps\ qs. List\ tl\ p\ ps \wedge List\ tl\ q\ qs \wedge set\ ps \cap set\ qs = \{\}\ \wedge$
 $rev\ ps\ @\ qs = rev\ Ps\ @\ Qs\}$
 $DO\ r := p; p := p.^tl; r.^tl := q; q := r\ OD$
 $\{List\ tl\ q\ (rev\ Ps\ @\ Qs)\}$
proof *vcg*
fix *tl p q r*
assume $List\ tl\ p\ Ps \wedge List\ tl\ q\ Qs \wedge set\ Ps \cap set\ Qs = \{\}$
thus $\exists ps\ qs. List\ tl\ p\ ps \wedge List\ tl\ q\ qs \wedge set\ ps \cap set\ qs = \{\}\ \wedge$
 $rev\ ps\ @\ qs = rev\ Ps\ @\ Qs$ **by** *fastsimp*
next
fix *tl p q r*
assume $(\exists ps\ qs. List\ tl\ p\ ps \wedge List\ tl\ q\ qs \wedge set\ ps \cap set\ qs = \{\}\ \wedge$
 $rev\ ps\ @\ qs = rev\ Ps\ @\ Qs) \wedge p \neq Null$
 $(is\ (\exists ps\ qs. ?I\ ps\ qs) \wedge -)$
then obtain *ps qs a* **where** *I*: $?I\ ps\ qs \wedge p = Ref\ a$
by *fast*
then obtain *ps'* **where** $ps = a \# ps'$ **by** *fastsimp*
hence $List\ (tl(p \rightarrow q))\ (p.^tl)\ ps' \wedge$
 $List\ (tl(p \rightarrow q))\ p\ (a \# qs) \wedge$
 $set\ ps' \cap set\ (a \# qs) = \{\}\ \wedge$
 $rev\ ps' \ @\ (a \# qs) = rev\ Ps\ @\ Qs$
using *I* **by** *fastsimp*
thus $\exists ps'\ qs'. List\ (tl(p \rightarrow q))\ (p.^tl)\ ps' \wedge$
 $List\ (tl(p \rightarrow q))\ p\ qs' \wedge$
 $set\ ps' \cap set\ qs' = \{\}\ \wedge$
 $rev\ ps' \ @\ qs' = rev\ Ps\ @\ Qs$ **by** *fast*
next

```

fix  $tl\ p\ q\ r$ 
assume  $(\exists ps\ qs. List\ tl\ p\ ps \wedge List\ tl\ q\ qs \wedge set\ ps \cap set\ qs = \{\} \wedge$ 
 $rev\ ps\ @\ qs = rev\ Ps\ @\ Qs) \wedge \neg p \neq Null$ 
thus  $List\ tl\ q\ (rev\ Ps\ @\ Qs)$  by fastsimp
qed

```

Finally, the functional version. A bit more verbose, but automatic!

```

lemma  $VARs\ tl\ p\ q\ r$ 
 $\{islist\ tl\ p \wedge islist\ tl\ q \wedge$ 
 $Ps = list\ tl\ p \wedge Qs = list\ tl\ q \wedge set\ Ps \cap set\ Qs = \{\}\}$ 
 $WHILE\ p \neq Null$ 
 $INV\ \{islist\ tl\ p \wedge islist\ tl\ q \wedge$ 
 $set(list\ tl\ p) \cap set(list\ tl\ q) = \{\} \wedge$ 
 $rev(list\ tl\ p)\ @\ (list\ tl\ q) = rev\ Ps\ @\ Qs\}$ 
 $DO\ r := p; p := p^.tl; r^.tl := q; q := r\ OD$ 
 $\{islist\ tl\ q \wedge list\ tl\ q = rev\ Ps\ @\ Qs\}$ 
apply vcg-simp
apply clarsimp
apply clarsimp
apply clarsimp
done

```

0.4.2 Searching in a list

What follows is a sequence of successively more intelligent proofs that a simple loop finds an element in a linked list.

We start with a proof based on the *List* predicate. This means it only works for acyclic lists.

```

lemma  $VARs\ tl\ p$ 
 $\{List\ tl\ p\ Ps \wedge X \in set\ Ps\}$ 
 $WHILE\ p \neq Null \wedge p \neq Ref\ X$ 
 $INV\ \{\exists ps. List\ tl\ p\ ps \wedge X \in set\ ps\}$ 
 $DO\ p := p^.tl\ OD$ 
 $\{p = Ref\ X\}$ 
apply vcg-simp
apply blast
apply clarsimp
apply clarsimp
done

```

Using *Path* instead of *List* generalizes the correctness statement to cyclic lists as well:

```

lemma  $VARs\ tl\ p$ 
 $\{Path\ tl\ p\ Ps\ X\}$ 
 $WHILE\ p \neq Null \wedge p \neq X$ 
 $INV\ \{\exists ps. Path\ tl\ p\ ps\ X\}$ 
 $DO\ p := p^.tl\ OD$ 
 $\{p = X\}$ 

```

```

apply vcg-simp
  apply blast
  apply fastsimp
apply clarsimp
done

```

Now it dawns on us that we do not need the list witness at all — it suffices to talk about reachability, i.e. we can use relations directly. The first version uses a relation on *'a ref*:

```

lemma VARs tl p
   $\{(p, X) \in \{(Ref\ x, tl\ x) \mid x. True\}^*\}$ 
  WHILE  $p \neq Null \wedge p \neq X$ 
  INV  $\{(p, X) \in \{(Ref\ x, tl\ x) \mid x. True\}^*\}$ 
  DO  $p := p.^{tl}\ OD$ 
   $\{p = X\}$ 
apply vcg-simp
apply clarsimp
apply(erule converse-rtranclE)
apply simp
apply(clarsimp elim:converse-rtranclE)
apply(fast elim:converse-rtranclE)
done

```

Finally, a version based on a relation on type *'a*:

```

lemma VARs tl p
   $\{p \neq Null \wedge (addr\ p, X) \in \{(x, y). tl\ x = Ref\ y\}^*\}$ 
  WHILE  $p \neq Null \wedge p \neq Ref\ X$ 
  INV  $\{p \neq Null \wedge (addr\ p, X) \in \{(x, y). tl\ x = Ref\ y\}^*\}$ 
  DO  $p := p.^{tl}\ OD$ 
   $\{p = Ref\ X\}$ 
apply vcg-simp
apply clarsimp
apply(erule converse-rtranclE)
apply simp
apply clarsimp
apply clarsimp
done

```

0.4.3 Splicing two lists

```

lemma VARs tl p q pp qq
   $\{List\ tl\ p\ Ps \wedge List\ tl\ q\ Qs \wedge set\ Ps \cap set\ Qs = \{\} \wedge size\ Qs \leq size\ Ps\}$ 
   $pp := p;$ 
  WHILE  $q \neq Null$ 
  INV  $\{\exists as\ bs\ qs.$ 
     $distinct\ as \wedge Path\ tl\ p\ as\ pp \wedge List\ tl\ pp\ bs \wedge List\ tl\ q\ qs \wedge$ 
     $set\ bs \cap set\ qs = \{\} \wedge set\ as \cap (set\ bs \cup set\ qs) = \{\} \wedge$ 
     $size\ qs \leq size\ bs \wedge splice\ Ps\ Qs = as\ @\ splice\ bs\ qs\}$ 
  DO  $qq := q.^{tl}; q.^{tl} := pp.^{tl}; pp.^{tl} := q; pp := q.^{tl}; q := qq\ OD$ 

```

```

    {List tl p (splice Ps Qs)}
  apply vcg-simp
    apply (rule-tac x = [] in exI)
    apply fastsimp
    apply clarsimp
    apply (rename-tac y bs qqs)
    apply (case-tac bs) apply simp
    apply clarsimp
    apply (rename-tac x bbs)
    apply (rule-tac x = as @ [x,y] in exI)
    apply simp
    apply (rule-tac x = bbs in exI)
    apply simp
    apply (rule-tac x = qqs in exI)
    apply simp
  apply (fastsimp simp:List-app)
done

```

0.4.4 Merging two lists

This is still a bit rough, especially the proof.

constdefs

```

  cor :: bool ⇒ bool ⇒ bool
  cor P Q == if P then True else Q
  cand :: bool ⇒ bool ⇒ bool
  cand P Q == if P then Q else False

```

consts merge :: 'a list * 'a list * ('a ⇒ 'a ⇒ bool) ⇒ 'a list

```

recdef merge measure (%(xs,ys,f). size xs + size ys)
merge(x#xs,y#ys,f) = (if f x y then x # merge(xs,y#ys,f)
                      else y # merge(x#xs,ys,f))
merge(x#xs,[],f) = x # merge(xs,[],f)
merge([],y#ys,f) = y # merge([],ys,f)
merge([],[],f) = []

