

# Hoare Logic

Various

May 22, 2012

### **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 . . . . .	10
0.0.2	References . . . . .	11
0.0.3	Field access and update . . . . .	11
0.1	The heap . . . . .	11
0.1.1	Paths in the heap . . . . .	11
0.1.2	Lists on the heap . . . . .	12
0.1.3	Functional abstraction . . . . .	13
0.2	Verifications . . . . .	13
0.2.1	List reversal . . . . .	13
0.2.2	Searching in a list . . . . .	14
0.2.3	Merging two lists . . . . .	15
0.2.4	Storage allocation . . . . .	15
0.2.5	References . . . . .	16
0.3	The heap . . . . .	16
0.3.1	Paths in the heap . . . . .	16
0.3.2	Non-repeating paths . . . . .	17
0.3.3	Lists on the heap . . . . .	17
0.3.4	Functional abstraction . . . . .	18
0.3.5	Field access and update . . . . .	19
0.4	Verifications . . . . .	20
0.4.1	List reversal . . . . .	20
0.4.2	Searching in a list . . . . .	21
0.4.3	Splicing two lists . . . . .	22
0.4.4	Merging two lists . . . . .	22
0.4.5	Cyclic list reversal . . . . .	24
0.4.6	Storage allocation . . . . .	25
0.4.7	Field access and update . . . . .	26
0.5	Verifications . . . . .	27
0.5.1	List reversal . . . . .	27
0.6	Machinery for the Schorr-Waite proof . . . . .	27
0.7	The Schorr-Waite algorithm . . . . .	30
0.7.1	Paths in the heap . . . . .	31
0.7.2	Lists on the heap . . . . .	31

```

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

type-synonym 'a bexp = 'a set
type-synonym 'a assn = 'a set

datatype
  'a com = Basic 'a ⇒ '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)

abbreviation annskip (SKIP) where SKIP == Basic id

type-synonym 'a sem = 'a ⇒ 'a ⇒ bool

inductive Sem :: 'a com ⇒ 'a sem
where
  Sem (Basic f) s (f s)
| Sem c1 s s'' ⇒ Sem c2 s'' s' ⇒ Sem (c1;c2) s s'
| s ∈ b ⇒ Sem c1 s s' ⇒ Sem (IF b THEN c1 ELSE c2 FI) s s'
| s ∉ b ⇒ Sem c2 s s' ⇒ Sem (IF b THEN c1 ELSE c2 FI) s s'
| s ∉ b ⇒ Sem (While b x c) s s
| s ∈ b ⇒ Sem c s s'' ⇒ Sem (While b x c) s'' s' ⇒
  Sem (While b x c) s s'

inductive-cases [elim!]:
  Sem (Basic f) s s' Sem (c1;c2) s s'
  Sem (IF b THEN c1 ELSE c2 FI) s s'

definition Valid :: 'a bexp ⇒ 'a com ⇒ 'a bexp ⇒ bool
  where Valid p c q ⇔ (!s s'. Sem c s s' → s : p → s' : q)

syntax
  -assign :: idt ⇒ 'b ⇒ 'a com ((2- :=/ -) [70, 65] 61)

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)

```

$\langle ML \rangle$

**lemma** *SkipRule*:  $p \subseteq q \implies \text{Valid } p \text{ (Basic id) } q$   
 $\langle \text{proof} \rangle$

**lemma** *BasicRule*:  $p \subseteq \{s. f s \in q\} \implies \text{Valid } p \text{ (Basic f) } q$   
 $\langle \text{proof} \rangle$

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

**lemma** *CondRule*:  
 $p \subseteq \{s. (s \in b \implies s \in w) \wedge (s \notin b \implies s \in w')\}$   
 $\implies \text{Valid } w \text{ } c1 \text{ } q \implies \text{Valid } w' \text{ } c2 \text{ } q \implies \text{Valid } p \text{ (Cond } b \text{ } c1 \text{ } c2 \text{)} q$   
 $\langle \text{proof} \rangle$

**lemma** *While-aux*:  
**assumes** *Sem* (*WHILE* *b INV*  $\{i\}$  *DO* *c OD*) *s s'*  
**shows**  $\forall s s'. \text{Sem } c \text{ } s \text{ } s' \implies s \in I \wedge s \in b \implies s' \in I \implies$   
 $s \in I \implies s' \in I \wedge s' \notin b$   
 $\langle \text{proof} \rangle$

**lemma** *WhileRule*:  
 $p \subseteq i \implies \text{Valid } (i \cap b) \text{ } c \text{ } i \implies i \cap (-b) \subseteq q \implies \text{Valid } p \text{ (While } b \text{ } i \text{ } c \text{)} q$   
 $\langle \text{proof} \rangle$

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

**lemmas** *AbortRule* = *SkipRule* — dummy version  
 $\langle ML \rangle$

**end**

**theory** *Arith2*  
**imports** *Main*  
**begin**

**definition** *cd* ::  $[nat, nat, nat] \implies bool$   
**where**  $cd \ x \ m \ n \longleftrightarrow x \ dvd \ m \ \& \ x \ dvd \ n$

**definition** *gcd* ::  $[nat, nat] \implies nat$   
**where**  $gcd \ m \ n = (\text{SOME } x. cd \ x \ m \ n \ \& \ (!y. (cd \ y \ m \ n) \implies y \leq x))$