```

Simplifies the proof a little:

```

lemma [simp]: ({ } = insert a A ∩ B) = (a ∉ B & { } = A ∩ B)
by blast
lemma [simp]: ({ } = A ∩ insert b B) = (b ∉ A & { } = A ∩ B)
by blast
lemma [simp]: ({ } = A ∩ (B ∪ C)) = ({ } = A ∩ B & { } = A ∩ C)
by blast

```

```

lemma VARS hd tl p q r s
  {List tl p Ps ∧ List tl q Qs ∧ set Ps ∩ set Qs = { } ∧
   (p ≠ Null ∨ q ≠ Null)}
  IF cor (q = Null) (cand (p ≠ Null) (p^.hd ≤ q^.hd))
  THEN r := p; p := p^.tl ELSE r := q; q := q^.tl FI;

```

```

s := r;
WHILE p ≠ Null ∨ q ≠ Null
INV {EX rs ps qs a. Path tl r rs s ∧ List tl p ps ∧ List tl q qs ∧
    distinct(a # ps @ qs @ rs) ∧ s = Ref a ∧
    merge(Ps, Qs, λx y. hd x ≤ hd y) =
    rs @ a # merge(ps, qs, λx y. hd x ≤ hd y) ∧
    (tl a = p ∨ tl a = q)}
DO IF cor (q = Null) (cand (p ≠ Null) (p^.hd ≤ q^.hd))
    THEN s^.tl := p; p := p^.tl ELSE s^.tl := q; q := q^.tl FI;
s := s^.tl
OD
{List tl r (merge(Ps, Qs, λx y. hd x ≤ hd y))}
apply vcg-simp
apply (simp-all add: cand-def cor-def)

apply (fastsimp)

apply clarsimp
apply(rule conjI)
apply clarsimp
apply(rule conjI)
apply (fastsimp intro!:Path-snoc intro:Path-upd[THEN iffD2] notin-List-update[THEN
iffD2] simp:eq-sym-conv)
apply clarsimp
apply(rule conjI)
apply (clarsimp)
apply(rule-tac x = rs @ [a] in exI)
apply(clarsimp simp:eq-sym-conv)
apply(rule-tac x = bs in exI)
apply(clarsimp simp:eq-sym-conv)
apply(rule-tac x = ya#bsa in exI)
apply(simp)
apply(clarsimp simp:eq-sym-conv)
apply(rule-tac x = rs @ [a] in exI)
apply(clarsimp simp:eq-sym-conv)
apply(rule-tac x = y#bs in exI)
apply(clarsimp simp:eq-sym-conv)
apply(rule-tac x = bsa in exI)
apply(simp)
apply (fastsimp intro!:Path-snoc intro:Path-upd[THEN iffD2] notin-List-update[THEN
iffD2] simp:eq-sym-conv)

apply(clarsimp simp add:List-app)
done

```

And now with ghost variables:

```

lemma VARS elem next p q r s ps qs rs a
{List next p Ps ∧ List next q Qs ∧ set Ps ∩ set Qs = {} ∧
(p ≠ Null ∨ q ≠ Null) ∧ ps = Ps ∧ qs = Qs}

```

```

IF cor (q = Null) (cand (p ≠ Null) (p^.elem ≤ q^.elem))
THEN r := p; p := p^.next; ps := tl ps
ELSE r := q; q := q^.next; qs := tl qs FI;
s := r; rs := []; a := addr s;
WHILE p ≠ Null ∨ q ≠ Null
INV {Path next r rs s ∧ List next p ps ∧ List next q qs ∧
    distinct(a # ps @ qs @ rs) ∧ s = Ref a ∧
    merge(Ps,Qs,λx y. elem x ≤ elem y) =
    rs @ a # merge(ps,qs,λx y. elem x ≤ elem y) ∧
    (next a = p ∨ next a = q)}
DO IF cor (q = Null) (cand (p ≠ Null) (p^.elem ≤ q^.elem))
THEN s^.next := p; p := p^.next; ps := tl ps
ELSE s^.next := q; q := q^.next; qs := tl qs FI;
rs := rs @ [a]; s := s^.next; a := addr s
OD
{List next r (merge(Ps,Qs,λx y. elem x ≤ elem y))}
apply vcg-simp
apply (simp-all add: cand-def cor-def)

apply (fastsimp)

apply clarsimp
apply(rule conjI)
apply(clarsimp)
apply(rule conjI)
apply(clarsimp simp:neq-commute)
apply(clarsimp simp:neq-commute)
apply(clarsimp simp:neq-commute)

apply(clarsimp simp add:List-app)
done

```

The proof is a LOT simpler because it does not need instantiations anymore, but it is still not quite automatic, probably because of this wrong orientation business.

More of the previous proof without ghost variables can be automated, but the runtime goes up drastically. In general it is usually more efficient to give the witness directly than to have it found by proof.

Now we try a functional version of the abstraction relation *Path*. Since the result is not that convincing, we do not prove any of the lemmas.

```

consts ispath:: ('a ⇒ 'a ref) ⇒ 'a ref ⇒ 'a ref ⇒ bool
      path:: ('a ⇒ 'a ref) ⇒ 'a ref ⇒ 'a ref ⇒ 'a list

```

First some basic lemmas:

```

lemma [simp]: ispath f p p
by (rule unproven)
lemma [simp]: path f p p = []
by (rule unproven)

```

lemma [simp]: $\text{ispath } f \ p \ q \implies a \notin \text{set}(\text{path } f \ p \ q) \implies \text{ispath } (f(a := r)) \ p \ q$
by (rule unproven)
lemma [simp]: $\text{ispath } f \ p \ q \implies a \notin \text{set}(\text{path } f \ p \ q) \implies$
 $\text{path } (f(a := r)) \ p \ q = \text{path } f \ p \ q$
by (rule unproven)

Some more specific lemmas needed by the example:

lemma [simp]: $\text{ispath } (f(a := q)) \ p \ (\text{Ref } a) \implies \text{ispath } (f(a := q)) \ p \ q$
by (rule unproven)
lemma [simp]: $\text{ispath } (f(a := q)) \ p \ (\text{Ref } a) \implies$
 $\text{path } (f(a := q)) \ p \ q = \text{path } (f(a := q)) \ p \ (\text{Ref } a) \ @ \ [a]$
by (rule unproven)
lemma [simp]: $\text{ispath } f \ p \ (\text{Ref } a) \implies f \ a = \text{Ref } b \implies$
 $b \notin \text{set}(\text{path } f \ p \ (\text{Ref } a))$
by (rule unproven)
lemma [simp]: $\text{ispath } f \ p \ (\text{Ref } a) \implies f \ a = \text{Null} \implies \text{islist } f \ p$
by (rule unproven)
lemma [simp]: $\text{ispath } f \ p \ (\text{Ref } a) \implies f \ a = \text{Null} \implies \text{list } f \ p = \text{path } f \ p \ (\text{Ref } a) \ @ \ [a]$
by (rule unproven)
lemma [simp]: $\text{islist } f \ p \implies \text{distinct } (\text{list } f \ p)$
by (rule unproven)

lemma VARS $hd \ tl \ p \ q \ r \ s$
 $\{ \text{islist } tl \ p \ \& \ Ps = \text{list } tl \ p \ \wedge \ \text{islist } tl \ q \ \& \ Qs = \text{list } tl \ q \ \wedge$
 $\text{set } Ps \cap \text{set } Qs = \{ \} \ \wedge$
 $(p \neq \text{Null} \vee q \neq \text{Null}) \}$
 $IF \text{ cor } (q = \text{Null}) \ (\text{cand } (p \neq \text{Null}) \ (p.^{hd} \leq q.^{hd}))$
 $THEN \ r := p; \ p := p.^{tl} \ ELSE \ r := q; \ q := q.^{tl} \ FI;$
 $s := r;$
 $WHILE \ p \neq \text{Null} \vee q \neq \text{Null}$
 $INV \ \{ EX \ rs \ ps \ qs \ a. \ \text{ispath } tl \ r \ s \ \& \ rs = \text{path } tl \ r \ s \ \wedge$
 $\text{islist } tl \ p \ \& \ ps = \text{list } tl \ p \ \wedge \ \text{islist } tl \ q \ \& \ qs = \text{list } tl \ q \ \wedge$
 $\text{distinct}(a \ \# \ ps \ @ \ qs \ @ \ rs) \ \wedge \ s = \text{Ref } a \ \wedge$
 $\text{merge}(Ps, Qs, \lambda x \ y. \ hd \ x \leq hd \ y) =$
 $rs \ @ \ a \ \# \ \text{merge}(ps, qs, \lambda x \ y. \ hd \ x \leq hd \ y) \ \wedge$
 $(tl \ a = p \vee tl \ a = q) \}$
 $DO \ IF \text{ cor } (q = \text{Null}) \ (\text{cand } (p \neq \text{Null}) \ (p.^{hd} \leq q.^{hd}))$
 $THEN \ s.^{tl} := p; \ p := p.^{tl} \ ELSE \ s.^{tl} := q; \ q := q.^{tl} \ FI;$
 $s := s.^{tl}$
 OD
 $\{ \text{islist } tl \ r \ \& \ \text{list } tl \ r = (\text{merge}(Ps, Qs, \lambda x \ y. \ hd \ x \leq hd \ y)) \}$
apply vcg-simp

apply (simp-all add: cand-def cor-def)
apply (fastsimp)
apply (fastsimp simp: eq-sym-conv)
apply (clarsimp)

done

The proof is automatic, but requires a number of special lemmas.

0.4.5 Cyclic list reversal

We consider two algorithms for the reversal of circular lists.

lemma *circular-list-rev-I:*

```

  VARS next root p q tmp
  {root = Ref r ∧ distPath next root (r#Ps) root}
  p := root; q := root^.next;
  WHILE q ≠ root
  INV {∃ ps qs. distPath next p ps root ∧ distPath next q qs root ∧
      root = Ref r ∧ r ∉ set Ps ∧ set ps ∩ set qs = {} ∧
      Ps = (rev ps) @ qs }
  DO tmp := q; q := q^.next; tmp^.next := p; p:=tmp OD;
  root^.next := p
  { root = Ref r ∧ distPath next root (r#rev Ps) root}
apply (simp only:distPath-def)
apply vcg-simp
  apply (rule-tac x=[] in exI)
  apply auto
  apply (drule (2) neq-dP)
  apply clarsimp
  apply(rule-tac x=a # ps in exI)
apply clarsimp
done

```

In the beginning, we are able to assert *distPath next root as root*, with *as* set to [] or [r, a, b, c]. Note that *Path next root as root* would additionally give us an infinite number of lists with the recurring sequence [r, a, b, c].

The precondition states that there exists a non-empty non-repeating path $r \# Ps$ from pointer *root* to itself, given that *root* points to location *r*. Pointers *p* and *q* are then set to *root* and the successor of *root* respectively. If $q = root$, we have circled the loop, otherwise we set the *next* pointer field of *q* to point to *p*, and shift *p* and *q* one step forward. The invariant thus states that *p* and *q* point to two disjoint lists *ps* and *qs*, such that $Ps = rev\ ps \ @ \ qs$. After the loop terminates, one extra step is needed to close the loop. As expected, the postcondition states that the *distPath* from *root* to itself is now $r \# rev\ Ps$.

It may come as a surprise to the reader that the simple algorithm for acyclic list reversal, with modified annotations, works for cyclic lists as well:

lemma *circular-list-rev-II:*

```

  VARS next p q tmp
  {p = Ref r ∧ distPath next p (r#Ps) p}
  q:=Null;
  WHILE p ≠ Null

```

```

INV
{ ((q = Null)  $\longrightarrow$  ( $\exists ps. distPath\ next\ p\ (ps)\ (Ref\ r) \wedge ps = r\#\ Ps$ ))  $\wedge$ 
  ((q  $\neq$  Null)  $\longrightarrow$  ( $\exists ps\ qs. distPath\ next\ q\ (qs)\ (Ref\ r) \wedge List\ next\ p\ ps \wedge$ 
     $set\ ps \cap set\ qs = \{\}$   $\wedge rev\ qs\ @\ ps = Ps@[r]$ ))  $\wedge$ 
   $\neg (p = Null \wedge q = Null)$  }
DO tmp := p; p := p^.next; tmp^.next := q; q:=tmp OD
{q = Ref r  $\wedge distPath\ next\ q\ (r\ \# rev\ Ps)\ q$ }
apply (simp only:distPath-def)
apply vcg-simp
  apply clarsimp
  apply clarsimp
apply (case-tac (q = Null))
  apply (fastsimp intro: Path-is-List)
apply clarsimp
apply (rule-tac x= bs in exI)
apply (rule-tac x= y # qs in exI)
apply clarsimp
apply (auto simp:fun-upd-apply)
done

```

0.4.6 Storage allocation

```

constdefs new :: 'a set  $\Rightarrow$  'a
new A == SOME a. a  $\notin$  A

```

```

lemma new-notin:
 $\llbracket \sim finite(UNIV::'a\ set); finite(A::'a\ set); B \subseteq A \rrbracket \Longrightarrow new\ (A) \notin B$ 
apply(unfold new-def)
apply(rule someI2-ex)
  apply (fast intro:ex-new-if-finite)
apply (fast)
done

```

```

lemma  $\sim finite(UNIV::'a\ set) \Longrightarrow$ 
  VARS xs elem next alloc p q
  {Xs = xs  $\wedge p = (Null::'a\ ref)$ }
  WHILE xs  $\neq []$ 
  INV {islist next p  $\wedge set(list\ next\ p) \subseteq set\ alloc \wedge$ 
     $map\ elem\ (rev(list\ next\ p))\ @\ xs = Xs$ }
  DO q := Ref(new(set alloc)); alloc := (addr q)#alloc;
    q^.next := p; q^.elem := hd xs; xs := tl xs; p := q
  OD
  {islist next p  $\wedge map\ elem\ (rev(list\ next\ p)) = Xs$ }
apply vcg-simp
apply (clarsimp simp: subset-insert-iff neq-Nil-conv fun-upd-apply new-notin)
apply fastsimp
done

```

end

theory *HeapSyntaxAbort* **imports** *HoareAbort Heap* **begin**

0.4.7 Field access and update

Heap update $p \hat{.} h := e$ is now guarded against p being Null. However, p may still be illegal, e.g. uninitialized or dangling. To guard against that, one needs a more detailed model of the heap where allocated and free addresses are distinguished, e.g. by making the heap a map, or by carrying the set of free addresses around. This is needed anyway as soon as we want to reason about storage allocation/deallocation.

syntax

```

refupdate :: ('a ⇒ 'b) ⇒ 'a ref ⇒ 'b ⇒ ('a ⇒ 'b)
  (-/'((- → -)') [1000,0] 900)
@fassign :: 'a ref => id => 'v => 's com
  ((2-^.- := / -) [70,1000,65] 61)
@faccess :: 'a ref => ('a ref ⇒ 'v) => 'v
  (-^.- [65,1000] 65)

```

translations

```

refupdate f r v == f(addr r := v)
p^f := e => (p ≠ Null) → (f := refupdate f p e)
p^f      => f(addr p)

```

declare *fun-upd-apply*[simp del] *fun-upd-same*[simp] *fun-upd-other*[simp]

An example due to Suzuki:

lemma *VARs* $v\ n$

```

{w = Ref w0 & x = Ref x0 & y = Ref y0 & z = Ref z0 &
  distinct[w0,x0,y0,z0]}
w^v := (1::int); w^n := x;
x^v := 2; x^n := y;
y^v := 3; y^n := z;
z^v := 4; x^n := z
{w^n.n^n.v = 4}

```

by *vcg-simp*

end

theory *Pointer-ExamplesAbort* **imports** *HeapSyntaxAbort* **begin**

0.5 Verifications

0.5.1 List reversal

Interestingly, this proof is the same as for the unguarded program:

```

lemma VARs tl p q r
  {List tl p Ps  $\wedge$  List tl q Qs  $\wedge$  set Ps  $\cap$  set Qs = {}}
  WHILE p  $\neq$  Null
  INV { $\exists$  ps qs. List tl p ps  $\wedge$  List tl q qs  $\wedge$  set ps  $\cap$  set qs = {}  $\wedge$ 
    rev ps @ qs = rev Ps @ Qs}
  DO r := p; (p  $\neq$  Null  $\rightarrow$  p := p^.tl); r^.tl := q; q := r OD
  {List tl q (rev Ps @ Qs)}
apply vcg-simp
apply fastsimp
apply(fastsimp intro:notin-List-update[THEN iffD2])
apply fastsimp
done

end