**primrec** *fac* ::  $nat \implies nat$

**where**

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

**cd**

**lemma** *cd-nnn*:  $0 < n \implies cd\ n\ n\ n$   
*<proof>*

**lemma** *cd-le*:  $[| cd\ x\ m\ n; 0 < m; 0 < n |] \implies x \leq m \ \& \ x \leq n$   
*<proof>*

**lemma** *cd-swap*:  $cd\ x\ m\ n = cd\ x\ n\ m$   
*<proof>*

**lemma** *cd-diff-l*:  $n \leq m \implies cd\ x\ m\ n = cd\ x\ (m-n)\ n$   
*<proof>*

**lemma** *cd-diff-r*:  $m \leq n \implies cd\ x\ m\ n = cd\ x\ m\ (n-m)$   
*<proof>*

**gcd**

**lemma** *gcd-nnn*:  $0 < n \implies n = gcd\ n\ n$   
*<proof>*

**lemma** *gcd-swap*:  $gcd\ m\ n = gcd\ n\ m$   
*<proof>*

**lemma** *gcd-diff-l*:  $n \leq m \implies gcd\ m\ n = gcd\ (m-n)\ n$   
*<proof>*

**lemma** *gcd-diff-r*:  $m \leq n \implies gcd\ m\ n = gcd\ m\ (n-m)$   
*<proof>*

**pow**

**lemma** *sq-pow-div2* [*simp*]:  
 $m \bmod 2 = 0 \implies ((n::nat)*n)^{(m \div 2)} = n^m$   
*<proof>*

**end**

**theory** *Examples* **imports** *Hoare-Logic Arith2* **begin**

**lemma multiply-by-add:** *VARs*  $m\ s\ a\ b$

$\{a=A \ \& \ b=B\}$   
 $m := 0; s := 0;$   
*WHILE*  $m \sim = a$   
*INV*  $\{s=m*b \ \& \ a=A \ \& \ b=B\}$   
*DO*  $s := s+b; m := m+(1::nat)$  *OD*  
 $\{s = A*B\}$   
*<proof>*

**lemma VARs**  $M\ N\ P :: int$

$\{m=M \ \& \ n=N\}$   
*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*  
 $\{P = m*n\}$   
*<proof>*

**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*  
 $\{a = gcd\ A\ B\}$   
*<proof>*

**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*  
 $\{a = gcd\ A\ B \ \& \ 2*A*B = a*(x+y)\}$   
*<proof>*

**lemma power-by-mult:** *VARs*  $a\ b\ c$

```

{a=A & b=B}
c := (1::nat);
WHILE b ~ = 0
INV {A^B = c * a^b}
DO WHILE b mod 2 = 0
  INV {A^B = c * a^b}
  DO a := a*a; b := b div 2 OD;
  c := c*a; b := b - 1
OD
{c = A^B}
<proof>

```

**lemma factorial:** VARS a b

```

{a=A}
b := 1;
WHILE a ~ = 0
INV {fac A = b * fac a}
DO b := b*a; a := a - 1 OD
{b = fac A}
<proof>

```

**lemma [simp]:**  $1 \leq i \implies \text{fac } (i - \text{Suc } 0) * i = \text{fac } i$

<proof>

**lemma VARS i f**

```

{True}
i := (1::nat); f := 1;
WHILE i <= n INV {f = fac(i - 1) & 1 <= i & i <= n+1}
DO f := f*i; i := i+1 OD
{f = fac n}
<proof>

```

**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)}
<proof>

```

**lemma** *sqrt-without-multiplication*: VARS  $u\ w\ r\ x$   
 $\{True\}$   
 $x := X; u := 1; w := 1; r := (0::nat);$   
**WHILE**  $w \leq x$   
**INV**  $\{u = r+r+1 \ \& \ w = (r+1)*(r+1) \ \& \ r*r \leq x \ \& \ x=X\}$   
**DO**  $r := r + 1; w := w + u + 2; u := u + 2$  **OD**  
 $\{r*r \leq X \ \& \ X < (r+1)*(r+1)\}$   
 $\langle proof \rangle$

**lemma** *imperative-reverse*: VARS  $y\ x$   
 $\{x=X\}$   
 $y := [];$   
**WHILE**  $x \sim = []$   
**INV**  $\{rev(x)@y = rev(X)\}$   
**DO**  $y := (hd\ x \ \# \ y); x := tl\ x$  **OD**  
 $\{y=rev(X)\}$   
 $\langle proof \rangle$

**lemma** *imperative-append*: VARS  $x\ y$   
 $\{x=X \ \& \ y=Y\}$   
 $x := rev(x);$   
**WHILE**  $x \sim = []$   
**INV**  $\{rev(x)@y = X@Y\}$   
**DO**  $y := (hd\ x \ \# \ y);$   
 $x := tl\ x$   
**OD**  
 $\{y = X@Y\}$   
 $\langle proof \rangle$

**lemma** *zero-search*: VARS  $A\ i$   
 $\{True\}$   
 $i := 0;$   
**WHILE**  $i < length\ A \ \& \ A!i \sim = key$   
**INV**  $\{!j. j < i \ \longrightarrow \ A!j \sim = key\}$   
**DO**  $i := i+1$  **OD**  
 $\{(i < length\ A \ \longrightarrow \ A!i = key) \ \& \$   
 $(i = length\ A \ \longrightarrow \ (!j. j < length\ A \ \longrightarrow \ A!j \sim = key))\}$   
 $\langle proof \rangle$

**lemma** *lem*:  $m - Suc\ 0 < n \implies m < Suc\ n$   
 $\langle proof \rangle$

```

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

```

*<proof>*

**end**

```

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

```

```

type-synonym 'a bexp = 'a set
type-synonym 'a assn = 'a set

```

```

datatype
  'a com = Basic 'a => 'a
  | Abort
  | 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)

```