```

```

theory SchorrWaite imports HeapSyntax begin

```

0.6 Machinery for the Schorr-Waite proof

constdefs

```

— Relations induced by a mapping
rel :: ('a  $\Rightarrow$  'a ref)  $\Rightarrow$  ('a  $\times$  'a) set
rel m == {(x,y). m x = Ref y}
relS :: ('a  $\Rightarrow$  'a ref) set  $\Rightarrow$  ('a  $\times$  'a) set
relS M == ( $\bigcup$  m  $\in$  M. rel m)
addrs :: 'a ref set  $\Rightarrow$  'a set
addrs P == {a. Ref a  $\in$  P}
reachable :: ('a  $\times$  'a) set  $\Rightarrow$  'a ref set  $\Rightarrow$  'a set
reachable r P == (r* “ addrs P)

```

lemmas *rel-defs* = *relS-def rel-def*

Rewrite rules for relations induced by a mapping

```

lemma self-reachable: b  $\in$  B  $\Longrightarrow$  b  $\in$  R* “ B
apply blast
done

```

```

lemma oneStep-reachable: b  $\in$  R“B  $\Longrightarrow$  b  $\in$  R* “ B
apply blast
done

```

lemma *still-reachable*: $\llbracket B \subseteq Ra^* \text{ `` } A; \forall (x,y) \in Rb - Ra. y \in (Ra^* \text{ `` } A) \rrbracket \implies Rb^* \text{ `` } A$
 $B \subseteq Ra^* \text{ `` } A$
apply (*clarsimp simp only: Image-iff intro: subsetI*)
apply (*erule rtrancl-induct*)
apply *blast*
apply (*subgoal-tac* $(y, z) \in Ra \cup (Rb - Ra)$)
apply (*erule UnE*)
apply (*auto intro: rtrancl-into-rtrancl*)
apply *blast*
done

lemma *still-reachable-eq*: $\llbracket A \subseteq Rb^* \text{ `` } B; B \subseteq Ra^* \text{ `` } A; \forall (x,y) \in Ra - Rb. y \in (Rb^* \text{ `` } B); \forall (x,y) \in Rb - Ra. y \in (Ra^* \text{ `` } A) \rrbracket \implies Ra^* \text{ `` } A = Rb^* \text{ `` } B$
apply (*rule equalityI*)
apply (*erule still-reachable ,assumption*)
done

lemma *reachable-null*: $\text{reachable } mS \ \{Null\} = \{\}$
apply (*simp add: reachable-def addr-def*)
done

lemma *reachable-empty*: $\text{reachable } mS \ \{\} = \{\}$
apply (*simp add: reachable-def addr-def*)
done

lemma *reachable-union*: $(\text{reachable } mS \ aS \cup \text{reachable } mS \ bS) = \text{reachable } mS \ (aS \cup bS)$
apply (*simp add: reachable-def rel-defs addr-def*)
apply *blast*
done

lemma *reachable-union-sym*: $\text{reachable } r \ (\text{insert } a \ aS) = (r^* \text{ `` } \text{addr } \{a\}) \cup \text{reachable } r \ aS$
apply (*simp add: reachable-def rel-defs addr-def*)
apply *blast*
done

lemma *rel-upd1*: $(a,b) \notin \text{rel } (r(q:=t)) \implies (a,b) \in \text{rel } r \implies a=q$
apply (*rule classical*)
apply (*simp add: rel-defs fun-upd-apply*)
done

lemma *rel-upd2*: $(a,b) \notin \text{rel } r \implies (a,b) \in \text{rel } (r(q:=t)) \implies a=q$
apply (*rule classical*)
apply (*simp add: rel-defs fun-upd-apply*)
done

constdefs
— Restriction of a relation

$restr :: ('a \times 'a) \text{ set} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow ('a \times 'a) \text{ set} \quad ((-/ | -) [50, 51] 50)$
 $restr\ r\ m == \{(x,y). (x,y) \in r \wedge \neg m\ x\}$

Rewrite rules for the restriction of a relation

lemma *restr-identity*[simp]:
 $(\forall x. \neg m\ x) \Longrightarrow (R | m) = R$
by (*auto simp add:restr-def*)

lemma *restr-rtrancl*[simp]: $\llbracket m\ l \rrbracket \Longrightarrow (R | m)^* \text{ `` } \{l\} = \{l\}$
by (*auto simp add:restr-def elim:converse-rtranclE*)

lemma [simp]: $\llbracket m\ l \rrbracket \Longrightarrow (l,x) \in (R | m)^* = (l=x)$
by (*auto simp add:restr-def elim:converse-rtranclE*)

lemma *restr-upd*: $((rel\ (r\ (q := t)))(m(q := \text{True}))) = ((rel\ (r))(m(q := \text{True})))$

apply (*auto simp:restr-def rel-def fun-upd-apply*)
apply (*rename-tac a b*)
apply (*case-tac a=q*)
apply *auto*
done

lemma *restr-un*: $((r \cup s)|m) = (r|m) \cup (s|m)$
by (*auto simp add:restr-def*)

lemma *rel-upd3*: $(a, b) \notin (r|(m(q := t))) \Longrightarrow (a,b) \in (r|m) \Longrightarrow a = q$
apply (*rule classical*)
apply (*simp add:restr-def fun-upd-apply*)
done

constdefs

— A short form for the stack mapping function for List
 $S :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a\ \text{ref}) \Rightarrow ('a \Rightarrow 'a\ \text{ref}) \Rightarrow ('a \Rightarrow 'a\ \text{ref})$
 $S\ c\ l\ r == (\lambda x. \text{if } c\ x \text{ then } r\ x \text{ else } l\ x)$

Rewrite rules for Lists using S as their mapping

lemma [rule-format,simp]:
 $\forall p. a \notin \text{set } stack \longrightarrow List\ (S\ c\ l\ r)\ p\ stack = List\ (S\ (c(a:=x))\ (l(a:=y))\ (r(a:=z)))\ p\ stack$
apply(*induct-tac stack*)
apply(*simp add:fun-upd-apply S-def*)
done

lemma [rule-format,simp]:
 $\forall p. a \notin \text{set } stack \longrightarrow List\ (S\ c\ l\ (r(a:=z)))\ p\ stack = List\ (S\ c\ l\ r)\ p\ stack$
apply(*induct-tac stack*)
apply(*simp add:fun-upd-apply S-def*)
done

lemma *[rule-format,simp]*:
 $\forall p. a \notin \text{set stack} \longrightarrow \text{List } (S \ c \ (l(a:=z)) \ r) \ p \ \text{stack} = \text{List } (S \ c \ l \ r) \ p \ \text{stack}$
apply (*induct-tac stack*)
apply (*simp add:fun-upd-apply S-def*) +
done

lemma *[rule-format,simp]*:
 $\forall p. a \notin \text{set stack} \longrightarrow \text{List } (S \ (c(a:=z)) \ l \ r) \ p \ \text{stack} = \text{List } (S \ c \ l \ r) \ p \ \text{stack}$
apply (*induct-tac stack*)
apply (*simp add:fun-upd-apply S-def*) +
done

consts

— Recursive definition of what it means for a the graph/stack structure to be reconstructible

$\text{stkOk} :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a \ \text{ref}) \Rightarrow ('a \Rightarrow 'a \ \text{ref}) \Rightarrow ('a \Rightarrow 'a \ \text{ref}) \Rightarrow ('a \Rightarrow 'a \ \text{ref}) \Rightarrow 'a \ \text{ref} \Rightarrow 'a \ \text{list} \Rightarrow \text{bool}$

primrec

$\text{stkOk-nil: } \text{stkOk } c \ l \ r \ iL \ iR \ t \ [] = \text{True}$

$\text{stkOk-cons: } \text{stkOk } c \ l \ r \ iL \ iR \ t \ (p \# \text{stk}) = (\text{stkOk } c \ l \ r \ iL \ iR \ (\text{Ref } p) \ (\text{stk}) \wedge$
 $iL \ p = (\text{if } c \ p \ \text{then } l \ p \ \text{else } t) \wedge$
 $iR \ p = (\text{if } c \ p \ \text{then } t \ \text{else } r \ p))$

Rewrite rules for `stkOk`

lemma *[simp]*: $\bigwedge t. \llbracket x \notin \text{set } xs; \text{Ref } x \neq t \rrbracket \Longrightarrow$
 $\text{stkOk } (c(x := f)) \ l \ r \ iL \ iR \ t \ xs = \text{stkOk } c \ l \ r \ iL \ iR \ t \ xs$
apply (*induct xs*)
apply (*auto simp:eq-sym-conv*)
done

lemma *[simp]*: $\bigwedge t. \llbracket x \notin \text{set } xs; \text{Ref } x \neq t \rrbracket \Longrightarrow$
 $\text{stkOk } c \ (l(x := g)) \ r \ iL \ iR \ t \ xs = \text{stkOk } c \ l \ r \ iL \ iR \ t \ xs$
apply (*induct xs*)
apply (*auto simp:eq-sym-conv*)
done

lemma *[simp]*: $\bigwedge t. \llbracket x \notin \text{set } xs; \text{Ref } x \neq t \rrbracket \Longrightarrow$
 $\text{stkOk } c \ l \ (r(x := g)) \ iL \ iR \ t \ xs = \text{stkOk } c \ l \ r \ iL \ iR \ t \ xs$
apply (*induct xs*)
apply (*auto simp:eq-sym-conv*)
done

lemma *stkOk-r-rewrite [simp]*: $\bigwedge x. x \notin \text{set } xs \Longrightarrow$
 $\text{stkOk } c \ l \ (r(x := g)) \ iL \ iR \ (\text{Ref } x) \ xs = \text{stkOk } c \ l \ r \ iL \ iR \ (\text{Ref } x) \ xs$
apply (*induct xs*)
apply (*auto simp:eq-sym-conv*)
done

lemma *[simp]*: $\bigwedge x. x \notin \text{set } xs \Longrightarrow$

```

  stkOk c (l(x := g)) r iL iR (Ref x) xs = stkOk c l r iL iR (Ref x) xs
apply (induct xs)
apply (auto simp: eq-sym-conv)
done

```

```

lemma [simp]:  $\bigwedge x. x \notin \text{set } xs \implies$ 
  stkOk (c(x := g)) l r iL iR (Ref x) xs = stkOk c l r iL iR (Ref x) xs
apply (induct xs)
apply (auto simp: eq-sym-conv)
done

```

0.7 The Schorr-Waite algorithm

theorem *SchorrWaiteAlgorithm*:

```

VARs c m l r t p q root
{R = reachable (relS {l, r}) {root}  $\wedge (\forall x. \neg m\ x) \wedge iR = r \wedge iL = l$ }
t := root; p := Null;
WHILE p  $\neq$  Null  $\vee$  t  $\neq$  Null  $\wedge \neg t.^m$ 
INV { $\exists$  stack.
  List (S c l r) p stack  $\wedge$  (*i1*)
  ( $\forall x \in \text{set stack}. m\ x$ )  $\wedge$  (*i2*)
  R = reachable (relS {l, r}) {t, p}  $\wedge$  (*i3*)
  ( $\forall x. x \in R \wedge \neg m\ x \longrightarrow$  (*i4*)
    x  $\in$  reachable (relS {l, r} | m) ({t}  $\cup$  set (map r stack)))  $\wedge$ 
  ( $\forall x. m\ x \longrightarrow x \in R$ )  $\wedge$  (*i5*)
  ( $\forall x. x \notin \text{set stack} \longrightarrow r\ x = iR\ x \wedge l\ x = iL\ x$ )  $\wedge$  (*i6*)
  (stkOk c l r iL iR t stack) (*i7*) }
DO IF t = Null  $\vee$  t.^m
  THEN IF p.^c
    THEN q := t; t := p; p := p.^r; t.^r := q (*pop*)
    ELSE q := t; t := p.^r; p.^r := p.^l; (*swing*)
      p.^l := q; p.^c := True FI
  ELSE q := p; p := t; t := t.^l; p.^l := q; (*push*)
    p.^m := True; p.^c := False FI OD
{( $\forall x. (x \in R) = m\ x$ )  $\wedge$  (r = iR  $\wedge$  l = iL) }
(is VARs c m l r t p q root {?Pre c m l r root} (?c1; ?c2; ?c3) {?Post c m l r})
proof (vcg)
let While {(c, m, l, r, t, p, q, root). ?whileB m t p}
  {(c, m, l, r, t, p, q, root). ?inv c m l r t p} ?body = ?c3
{

  fix c m l r t p q root
  assume ?Pre c m l r root
  thus ?inv c m l r root Null by (auto simp add: reachable-def addrs-def)
next

  fix c m l r t p q
  let  $\exists$  stack. ?Inv stack = ?inv c m l r t p
  assume a: ?inv c m l r t p  $\wedge \neg$ (p  $\neq$  Null  $\vee$  t  $\neq$  Null  $\wedge \neg t.^m$ )

```


then obtain *stack* **where** *inv*: ?*Inv stack* **by** *blast*
from *a* **have** *pNull*: $p = \text{Null}$ **and** *tDisj*: $t = \text{Null} \vee (t \neq \text{Null} \wedge t^\wedge.m)$ **by** *auto*
let $?I1 \wedge - \wedge - \wedge ?I4 \wedge ?I5 \wedge ?I6 \wedge - = ?Inv\ stack$
from *inv* **have** *i1*: ?*I1* **and** *i4*: ?*I4* **and** *i5*: ?*I5* **and** *i6*: ?*I6* **by** *simp+*
from *pNull i1* **have** *stackEmpty*: *stack* = [] **by** *simp*
from *tDisj i4* **have** *RisMarked*[*rule-format*]: $\forall x. x \in R \longrightarrow m\ x$ **by** (*auto simp: reachable-def addrs-def stackEmpty*)
from *i5 i6* **show** $(\forall x.(x \in R) = m\ x) \wedge r = iR \wedge l = iL$ **by** (*auto simp: stackEmpty expand-fun-eq intro:RisMarked*)

next

fix *c m l r t p q root*
let $\exists\ stack. ?Inv\ stack = ?inv\ c\ m\ l\ r\ t\ p$
let $\exists\ stack. ?popInv\ stack = ?inv\ c\ m\ l\ (r(p \rightarrow t))\ p\ (p^\wedge.r)$
let $\exists\ stack. ?swInv\ stack =$
 $?inv\ (c(p \rightarrow \text{True}))\ m\ (l(p \rightarrow t))\ (r(p \rightarrow p^\wedge.l))\ (p^\wedge.r)\ p$
let $\exists\ stack. ?puInv\ stack =$
 $?inv\ (c(t \rightarrow \text{False}))\ (m(t \rightarrow \text{True}))\ (l(t \rightarrow p))\ r\ (t^\wedge.l)\ t$
let $?ifB1 = (t = \text{Null} \vee t^\wedge.m)$
let $?ifB2 = p^\wedge.c$

assume $(\exists\ stack. ?Inv\ stack) \wedge (p \neq \text{Null} \vee t \neq \text{Null} \wedge \neg t^\wedge.m)$ (**is** - \wedge *?whileB*)

then obtain *stack* **where** *inv*: ?*Inv stack* **and** *whileB*: ?*whileB* **by** *blast*
let $?I1 \wedge ?I2 \wedge ?I3 \wedge ?I4 \wedge ?I5 \wedge ?I6 \wedge ?I7 = ?Inv\ stack$
from *inv* **have** *i1*: ?*I1* **and** *i2*: ?*I2* **and** *i3*: ?*I3* **and** *i4*: ?*I4*
and *i5*: ?*I5* **and** *i6*: ?*I6* **and** *i7*: ?*I7* **by** *simp+*
have *stackDist*: *distinct* (*stack*) **using** *i1* **by** (*rule List-distinct*)

show $(?ifB1 \longrightarrow (?ifB2 \longrightarrow (\exists\ stack. ?popInv\ stack))) \wedge$
 $(\neg ?ifB2 \longrightarrow (\exists\ stack. ?swInv\ stack)) \wedge$
 $(\neg ?ifB1 \longrightarrow (\exists\ stack. ?puInv\ stack))$

proof –

{
assume *ifB1*: $t = \text{Null} \vee t^\wedge.m$ **and** *ifB2*: $p^\wedge.c$
from *ifB1 whileB* **have** *pNotNull*: $p \neq \text{Null}$ **by** *auto*
then obtain *addr-p* **where** *addr-p-eq*: $p = \text{Ref}\ \text{addr-p}$ **by** *auto*
with *i1* **obtain** *stack-tl* **where** *stack-eq*: $\text{stack} = (\text{addr}\ p) \# \text{stack-tl}$
by *auto*
with *i2* **have** *m-addr-p*: $p^\wedge.m$ **by** *auto*
have *stackDist*: *distinct* (*stack*) **using** *i1* **by** (*rule List-distinct*)
from *stack-eq stackDist* **have** *p-notin-stack-tl*: $\text{addr}\ p \notin \text{set}\ \text{stack-tl}$ **by**

simp

let $?poI1 \wedge ?poI2 \wedge ?poI3 \wedge ?poI4 \wedge ?poI5 \wedge ?poI6 \wedge ?poI7 = ?popInv\ \text{stack-tl}$
have *?popInv stack-tl*
proof –