```

abbreviation annskip (SKIP) where SKIP == Basic id

```

```

type-synonym 'a sem = 'a option => 'a option => bool

```

```

inductive Sem :: 'a com => 'a sem

```

**where**

```

  Sem (Basic f) None None
| Sem (Basic f) (Some s) (Some (f s))

```

| *Sem Abort*  $s$  *None*  
 | *Sem*  $c1$   $s$   $s'' \implies \text{Sem } c2$   $s''$   $s' \implies \text{Sem } (c1;c2)$   $s$   $s'$   
 | *Sem* (*IF*  $b$  *THEN*  $c1$  *ELSE*  $c2$  *FI*) *None* *None*  
 |  $s \in b \implies \text{Sem } c1$  (*Some*  $s$ )  $s' \implies \text{Sem } (IF$   $b$  *THEN*  $c1$  *ELSE*  $c2$  *FI*) (*Some*  $s$ )  
 $s'$   
 |  $s \notin b \implies \text{Sem } c2$  (*Some*  $s$ )  $s' \implies \text{Sem } (IF$   $b$  *THEN*  $c1$  *ELSE*  $c2$  *FI*) (*Some*  $s$ )  
 $s'$   
 | *Sem* (*While*  $b$   $x$   $c$ ) *None* *None*  
 |  $s \notin b \implies \text{Sem } (While$   $b$   $x$   $c$ ) (*Some*  $s$ ) (*Some*  $s$ )  
 |  $s \in b \implies \text{Sem } c$  (*Some*  $s$ )  $s'' \implies \text{Sem } (While$   $b$   $x$   $c$ )  $s''$   $s' \implies$   
 $\text{Sem } (While$   $b$   $x$   $c$ ) (*Some*  $s$ )  $s'$

**inductive-cases** [*elim!*]:

$\text{Sem } (Basic$   $f$ )  $s$   $s' \text{ Sem } (c1;c2)$   $s$   $s'$   
 $\text{Sem } (IF$   $b$  *THEN*  $c1$  *ELSE*  $c2$  *FI*)  $s$   $s'$

**definition** *Valid* ::  $'a$  *bexp*  $\implies 'a$  *com*  $\implies 'a$  *bexp*  $\implies bool$  **where**

*Valid*  $p$   $c$   $q \implies \forall s$   $s'$ .  $\text{Sem } c$   $s$   $s' \longrightarrow s : \text{Some } 'a$   $p \longrightarrow s' : \text{Some } 'a$   $q$

**syntax**

*-assign* :: *idt*  $\implies 'b \implies 'a$  *com* ((*2-* *:=/* *-*) [*70*, *65*] *61*)

**syntax**

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

**syntax (output)**

*-hoare-abort* :: [ $'a$  *assn*,  $'a$  *com*,  $'a$  *assn*]  $\implies bool$   
 ( $\{-\}$  *//* *- //*  $\{-\}$  [*0*,*55*,*0*] *50*)

$\langle ML \rangle$

**lemma** *SkipRule*:  $p \subseteq q \implies \text{Valid } p$  (*Basic* *id*)  $q$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *CondRule*:

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

$\langle proof \rangle$

**lemma** *While-aux*:

**assumes** *Sem* (*WHILE* *b INV* {*i*} *DO* *c OD*) *s s'*

**shows**  $\forall s s'. \text{Sem } c \ s \ s' \longrightarrow s \in \text{Some } 'I \cap b \longrightarrow s' \in \text{Some } 'I \implies$   
 $s \in \text{Some } 'I \implies s' \in \text{Some } '(I \cap \neg b)$

*<proof>*

**lemma** *WhileRule*:

$p \subseteq i \implies \text{Valid } (i \cap b) \ c \ i \implies i \cap (\neg b) \subseteq q \implies \text{Valid } p \ (\text{While } b \ i \ c) \ q$   
*<proof>*

**lemma** *AbortRule*:  $p \subseteq \{s. \text{False}\} \implies \text{Valid } p \ \text{Abort } q$

*<proof>*

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

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

*<proof>*

*<ML>*

**syntax**

*-guarded-com* ::  $\text{bool} \Rightarrow 'a \ \text{com} \Rightarrow 'a \ \text{com} \ ((\text{?} \rightarrow / \text{-}) \ 71)$

*-array-update* ::  $'a \ \text{list} \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a \ \text{com} \ ((\text{?}[\text{-}] \ := / \text{-}) \ [70, 65] \ 61)$

**translations**

$P \rightarrow c \ == \ \text{IF } P \ \text{THEN } c \ \text{ELSE } \text{CONST } \text{Abort } FI$

$a[i] := v \ == (i < \text{CONST } \text{length } a) \rightarrow (a := \text{CONST } \text{list-update } a \ i \ v)$

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** *Hoare-Logic-Abort* **begin**

**lemma** *VARS*  $x \ y \ z :: \text{nat}$

$\{y = z \ \& \ z \neq 0\} \ z \neq 0 \rightarrow x := y \ \text{div } z \ \{x = 1\}$

*<proof>*

**lemma**

*VARS*  $a \ i \ j$

$\{k \leq \text{length } a \ \& \ i < k \ \& \ j < k\} \ j < \text{length } a \rightarrow a[i] := a!j \ \{\text{True}\}$

*<proof>*

**lemma** *VARS*  $(a :: \text{int } \text{list}) \ i$

$\{\text{True}\}$

```

i := 0;
WHILE i < length a
INV {i <= length a}
DO a[i] := 7; i := i+1 OD
{True}
⟨proof⟩

```

**end**

**theory** *Pointers0* **imports** *Hoare-Logic* **begin**

## 0.0.2 References

```

class ref =
  fixes Null :: 'a

```

## 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 := CONST 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}
⟨proof⟩