— List property is maintained:
from *i1 p-notin-stack-tl ifB2*

```

have poI1: List (S c l (r(p → t))) (p^.r) stack-tl
  by(simp add: addr-p-eq stack-eq, simp add: S-def)

moreover
— Everything on the stack is marked:
from i2 have poI2: ∀ x ∈ set stack-tl. m x by (simp add:stack-eq)
moreover

— Everything is still reachable:
let (R = reachable ?Ra ?A) = ?I3
let ?Rb = (relS {l, r(p → t)})
let ?B = {p, p^.r}
— Our goal is R = reachable ?Rb ?B.
have ?Ra* “addrs ?A = ?Rb* “addrs ?B (is ?L = ?R)
proof
  show ?L ⊆ ?R
  proof (rule still-reachable)
    show addrs ?A ⊆ ?Rb* “addrs ?B by(fastsimp simp:addrs-def relS-def
rel-def addr-p-eq
intro:oneStep-reachable Image-iff[THEN iffD2])
    show ∀(x,y) ∈ ?Ra-?Rb. y ∈ (?Rb* “addrs ?B) by (clarsimp
simp:relS-def)
(fastsimp simp add:rel-def Image-iff addrs-def dest:rel-upd1)
  qed
  show ?R ⊆ ?L
  proof (rule still-reachable)
    show addrs ?B ⊆ ?Ra* “addrs ?A
    by(fastsimp simp:addrs-def rel-defs addr-p-eq
intro:oneStep-reachable Image-iff[THEN iffD2])
  next
    show ∀(x, y) ∈ ?Rb-?Ra. y ∈ (?Ra* “addrs ?A)
    by (clarsimp simp:relS-def)
(fastsimp simp add:rel-def Image-iff addrs-def dest:rel-upd2)
  qed
qed
with i3 have poI3: R = reachable ?Rb ?B by (simp add:reachable-def)
moreover

— If it is reachable and not marked, it is still reachable using...
let ∀x. x ∈ R ∧ ¬ m x → x ∈ reachable ?Ra ?A = ?I4
let ?Rb = relS {l, r(p → t)} | m
let ?B = {p} ∪ set (map (r(p → t)) stack-tl)
— Our goal is ∀x. x ∈ R ∧ ¬ m x → x ∈ reachable ?Rb ?B.
let ?T = {t, p^.r}

have ?Ra* “addrs ?A ⊆ ?Rb* “(addrs ?B ∪ addrs ?T)
proof (rule still-reachable)
  have rewrite: ∀ s ∈ set stack-tl. (r(p → t)) s = r s
  by (auto simp add:p-notin-stack-tl intro:fun-upd-other)

```

show $addr\ ?A \subseteq ?Rb^* \text{ `` } (addr\ ?B \cup addr\ ?T)$
by (*fastsimp cong:map-cong simp:stack-eq addr-def rewrite intro:self-reachable*)
show $\forall (x, y) \in ?Ra - ?Rb. y \in (?Rb^* \text{ `` } (addr\ ?B \cup addr\ ?T))$
by (*clarsimp simp:restr-def relS-def*)
(fastsimp simp add:rel-def Image-iff addr-def dest:rel-upd1)
qed
— We now bring a term from the right to the left of the subset relation.
hence $subset: ?Ra^* \text{ `` } addr\ ?A - ?Rb^* \text{ `` } addr\ ?T \subseteq ?Rb^* \text{ `` } addr\ ?B$
by *blast*
have $poI4: \forall x. x \in R \wedge \neg m\ x \longrightarrow x \in reachable\ ?Rb\ ?B$
proof (*rule allI, rule impI*)
fix x
assume $a: x \in R \wedge \neg m\ x$
— First, a disjunction on r ($addr\ p$) used later in the proof
have $pDisj: p.^r = Null \vee (p.^r \neq Null \wedge p.^r.^m)$ **using** $poI1\ poI2$
by *auto*
— x belongs to the left hand side of *subset*:
have $incl: x \in ?Ra^* \text{ `` } addr\ ?A$ **using** $a\ i4$ **by** (*simp only:reachable-def,clarsimp*)
have $excl: x \notin ?Rb^* \text{ `` } addr\ ?T$ **using** $pDisj\ ifB1\ a$ **by** (*auto simp add:addr-def*)
— And therefore also belongs to the right hand side of *subset*,
— which corresponds to our goal.
from $incl\ excl\ subset$ **show** $x \in reachable\ ?Rb\ ?B$ **by** (*auto simp add:reachable-def*)
qed
moreover
— If it is marked, then it is reachable
from $i5$ **have** $poI5: \forall x. m\ x \longrightarrow x \in R$.
moreover
— If it is not on the stack, then its l and r fields are unchanged
from $i7\ i6\ ifB2$
have $poI6: \forall x. x \notin set\ stack\text{-}tl \longrightarrow (r(p \rightarrow t))\ x = iR\ x \wedge l\ x = iL\ x$
by (*auto simp: addr-p-eq stack-eq fun-upd-apply*)
moreover
— If it is on the stack, then its l and r fields can be reconstructed
from $p\text{-notin-stack-tl}\ i7$ **have** $poI7: stkOk\ c\ l\ (r(p \rightarrow t))\ iL\ iR\ p\ stack\text{-}tl$
by (*clarsimp simp:stack-eq addr-p-eq*)
ultimately show $?popInv\ stack\text{-}tl$ **by** *simp*
qed
hence $\exists stack. ?popInv\ stack\ ..$
}
moreover

— Proofs of the Swing and Push arm follow.
 — Since they are in principle simmilar to the Pop arm proof,
 — we show fewer comments and use frequent pattern matching.

```

{
  — Swing arm
  assume ifB1: ?ifB1 and nifB2: ¬?ifB2
  from ifB1 whileB have pNotNull: p ≠ Null by clarsimp
  then obtain addr-p where addr-p-eq: p = Ref addr-p by clarsimp
  with i1 obtain stack-tl where stack-eq: stack = (addr p) # stack-tl by
clarsimp
  with i2 have m-addr-p: p ^ .m by clarsimp
  from stack-eq stackDist have p-notin-stack-tl: (addr p) ∉ set stack-tl
    by simp
  let ?swI1 ∧ ?swI2 ∧ ?swI3 ∧ ?swI4 ∧ ?swI5 ∧ ?swI6 ∧ ?swI7 = ?swInv stack
  have ?swInv stack
  proof —

    — List property is maintained:
    from i1 p-notin-stack-tl nifB2
    have swI1: ?swI1
      by (simp add:addr-p-eq stack-eq, simp add:S-def)
    moreover

    — Everything on the stack is marked:
    from i2
    have swI2: ?swI2 .
    moreover

    — Everything is still reachable:
    let R = reachable ?Ra ?A = ?I3
    let R = reachable ?Rb ?B = ?swI3
    have ?Ra* “ addrs ?A = ?Rb* “ addrs ?B
    proof (rule still-reachable-eq)
      show addrs ?A ⊆ ?Rb* “ addrs ?B
      by(fastsimp simp:addrs-def rel-defs addr-p-eq intro:oneStep-reachable
Image-iff[THEN iffD2])
    next
      show addrs ?B ⊆ ?Ra* “ addrs ?A
      by(fastsimp simp:addrs-def rel-defs addr-p-eq intro:oneStep-reachable
Image-iff[THEN iffD2])
    next
      show ∀ (x, y) ∈ ?Ra − ?Rb. y ∈ (?Rb* “ addrs ?B)
      by (clarsimp simp:relS-def) (fastsimp simp add:rel-def Image-iff
addrs-def fun-upd-apply dest:rel-upd1)
    next
      show ∀ (x, y) ∈ ?Rb − ?Ra. y ∈ (?Ra* “ addrs ?A)
      by (clarsimp simp:relS-def) (fastsimp simp add:rel-def Image-iff
addrs-def fun-upd-apply dest:rel-upd2)
  