```

## 0.1 The heap

### 0.1.1 Paths in the heap

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

```

  Path h x [] y = (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*)

$\langle proof \rangle$

**lemma** [simp]:  $a \neq \text{Null} \implies \text{Path } h \ a \ as \ z =$   
 $(as = [] \wedge z = a \vee (\exists bs. as = a\#bs \wedge \text{Path } h \ (h \ a) \ bs \ z))$   
 $\langle proof \rangle$

**lemma** [simp]:  $\bigwedge x. \text{Path } f \ x \ (as@bs) \ z = (\exists y. \text{Path } f \ x \ as \ y \wedge \text{Path } f \ y \ bs \ z)$   
 $\langle proof \rangle$

**lemma** [simp]:  $\bigwedge x. u \notin \text{set } as \implies \text{Path } (f(u := v)) \ x \ as \ y = \text{Path } f \ x \ as \ y$   
 $\langle proof \rangle$

## 0.1.2 Lists on the heap

### Relational abstraction

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

**lemma** [simp]:  $List \ h \ x \ [] = (x = \text{Null})$   
 $\langle proof \rangle$

**lemma** [simp]:  $List \ h \ x \ (a\#as) = (x \neq \text{Null} \wedge x = a \wedge List \ h \ (h \ a) \ as)$   
 $\langle proof \rangle$

**lemma** [simp]:  $List \ h \ \text{Null} \ as = (as = [])$   
 $\langle proof \rangle$

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

**theorem**  $notin\text{-List}\text{-update}$ [simp]:  
 $\bigwedge x. a \notin \text{set } as \implies List \ (h(a := y)) \ x \ as = List \ h \ x \ as$   
 $\langle proof \rangle$

**declare**  $fun\text{-upd}\text{-apply}$ [simp]  $del$   $fun\text{-upd}\text{-same}$ [simp]  $fun\text{-upd}\text{-other}$ [simp]

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

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

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

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

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

### 0.1.3 Functional abstraction

**definition** *islist* ::  $('a::\text{ref} \Rightarrow 'a) \Rightarrow 'a \Rightarrow \text{bool}$   
 where  $\text{islist } h \ p \longleftrightarrow (\exists as. \text{List } h \ p \ as)$

**definition** *list* ::  $('a::\text{ref} \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \ \text{list}$   
 where  $\text{list } h \ p = (\text{SOME } as. \text{List } h \ p \ as)$

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

**lemma** [simp]:  $\text{islist } h \ \text{Null}$   
 ⟨proof⟩

**lemma** [simp]:  $a \neq \text{Null} \implies \text{islist } h \ a = \text{islist } h \ (h \ a)$   
 ⟨proof⟩

**lemma** [simp]:  $\text{list } h \ \text{Null} = []$   
 ⟨proof⟩

**lemma** *list-Ref-conv*[simp]:  
 $\llbracket a \neq \text{Null}; \text{islist } h \ (h \ a) \rrbracket \implies \text{list } h \ a = a \ \# \ \text{list } h \ (h \ a)$   
 ⟨proof⟩

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

**lemma** *list-upd-conv*[simp]:  
 $\text{islist } h \ p \implies y \notin \text{set}(\text{list } h \ p) \implies \text{list } (h(y := q)) \ p = \text{list } h \ p$   
 ⟨proof⟩

**lemma** *islist-upd*[simp]:  
 $\text{islist } h \ p \implies y \notin \text{set}(\text{list } h \ p) \implies \text{islist } (h(y := q)) \ p$   
 ⟨proof⟩

## 0.2 Verifications

### 0.2.1 List reversal

A short but unreadable proof:

**lemma** *VARs*  $tl \ p \ q \ r$   
 $\{ \text{List } tl \ p \ Ps \ \wedge \ \text{List } tl \ q \ Qs \ \wedge \ \text{set } Ps \ \cap \ \text{set } Qs = \{ \} \}$   
 $\text{WHILE } p \neq \text{Null}$   
 $\text{INV } \{ \exists ps \ qs. \ \text{List } tl \ p \ ps \ \wedge \ \text{List } tl \ q \ qs \ \wedge \ \text{set } ps \ \cap \ \text{set } qs = \{ \} \ \wedge$   
 $\quad \text{rev } ps \ @ \ qs = \text{rev } Ps \ @ \ Qs \}$

$DO\ r := p; p := p^.tl; r^.tl := q; q := r\ OD$   
 $\{List\ tl\ q\ (rev\ Ps\ @\ Qs)\}$   
 <proof>

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>

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

**lemma**  $VARS\ tl\ p\ q\ r$   
 $\{islist\ tl\ p \wedge islist\ tl\ q \wedge$   
 $Ps = list\ tl\ p \wedge Qs = list\ tl\ q \wedge set\ Ps \cap set\ Qs = \{\}\}$   
 $WHILE\ p \neq Null$   
 $INV\ \{islist\ tl\ p \wedge islist\ tl\ q \wedge$   
 $set(list\ tl\ p) \cap set(list\ tl\ q) = \{\} \wedge$   
 $rev(list\ tl\ p)\ @\ (list\ tl\ q) = rev\ Ps\ @\ Qs\}$   
 $DO\ r := p; p := p^.tl; r^.tl := q; q := r\ OD$   
 $\{islist\ tl\ q \wedge list\ tl\ q = rev\ Ps\ @\ Qs\}$   
 <proof>

### 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**  $VARS\ tl\ p$   
 $\{List\ tl\ p\ Ps \wedge X \in set\ Ps\}$   
 $WHILE\ p \neq Null \wedge p \neq X$   
 $INV\ \{p \neq Null \wedge (\exists ps.\ List\ tl\ p\ ps \wedge X \in set\ ps)\}$   
 $DO\ p := p^.tl\ OD$   
 $\{p = X\}$   
 <proof>

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

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

*<proof>*

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*  