```

qed
with $i3$
have $swI3$: $?swI3$ **by** ($simp$ $add:reachable-def$)
moreover

— If it is reachable and not marked, it is still reachable using...
let $\forall x. x \in R \wedge \neg m\ x \longrightarrow x \in reachable\ ?Ra\ ?A = ?I4$
let $\forall x. x \in R \wedge \neg m\ x \longrightarrow x \in reachable\ ?Rb\ ?B = ?swI4$
let $?T = \{t\}$
have $?Ra^* \text{“} \textit{addrs}\ ?A \subseteq ?Rb^* \text{“} (\textit{addrs}\ ?B \cup \textit{addrs}\ ?T)$
proof ($rule\ still-reachable$)
 have $rewrite: (\forall s \in set\ stack-tl. (r(addr\ p := l(addr\ p)))\ s = r\ s)$
 by ($auto\ simp\ add:p-notin-stack-tl\ intro:fun-upd-other$)
 show $\textit{addrs}\ ?A \subseteq ?Rb^* \text{“} (\textit{addrs}\ ?B \cup \textit{addrs}\ ?T)$
 by ($fastsimp\ cong:map-cong\ simp:stack-eq\ \textit{addrs-def}\ rewrite\ intro:self-reachable$)
 next
 show $\forall (x, y) \in ?Ra - ?Rb. y \in ?Rb^* \text{“} (\textit{addrs}\ ?B \cup \textit{addrs}\ ?T)$
 by ($clarsimp\ simp:relS-def\ restr-def$) ($fastsimp\ simp\ add:rel-def\ Image-iff\ \textit{addrs-def}\ fun-upd-apply\ dest:rel-upd1$)
qed
then have $subset: ?Ra^* \text{“} \textit{addrs}\ ?A - ?Rb^* \text{“} \textit{addrs}\ ?T \subseteq ?Rb^* \text{“} \textit{addrs}\ ?B$
 by $blast$
have $?swI4$
proof ($rule\ allI, rule\ impI$)
 fix x
 assume $a: x \in R \wedge \neg m\ x$
 with $i4\ \textit{addr-p-eq}\ \textit{stack-eq}$ **have** $inc: x \in ?Ra^* \text{“} \textit{addrs}\ ?A$
 by ($simp\ only:reachable-def, clarsimp$)
 with $ifB1\ a$
 have $exc: x \notin ?Rb^* \text{“} \textit{addrs}\ ?T$
 by ($auto\ simp\ add:\textit{addrs-def}$)
 from $inc\ exc\ subset$ **show** $x \in reachable\ ?Rb\ ?B$
 by ($auto\ simp\ add:reachable-def$)
qed
moreover

— If it is marked, then it is reachable
from $i5$
have $?swI5$.
moreover

— If it is not on the stack, then its l and r fields are unchanged
from $i6\ stack-eq$
have $?swI6$
 by $clarsimp$
moreover

— If it is on the stack, then its l and r fields can be reconstructed

```

    from stackDist i7 nifB2
    have ?swI7
    by (clarsimp simp:addr-p-eq stack-eq)

    ultimately show ?thesis by auto
  qed
  then have  $\exists$  stack. ?swInv stack by blast
}
moreover

{
  — Push arm
  assume nifB1:  $\neg ?ifB1$ 
  from nifB1 whileB have tNotNull:  $t \neq \text{Null}$  by clarsimp
  then obtain addr-t where addr-t-eq:  $t = \text{Ref } \text{addr-t}$  by clarsimp
  with i1 obtain new-stack where new-stack-eq:  $\text{new-stack} = (\text{addr } t) \#$ 
stack by clarsimp
  from tNotNull nifB1 have n-m-addr-t:  $\neg (t \hat{.} m)$  by clarsimp
  with i2 have t-notin-stack:  $(\text{addr } t) \notin \text{set stack}$  by blast
  let ?puI1  $\wedge$  ?puI2  $\wedge$  ?puI3  $\wedge$  ?puI4  $\wedge$  ?puI5  $\wedge$  ?puI6  $\wedge$  ?puI7 = ?puInv new-stack
  have ?puInv new-stack
  proof —

    — List property is maintained:
    from i1 t-notin-stack
    have puI1: ?puI1
    by (simp add:addr-t-eq new-stack-eq, simp add:S-def)
    moreover

    — Everything on the stack is marked:
    from i2
    have puI2: ?puI2
    by (simp add:new-stack-eq fun-upd-apply)
    moreover

    — Everything is still reachable:
    let R = reachable ?Ra ?A = ?I3
    let R = reachable ?Rb ?B = ?puI3
    have ?Ra* “adds ?A = ?Rb* “adds ?B
    proof (rule still-reachable-eq)
      show adds ?A  $\subseteq$  ?Rb* “adds ?B
      by (fastsimp simp:adds-def rel-defs addr-t-eq intro:oneStep-reachable
Image-iff[THEN iffD2])
    next
      show adds ?B  $\subseteq$  ?Ra* “adds ?A
      by (fastsimp simp:adds-def rel-defs addr-t-eq intro:oneStep-reachable
Image-iff[THEN iffD2])
    next
      show  $\forall (x, y) \in ?Ra - ?Rb. y \in (?Rb^* \text{ “ “ } \text{adds } ?B)$ 

```



```

    — If it is not on the stack, then its  $l$  and  $r$  fields are unchanged
    from  $i6$ 
    have ?puI6
      by (simp add:new-stack-eq)
    moreover

    — If it is on the stack, then its  $l$  and  $r$  fields can be reconstructed
    from  $stackDist\ i6\ t-notin-stack\ i7$ 
    have ?puI7 by (clarsimp simp:addr-t-eq new-stack-eq)

    ultimately show ?thesis by auto
  qed
then have  $\exists\ stack. ?puInv\ stack$  by blast
}
ultimately show ?thesis by blast
qed
}
qed
end

```

```

theory SepLogHeap
imports Main
begin

```

```

types heap = (nat  $\Rightarrow$  nat option)

```

Some means allocated, *None* means free. Address 0 serves as the null reference.

0.7.1 Paths in the heap

```

consts
  Path :: heap  $\Rightarrow$  nat  $\Rightarrow$  nat list  $\Rightarrow$  nat  $\Rightarrow$  bool
primrec
  Path h x [] y = (x = y)
  Path h x (a#as) y = (x  $\neq$  0  $\wedge$  a=x  $\wedge$  ( $\exists$  b. h x = Some b  $\wedge$  Path h b as y))

lemma [iff]: Path h 0 xs y = (xs = []  $\wedge$  y = 0)
by (cases xs) simp-all

lemma [simp]: x  $\neq$  0  $\implies$  Path h x as z =
  (as = []  $\wedge$  z = x  $\vee$  ( $\exists$  y bs. as = x#bs  $\wedge$  h x = Some y  $\&$  Path h y bs z))
by (cases as) auto

```


lemma [simp]: $\bigwedge x. \text{Path } f \ x \ (as@bs) \ z = (\exists y. \text{Path } f \ x \ as \ y \wedge \text{Path } f \ y \ bs \ z)$
by (induct as) auto

lemma Path-upd[simp]:
 $\bigwedge x. u \notin \text{set } as \implies \text{Path } (f(u := v)) \ x \ as \ y = \text{Path } f \ x \ as \ y$
by (induct as) simp-all

0.7.2 Lists on the heap

constdefs
 $List :: \text{heap} \Rightarrow \text{nat} \Rightarrow \text{nat list} \Rightarrow \text{bool}$
 $List \ h \ x \ as == \text{Path } h \ x \ as \ 0$

lemma [simp]: $List \ h \ x \ [] = (x = 0)$
by (simp add: List-def)

lemma [simp]:
 $List \ h \ x \ (a\#as) = (x \neq 0 \wedge a = x \wedge (\exists y. h \ x = \text{Some } y \wedge List \ h \ y \ as))$
by (simp add: List-def)

lemma [simp]: $List \ h \ 0 \ as = (as = [])$
by (cases as) simp-all

lemma List-non-null: $a \neq 0 \implies$
 $List \ h \ a \ as = (\exists b \ bs. as = a\#bs \wedge h \ a = \text{Some } b \wedge List \ h \ b \ bs)$
by (cases as) simp-all

theorem notin-List-update[simp]:
 $\bigwedge x. a \notin \text{set } as \implies List \ (h(a := y)) \ x \ as = List \ h \ x \ as$
by (induct as) simp-all

lemma List-unique: $\bigwedge x \ bs. List \ h \ x \ as \implies List \ h \ x \ bs \implies as = bs$
by (induct as) (auto simp add: List-non-null)

lemma List-unique1: $List \ h \ p \ as \implies \exists! as. List \ h \ p \ as$
by (blast intro: List-unique)

lemma List-app: $\bigwedge x. List \ h \ x \ (as@bs) = (\exists y. \text{Path } h \ x \ as \ y \wedge List \ h \ y \ bs)$
by (induct as) auto

lemma List-hd-not-in-tl[simp]: $List \ h \ b \ as \implies h \ a = \text{Some } b \implies a \notin \text{set } as$
apply (clarsimp simp add: in-set-conv-decomp)
apply (frule List-app[THEN iffD1])
apply (fastsimp dest: List-unique)
done

lemma List-distinct[simp]: $\bigwedge x. List \ h \ x \ as \implies \text{distinct } as$
by (induct as) (auto dest: List-hd-not-in-tl)