  $\{(p,X) \in \{(x,y). y = tl\ x \ \& \ x \neq Null\}^*\}$   
 *WHILE*  $p \neq Null \wedge p \neq X$   
 *INV*  $\{(p,X) \in \{(x,y). y = tl\ x \ \& \ x \neq Null\}^*\}$   
 *DO*  $p := p.^{tl}$  *OD*  
  $\{p = X\}$   
*<proof>*

### 0.2.3 Merging two lists

This is still a bit rough, especially the proof.

**fun** *merge* :: 'a list \* 'a list \* ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list **where**  
*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*:  $(P|Q \longrightarrow R) = ((P \longrightarrow R) \wedge (\sim P \longrightarrow Q \longrightarrow R))$   
*<proof>*

**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*  
  $\{List\ tl\ r\ (merge(Ps, Qs, \lambda x\ y. hd\ x \leq hd\ y))\}$   
*<proof>*

### 0.2.4 Storage allocation

**definition** *new* :: 'a set  $\Rightarrow$  'a::ref

**where**  $\text{new } A = (\text{SOME } a. a \notin A \ \& \ a \neq \text{Null})$

**lemma** *new-notin*:

$\llbracket \sim \text{finite}(\text{UNIV}::('a::\text{ref})\text{set}); \text{finite}(A::'a\ \text{set}); B \subseteq A \rrbracket \implies$   
 $\text{new } (A) \notin B \ \& \ \text{new } A \neq \text{Null}$   
 $\langle \text{proof} \rangle$

**lemma**  $\sim \text{finite}(\text{UNIV}::('a::\text{ref})\text{set}) \implies$

$\text{VARS } xs \ \text{elem } \text{next } \text{alloc } p \ q$   
 $\{Xs = xs \ \wedge \ p = (\text{Null}::'a)\}$   
 $\text{WHILE } xs \neq []$   
 $\text{INV } \{ \text{islist } \text{next } p \ \wedge \ \text{set}(\text{list } \text{next } p) \subseteq \text{set } \text{alloc} \ \wedge$   
 $\quad \text{map } \text{elem } (\text{rev}(\text{list } \text{next } p)) \ @ \ xs = Xs \}$   
 $\text{DO } q := \text{new}(\text{set } \text{alloc}); \ \text{alloc} := q\#\text{alloc};$   
 $\quad q.\text{next} := p; \ q.\text{elem} := \text{hd } xs; \ xs := \text{tl } xs; \ p := q$   
 $\text{OD}$   
 $\{ \text{islist } \text{next } p \ \wedge \ \text{map } \text{elem } (\text{rev}(\text{list } \text{next } p)) = Xs \}$   
 $\langle \text{proof} \rangle$

**end**

**theory** *Heap* **imports** *Main* **begin**

### 0.2.5 References

**datatype**  $'a \ \text{ref} = \text{Null} \ | \ \text{Ref } 'a$

**lemma** *not-Null-eq* [iff]:  $(x \sim = \text{Null}) = (\text{EX } y. x = \text{Ref } y)$   
 $\langle \text{proof} \rangle$

**lemma** *not-Ref-eq* [iff]:  $(\text{ALL } y. x \sim = \text{Ref } y) = (x = \text{Null})$   
 $\langle \text{proof} \rangle$

**primrec** *addr* ::  $'a \ \text{ref} \Rightarrow 'a$  **where**  
 $\text{addr } (\text{Ref } a) = a$

## 0.3 The heap

### 0.3.1 Paths in the heap

**primrec** *Path* ::  $('a \Rightarrow 'a \ \text{ref}) \Rightarrow 'a \ \text{ref} \Rightarrow 'a \ \text{list} \Rightarrow 'a \ \text{ref} \Rightarrow \text{bool}$  **where**  
 $\text{Path } h \ x \ [] \ y \longleftrightarrow x = y$   
 $| \ \text{Path } h \ x \ (a\#\text{as}) \ y \longleftrightarrow x = \text{Ref } a \ \wedge \ \text{Path } h \ (h \ a) \ \text{as } y$

**lemma** [iff]:  $\text{Path } h \ \text{Null } xs \ y = (xs = [] \ \wedge \ y = \text{Null})$

$\langle proof \rangle$

**lemma** [simp]:  $Path\ h\ (Ref\ a)\ as\ z =$   
 $(as = [] \wedge z = Ref\ a \vee (\exists bs. as = a\#\!bs \wedge Path\ h\ (h\ a)\ bs\ z))$   
 $\langle proof \rangle$

**lemma** [simp]:  $\bigwedge x. Path\ f\ x\ (as@bs)\ z = (\exists y. Path\ f\ x\ as\ y \wedge Path\ f\ y\ bs\ z)$   
 $\langle proof \rangle$

**lemma** Path-upd[simp]:  
 $\bigwedge x. u \notin set\ as \implies Path\ (f(u := v))\ x\ as\ y = Path\ f\ x\ as\ y$   
 $\langle proof \rangle$

**lemma** Path-snoc:  
 $Path\ (f(a := q))\ p\ as\ (Ref\ a) \implies Path\ (f(a := q))\ p\ (as\ @\ [a])\ q$   
 $\langle proof \rangle$

### 0.3.2 Non-repeating paths

**definition**  $distPath :: ('a \Rightarrow 'a\ ref) \Rightarrow 'a\ ref \Rightarrow 'a\ list \Rightarrow 'a\ ref \Rightarrow bool$   
**where**  $distPath\ h\ x\ as\ y \longleftrightarrow Path\ h\ x\ as\ y \wedge distinct\ as$

The term  $distPath\ h\ x\ as\ y$  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**  $neg-dP: p \neq q \implies Path\ h\ p\ Ps\ q \implies distinct\ Ps \implies$   
 $EX\ a\ Qs. p = Ref\ a \ \&\ Ps = a\#\!Qs \ \&\ a \notin set\ Qs$   
 $\langle proof \rangle$

**lemma**  $neg-dP-disp: \llbracket p \neq q; distPath\ h\ p\ Ps\ q \rrbracket \implies$   
 $EX\ a\ Qs. p = Ref\ a \wedge Ps = a\#\!Qs \wedge a \notin set\ Qs$   
 $\langle proof \rangle$

### 0.3.3 Lists on the heap

#### Relational abstraction

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

**lemma** [simp]:  $List\ h\ x\ [] = (x = Null)$   
 $\langle proof \rangle$

**lemma** [simp]:  $List\ h\ x\ (a\#\!as) = (x = Ref\ a \wedge List\ h\ (h\ a)\ as)$   
 $\langle proof \rangle$

**lemma** [simp]:  $List\ h\ Null\ as = (as = [])$   
 $\langle proof \rangle$

**lemma** *List-Ref[simp]*:  $List\ h\ (Ref\ a)\ as = (\exists\ bs.\ as = a\#\!bs \wedge List\ h\ (h\ a)\ bs)$   
 ⟨proof⟩

**theorem** *notin-List-update[simp]*:  
 $\bigwedge x.\ a \notin set\ as \implies List\ (h(a := y))\ x\ as = List\ h\ x\ as$   
 ⟨proof⟩

**lemma** *List-unique*:  $\bigwedge x\ bs.\ List\ h\ x\ as \implies List\ h\ x\ bs \implies as = bs$   
 ⟨proof⟩

**lemma** *List-unique1*:  $List\ h\ p\ as \implies \exists! as.\ List\ h\ p\ as$   
 ⟨proof⟩

**lemma** *List-app*:  $\bigwedge x.\ List\ h\ x\ (as@bs) = (\exists\ y.\ Path\ h\ x\ as\ y \wedge List\ h\ y\ bs)$   
 ⟨proof⟩

**lemma** *List-hd-not-in-tl[simp]*:  $List\ h\ (h\ a)\ as \implies a \notin set\ as$   
 ⟨proof⟩

**lemma** *List-distinct[simp]*:  $\bigwedge x.\ List\ h\ x\ as \implies distinct\ as$   
 ⟨proof⟩

**lemma** *Path-is-List*:  
 $\llbracket Path\ h\ b\ Ps\ (Ref\ a); a \notin set\ Ps \rrbracket \implies List\ (h(a := Null))\ b\ (Ps\ @\ [a])$   
 ⟨proof⟩

### 0.3.4 Functional abstraction

**definition** *islist* ::  $('a \Rightarrow 'a\ ref) \Rightarrow 'a\ ref \Rightarrow bool$   
 where  $islist\ h\ p \longleftrightarrow (\exists\ as.\ List\ h\ p\ as)$

**definition** *list* ::  $('a \Rightarrow 'a\ ref) \Rightarrow 'a\ ref \Rightarrow 'a\ list$   
 where  $list\ h\ p = (SOME\ as.\ List\ h\ p\ as)$

**lemma** *List-conv-islist-list*:  $List\ h\ p\ as = (islist\ h\ p \wedge as = list\ h\ p)$   
 ⟨proof⟩

**lemma** [simp]:  $islist\ h\ Null$   
 ⟨proof⟩

**lemma** [simp]:  $islist\ h\ (Ref\ a) = islist\ h\ (h\ a)$   
 ⟨proof⟩

**lemma** [simp]:  $list\ h\ Null = []$   
 ⟨proof⟩

**lemma** *list-Ref-conv[simp]*:  
 $islist\ h\ (h\ a) \implies list\ h\ (Ref\ a) = a\ \#\ list\ h\ (h\ a)$

*<proof>*

**lemma** [simp]:  $islist\ h\ (h\ a) \implies a \notin set(list\ h\ (h\ a))$   
*<proof>*

**lemma** list-upd-conv[simp]:  
 $islist\ h\ p \implies y \notin set(list\ h\ p) \implies list\ (h(y := q))\ p = list\ h\ p$   
*<proof>*

**lemma** islist-upd[simp]:  
 $islist\ h\ p \implies y \notin set(list\ h\ p) \implies islist\ (h(y := q))\ p$   
*<proof>*

**end**

**theory** HeapSyntax imports Hoare-Logic Heap begin

### 0.3.5 Field access and update

**syntax**

-refupdate ::  $(a \Rightarrow b) \Rightarrow a\ ref \Rightarrow b \Rightarrow (a \Rightarrow b)$   
(-/'((-  $\rightarrow$  -)') [1000,0] 900)  
-fassign ::  $a\ ref \Rightarrow id \Rightarrow v \Rightarrow s\ com$   
((2-^.- :=/ -) [70,1000,65] 61)  
-faccess ::  $a\ ref \Rightarrow (a\ ref \Rightarrow v) \Rightarrow v$   
(-^.- [65,1000] 65)

**translations**

$f(r \rightarrow v) == f(CONST\ addr\ r := v)$   
 $p^{\wedge}.f := e \Rightarrow f := f(p \rightarrow e)$   
 $p^{\wedge}.f \Rightarrow f(CONST\ addr\ p)$

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

An example due to Suzuki:

**lemma** VARS  $v\ n$

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

*<proof>*

**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*  
 $\{List\ tl\ p\ Ps\ \wedge\ List\ tl\ q\ Qs\ \wedge\ set\ Ps\ \cap\ set\ Qs\ =\ \{\}\}$   
*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>*

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*  
 $\{List\ next\ q\ (rev\ Ps\ @\ Qs)\}$   
*<proof>*

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>*

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*  $\{islist\ tl\ p\ \wedge\ islist\ tl\ q\ \wedge$   
 $set(list\ tl\ p) \cap set(list\ tl\ q) = \{\}\ \wedge$

$$\begin{array}{l}
\text{rev(list tl } p) \text{ @ (list tl } q) = \text{rev } Ps \text{ @ } Qs\} \\
DO \ r := p; \ p := p.^{.}tl; \ r.^{.}tl := q; \ q := r \ OD \\
\{islist \ tl \ q \wedge list \ tl \ q = \text{rev } Ps \text{ @ } Qs\} \\
\langle proof \rangle
\end{array}$$

### 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*  
 $\{List \ tl \ p \ Ps \wedge X \in set \ Ps\}$   
 $WHILE \ p \neq Null \wedge p \neq Ref \ X$   
 $INV \ \{\exists ps. List \ tl \ p \ ps \wedge X \in set \ ps\}$   
 $DO \ p := p.^{.}tl \ OD$   
 $\{p = Ref \ X\}$   
 $\langle proof \rangle$

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

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

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*  
 $\{(p, X) \in \{(Ref \ x, tl \ x) \mid x. True\}^{\wedge*}\}$   
 $WHILE \ p \neq Null \wedge p \neq X$   
 $INV \ \{(p, X) \in \{(Ref \ x, tl \ x) \mid x. True\}^{\wedge*}\}$   
 $DO \ p := p.^{.}tl \ OD$   
 $\{p = X\}$   
 $\langle proof \rangle$

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

**lemma** *VARs* *tl p*  
 $\{p \neq Null \wedge (addr \ p, X) \in \{(x, y). tl \ x = Ref \ y\}^{\wedge*}\}$   
 $WHILE \ p \neq Null \wedge p \neq Ref \ X$   
 $INV \ \{p \neq Null \wedge (addr \ p, X) \in \{(x, y). tl \ x = Ref \ y\}^{\wedge*}\}$   
 $DO \ p := p.^{.}tl \ OD$   
 $\{p = Ref \ X\}$   
 $\langle proof \rangle$