lemma *list-in-heap*: $\bigwedge p. \text{List } h \ p \ ps \implies \text{set } ps \subseteq \text{dom } h$
by (*induct ps*) *auto*

lemma *list-ortho-sum1*[*simp*]:
 $\bigwedge p. \llbracket \text{List } h1 \ p \ ps; \text{dom } h1 \cap \text{dom } h2 = \{\} \rrbracket \implies \text{List } (h1 ++ h2) \ p \ ps$
by (*induct ps*) (*auto simp add:map-add-def split:option.split*)

lemma *list-ortho-sum2*[*simp*]:
 $\bigwedge p. \llbracket \text{List } h2 \ p \ ps; \text{dom } h1 \cap \text{dom } h2 = \{\} \rrbracket \implies \text{List } (h1 ++ h2) \ p \ ps$
by (*induct ps*) (*auto simp add:map-add-def split:option.split*)

end

theory *Separation* **imports** *HoareAbort SepLogHeap* **begin**

The semantic definition of a few connectives:

constdefs

ortho:: *heap* \Rightarrow *heap* \Rightarrow *bool* (**infix** \perp 55)
h1 \perp *h2* == *dom h1* \cap *dom h2* = $\{\}$

is-empty :: *heap* \Rightarrow *bool*
is-empty h == *h* = *empty*

singl:: *heap* \Rightarrow *nat* \Rightarrow *nat* \Rightarrow *bool*
singl h x y == *dom h* = $\{x\}$ & *h x* = *Some y*

star:: (*heap* \Rightarrow *bool*) \Rightarrow (*heap* \Rightarrow *bool*) \Rightarrow (*heap* \Rightarrow *bool*)
star P Q == $\lambda h. \exists h1 \ h2. h = h1 ++ h2 \wedge h1 \perp h2 \wedge P \ h1 \wedge Q \ h2$

wand:: (*heap* \Rightarrow *bool*) \Rightarrow (*heap* \Rightarrow *bool*) \Rightarrow (*heap* \Rightarrow *bool*)
wand P Q == $\lambda h. \forall h'. h' \perp h \wedge P \ h' \longrightarrow Q(h ++ h')$

This is what assertions look like without any syntactic sugar:

lemma *VARs* *x y z w h*
 $\{ \text{star } (\%h. \text{singl } h \ x \ y) (\%h. \text{singl } h \ z \ w) \ h \}$
SKIP
 $\{ x \neq z \}$
apply *vcg*
apply(*auto simp:star-def ortho-def singl-def*)
done

Now we add nice input syntax. To suppress the heap parameter of the connectives, we assume it is always called *H* and add/remove it upon parsing/printing. Thus every pointer program needs to have a program variable *H*, and assertions should not contain any locally bound *Hs* - otherwise they may bind the implicit *H*.

syntax

```

@emp :: bool (emp)
@singl :: nat ⇒ nat ⇒ bool ([- ↦ -])
@star :: bool ⇒ bool ⇒ bool (infixl ** 60)
@wand :: bool ⇒ bool ⇒ bool (infixl -* 60)

```

ML

```

(* free-tr takes care of free vars in the scope of sep. logic connectives:
   they are implicitly applied to the heap *)
fun free-tr (t as Free -) = t $ Syntax.free H
(*)
| free-tr ((list as Free (List, -)) $ p $ ps) = list $ Syntax.free H $ p $ ps
*)
| free-tr t = t

fun emp-tr [] = Syntax.const is-empty $ Syntax.free H
  | emp-tr ts = raise TERM (emp-tr, ts);
fun singl-tr [p,q] = Syntax.const singl $ Syntax.free H $ p $ q
  | singl-tr ts = raise TERM (singl-tr, ts);
fun star-tr [P,Q] = Syntax.const star $
  absfree(H, dummyT, free-tr P) $ absfree(H, dummyT, free-tr Q) $
  Syntax.free H
  | star-tr ts = raise TERM (star-tr, ts);
fun wand-tr [P,Q] = Syntax.const wand $
  absfree(H, dummyT, P) $ absfree(H, dummyT, Q) $ Syntax.free H
  | wand-tr ts = raise TERM (wand-tr, ts);
»

```

parse-translation

```

« [ (@emp, emp-tr), (@singl, singl-tr),
    (@star, star-tr), (@wand, wand-tr) ] »

```

Now it looks much better:

```

lemma VARS H x y z w
  {[x↦y] ** [z↦w]}
  SKIP
  {x ≠ z}
apply vcg
apply (auto simp:star-def ortho-def singl-def)
done

lemma VARS H x y z w
  {emp ** emp}
  SKIP
  {emp}
apply vcg
apply (auto simp:star-def ortho-def is-empty-def)
done

```

But the output is still unreadable. Thus we also strip the heap parameters upon output:

```

ML⟨⟨
local
fun strip (Abs(-,-,(t as Const(-free,-) $ Free -) $ Bound 0)) = t
  | strip (Abs(-,-,(t as Free -) $ Bound 0)) = t
(*
  | strip (Abs(-,-,((list as Const(List,-))$ Bound 0 $ p $ ps))) = list$ps
*)
  | strip (Abs(-,-,(t as Const(-var,-) $ Var -) $ Bound 0)) = t
  | strip (Abs(-,-,P)) = P
  | strip (Const(is-empty,-)) = Syntax.const @emp
  | strip t = t;
in
fun is-empty-tr' [-] = Syntax.const @emp
fun singl-tr' [-,p,q] = Syntax.const @singl $ p $ q
fun star-tr' [P,Q,-] = Syntax.const @star $ strip P $ strip Q
fun wand-tr' [P,Q,-] = Syntax.const @wand $ strip P $ strip Q
end
⟩⟩

```

print-translation

```

⟨⟨ [(is-empty, is-empty-tr'),(singl, singl-tr'),
   (star, star-tr'),(wand, wand-tr')] ⟩⟩

```

Now the intermediate proof states are also readable:

```

lemma VARS H x y z w
{[x↦y] ** [z↦w]}
y := w
{x ≠ z}
apply vcg
apply(auto simp:star-def ortho-def singl-def)
done

lemma VARS H x y z w
{emp ** emp}
SKIP
{emp}
apply vcg
apply(auto simp:star-def ortho-def is-empty-def)
done

```

So far we have unfolded the separation logic connectives in proofs. Here comes a simple example of a program proof that uses a law of separation logic instead.

```

lemma star-comm: P ** Q = Q ** P
by(auto simp add:star-def ortho-def dest: map-add-comm)

```

```

lemma VARS H x y z w

```

```

{P ** Q}
SKIP
{Q ** P}
apply vcg
apply(simp add: star-comm)
done

```

```

lemma VARs H
  { $p \neq 0 \wedge [p \mapsto x] ** \text{List } H \ q \ qs$ }
   $H := H(p \mapsto q)$ 
  { $\text{List } H \ p \ (p \# qs)$ }
apply vcg
apply(simp add: star-def ortho-def singl-def)
apply clarify
apply(subgoal-tac p \notin set qs)
prefer 2
apply(blast dest:list-in-heap)
apply simp
done

```

```

lemma VARs H p q r
  { $\text{List } H \ p \ Ps ** \text{List } H \ q \ Qs$ }
  WHILE  $p \neq 0$ 
  INV { $\exists ps \ qs. (\text{List } H \ p \ ps ** \text{List } H \ q \ qs) \wedge \text{rev } ps @ qs = \text{rev } Ps @ Qs$ }
  DO  $r := p; p := \text{the}(H \ p); H := H(r \mapsto q); q := r \ OD$ 
  { $\text{List } H \ q \ (\text{rev } Ps @ Qs)$ }
apply vcg
apply(simp-all add: star-def ortho-def singl-def)

```

```

apply fastsimp

```

```

apply (clarsimp simp add:List-non-null)
apply(rename-tac ps')
apply(rule-tac x = ps' in exI)
apply(rule-tac x = p#qs in exI)
apply simp
apply(rule-tac x = h1(p:=None) in exI)
apply(rule-tac x = h2(p\mapsto q) in exI)
apply simp
apply(rule conjI)
apply(rule ext)
apply(simp add:map-add-def split:option.split)
apply(rule conjI)
apply blast
apply(simp add:map-add-def split:option.split)
apply(rule conjI)
apply(subgoal-tac p \notin set qs)
prefer 2

```

```
    apply(blast dest:list-in-heap)  
  apply(simp)  
  apply fast  
  
  apply(fastsimp)  
done  
  
end
```

Bibliography

- [1] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, volume 2741 of *LNCS*, pages 121–135. Springer, 2003.
- [2] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005.