```

    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))}
⟨proof⟩

```

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))}
⟨proof⟩

```

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.

#### axiomatization

```

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

```

First some basic lemmas:

```

lemma [simp]: ispath f p p
⟨proof⟩
lemma [simp]: path f p p = []
⟨proof⟩

```

**lemma** [simp]:  $ispath\ f\ p\ q \implies a \notin set(path\ f\ p\ q) \implies ispath\ (f(a := r))\ p\ q$   
 <proof>

**lemma** [simp]:  $ispath\ f\ p\ q \implies a \notin set(path\ f\ p\ q) \implies$   
 $path\ (f(a := r))\ p\ q = path\ f\ p\ q$   
 <proof>

Some more specific lemmas needed by the example:

**lemma** [simp]:  $ispath\ (f(a := q))\ p\ (Ref\ a) \implies ispath\ (f(a := q))\ p\ q$   
 <proof>

**lemma** [simp]:  $ispath\ (f(a := q))\ p\ (Ref\ a) \implies$   
 $path\ (f(a := q))\ p\ q = path\ (f(a := q))\ p\ (Ref\ a)\ @\ [a]$   
 <proof>

**lemma** [simp]:  $ispath\ f\ p\ (Ref\ a) \implies f\ a = Ref\ b \implies$   
 $b \notin set\ (path\ f\ p\ (Ref\ a))$   
 <proof>

**lemma** [simp]:  $ispath\ f\ p\ (Ref\ a) \implies f\ a = Null \implies islist\ f\ p$   
 <proof>

**lemma** [simp]:  $ispath\ f\ p\ (Ref\ a) \implies f\ a = Null \implies list\ f\ p = path\ f\ p\ (Ref\ a)\ @\ [a]$   
 <proof>

**lemma** [simp]:  $islist\ f\ p \implies distinct\ (list\ f\ p)$   
 <proof>

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

The proof is automatic, but requires a numbet 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*  
 $\{ \text{root} = \text{Ref } r \wedge \text{distPath next root } (r \# Ps) \text{ root} \}$   
 $p := \text{root}; q := \text{root}^{\wedge}.\text{next};$   
*WHILE*  $q \neq \text{root}$   
*INV*  $\{ \exists ps qs. \text{distPath next } p ps \text{ root} \wedge \text{distPath next } q qs \text{ root} \wedge$   
 $\text{root} = \text{Ref } r \wedge r \notin \text{set } Ps \wedge \text{set } ps \cap \text{set } qs = \{ \} \wedge$   
 $Ps = (\text{rev } ps) @ qs \}$   
*DO*  $tmp := q; q := q^{\wedge}.\text{next}; tmp^{\wedge}.\text{next} := p; p := tmp$  *OD*;  
 $\text{root}^{\wedge}.\text{next} := p$   
 $\{ \text{root} = \text{Ref } r \wedge \text{distPath next root } (r \# \text{rev } Ps) \text{ root} \}$   
 $\langle \text{proof} \rangle$

In the beginning, we are able to assert *distPath next root as root*, with *as* set to  $\square$  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  $q = \text{root}$ , 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  $Ps = \text{rev } ps @ qs$ . 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  $r \# \text{rev } Ps$ .

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*  
 $\{ p = \text{Ref } r \wedge \text{distPath next } p (r \# Ps) p \}$   
 $q := \text{Null};$   
*WHILE*  $p \neq \text{Null}$   
*INV*  
 $\{ ((q = \text{Null}) \longrightarrow (\exists ps. \text{distPath next } p (ps) (\text{Ref } r) \wedge ps = r \# Ps)) \wedge$   
 $((q \neq \text{Null}) \longrightarrow (\exists ps qs. \text{distPath next } q (qs) (\text{Ref } r) \wedge \text{List next } p ps \wedge$   
 $\text{set } ps \cap \text{set } qs = \{ \} \wedge \text{rev } qs @ ps = Ps @ [r])) \wedge$   
 $\neg (p = \text{Null} \wedge q = \text{Null}) \}$   
*DO*  $tmp := p; p := p^{\wedge}.\text{next}; tmp^{\wedge}.\text{next} := q; q := tmp$  *OD*  
 $\{ q = \text{Ref } r \wedge \text{distPath next } q (r \# \text{rev } Ps) q \}$   
 $\langle \text{proof} \rangle$

## 0.4.6 Storage allocation

**definition** *new* :: 'a set  $\Rightarrow$  'a  
**where** *new* *A* = (SOME *a*.  $a \notin A$ )

**lemma** *new-notin*:

$\llbracket \sim \text{finite}(\text{UNIV}::'a \text{ set}); \text{finite}(A::'a \text{ set}); B \subseteq A \rrbracket \implies \text{new } (A) \notin B$   
 <proof>

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

**end**

**theory** *HeapSyntaxAbort* **imports** *Hoare-Logic-Abort 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**

$\text{-refupdate} :: ('a \Rightarrow 'b) \Rightarrow 'a \text{ ref} \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b)$   
 $(-/((-\rightarrow -)') [1000,0] 900)$   
 $\text{-fassign} :: 'a \text{ ref} \Rightarrow id \Rightarrow 'v \Rightarrow 's \text{ com}$   
 $((2-\hat{-} := / -) [70,1000,65] 61)$   
 $\text{-faccess} :: 'a \text{ ref} \Rightarrow ('a \text{ ref} \Rightarrow 'v) \Rightarrow 'v$   
 $(-\hat{-} [65,1000] 65)$

**translations**

$\text{-refupdate } f \ r \ v == f(\text{CONST addr } r := v)$   
 $p.\hat{f} := e \Rightarrow (p \neq \text{CONST Null}) \rightarrow (f := \text{-refupdate } f \ p \ e)$   
 $p.\hat{f} \Rightarrow f(\text{CONST addr } p)$

**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}
⟨proof⟩

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 ∧ 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 ≠ Null → p := p^.tl); r^.tl := q; q := r OD
  {List tl q (rev Ps @ Qs)}
⟨proof⟩

end

```

```

theory SchorrWaite imports HeapSyntax begin

```

## 0.6 Machinery for the Schorr-Waite proof

### definition

— Relations induced by a mapping  
 $rel :: ('a \Rightarrow 'a\ ref) \Rightarrow ('a \times 'a)\ set$   
**where**  $rel\ m = \{(x,y). m\ x = Ref\ y\}$

### definition

$relS :: ('a \Rightarrow 'a\ ref)\ set \Rightarrow ('a \times 'a)\ set$   
**where**  $relS\ M = (\bigcup m \in M. rel\ m)$

### definition

*addr*s :: 'a ref set  $\Rightarrow$  'a set  
**where** *addr*s P = {a. Ref a  $\in$  P}

**definition**

*reachable* :: ('a  $\times$  'a) set  $\Rightarrow$  'a ref set  $\Rightarrow$  'a set  
**where** *reachable* r P = (r\* `` *addr*s P)

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

Rewrite rules for relations induced by a mapping

**lemma** *self-reachable*:  $b \in B \Longrightarrow b \in R^* `` B$   
 <proof>

**lemma** *oneStep-reachable*:  $b \in R `` B \Longrightarrow b \in R^* `` B$   
 <proof>

**lemma** *still-reachable*:  $\llbracket B \subseteq Ra^* `` A; \forall (x,y) \in Rb - Ra. y \in (Ra^* `` A) \rrbracket \Longrightarrow Rb^* `` B \subseteq Ra^* `` A$   
 <proof>

**lemma** *still-reachable-eq*:  $\llbracket A \subseteq Rb^* `` B; B \subseteq Ra^* `` A; \forall (x,y) \in Ra - Rb. y \in (Rb^* `` B); \forall (x,y) \in Rb - Ra. y \in (Ra^* `` A) \rrbracket \Longrightarrow Ra^* `` A = Rb^* `` B$   
 <proof>

**lemma** *reachable-null*: *reachable* mS {Null} = {}  
 <proof>

**lemma** *reachable-empty*: *reachable* mS {} = {}  
 <proof>

**lemma** *reachable-union*: (*reachable* mS aS  $\cup$  *reachable* mS bS) = *reachable* mS (aS  $\cup$  bS)  
 <proof>

**lemma** *reachable-union-sym*: *reachable* r (insert a aS) = (r\* `` *addr*s {a})  $\cup$  *reachable* r aS  
 <proof>

**lemma** *rel-upd1*:  $(a,b) \notin \text{rel } (r(q:=t)) \Longrightarrow (a,b) \in \text{rel } r \Longrightarrow a=q$   
 <proof>

**lemma** *rel-upd2*:  $(a,b) \notin \text{rel } r \Longrightarrow (a,b) \in \text{rel } (r(q:=t)) \Longrightarrow a=q$   
 <proof>

**definition**

— Restriction of a relation

*restr* :: ('a  $\times$  'a) set  $\Rightarrow$  ('a  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\times$  'a) set      ((-/ | -) [50, 51] 50)  
**where** *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]:  
 $(\forall x. \neg m\ x) \implies (R\ |m) = R$   
 <proof>

**lemma** *restr-rtrancl*[simp]:  $\llbracket m\ l \rrbracket \implies (R\ |m)^* \ \{l\} = \{l\}$   
 <proof>

**lemma** [simp]:  $\llbracket m\ l \rrbracket \implies (l, x) \in (R\ |m)^* = (l=x)$   
 <proof>

**lemma** *restr-upd*:  $((rel\ (r\ (q := t)))(m(q := True))) = ((rel\ (r))(m(q := True)))$   
 <proof>

**lemma** *restr-un*:  $((r \cup s)|m) = (r|m) \cup (s|m)$   
 <proof>

**lemma** *rel-upd3*:  $(a, b) \notin (r|(m(q := t))) \implies (a, b) \in (r|m) \implies a = b$   
 <proof>

**definition**

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

Rewrite rules for Lists using S as their mapping

**lemma** [rule-format,simp]:  
 $\forall p. a \notin \text{set } stack \longrightarrow List\ (S\ c\ l\ r)\ p\ stack = List\ (S\ (c(a:=x))\ (l(a:=y))\ (r(a:=z)))\ p\ stack$   
 <proof>

**lemma** [rule-format,simp]:  
 $\forall p. a \notin \text{set } stack \longrightarrow List\ (S\ c\ l\ (r(a:=z)))\ p\ stack = List\ (S\ c\ l\ r)\ p\ stack$   
 <proof>

**lemma** [rule-format,simp]:  
 $\forall p. a \notin \text{set } stack \longrightarrow List\ (S\ c\ (l(a:=z))\ r)\ p\ stack = List\ (S\ c\ l\ r)\ p\ stack$   
 <proof>

**lemma** [rule-format,simp]:  
 $\forall p. a \notin \text{set } stack \longrightarrow List\ (S\ (c(a:=z))\ l\ r)\ p\ stack = List\ (S\ c\ l\ r)\ p\ stack$   
 <proof>

**primrec**

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

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

**where**

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

Rewrite rules for `stkOk`

**lemma**  $[simp]: \bigwedge t. \llbracket x \notin set\ xs; Ref\ x \neq t \rrbracket \implies$   
 $stkOk\ (c\ (x := f))\ l\ r\ iL\ iR\ t\ xs = stkOk\ c\ l\ r\ iL\ iR\ t\ xs$   
 $\langle proof \rangle$

**lemma**  $[simp]: \bigwedge t. \llbracket x \notin set\ xs; Ref\ x \neq t \rrbracket \implies$   
 $stkOk\ c\ (l(x := g))\ r\ iL\ iR\ t\ xs = stkOk\ c\ l\ r\ iL\ iR\ t\ xs$   
 $\langle proof \rangle$

**lemma**  $[simp]: \bigwedge t. \llbracket x \notin set\ xs; Ref\ x \neq t \rrbracket \implies$   
 $stkOk\ c\ l\ (r(x := g))\ iL\ iR\ t\ xs = stkOk\ c\ l\ r\ iL\ iR\ t\ xs$   
 $\langle proof \rangle$

**lemma**  $stkOk\text{-r-rewrite} [simp]: \bigwedge x. x \notin set\ xs \implies$   
 $stkOk\ c\ l\ (r(x := g))\ iL\ iR\ (Ref\ x)\ xs = stkOk\ c\ l\ r\ iL\ iR\ (Ref\ x)\ xs$   
 $\langle proof \rangle$

**lemma**  $[simp]: \bigwedge x. x \notin set\ xs \implies$   
 $stkOk\ c\ (l(x := g))\ r\ iL\ iR\ (Ref\ x)\ xs = stkOk\ c\ l\ r\ iL\ iR\ (Ref\ x)\ xs$   
 $\langle proof \rangle$

**lemma**  $[simp]: \bigwedge x. x \notin set\ xs \implies$   
 $stkOk\ (c(x := g))\ l\ r\ iL\ iR\ (Ref\ x)\ xs = stkOk\ c\ l\ r\ iL\ iR\ (Ref\ x)\ xs$   
 $\langle proof \rangle$

## 0.7 The Schorr-Waite algorithm

**theorem** *SchorrWaiteAlgorithm*:

$VARs\ c\ m\ l\ r\ t\ p\ q\ root$   
 $\{R = reachable\ (relS\ \{l, r\})\ \{root\} \wedge (\forall x. \neg m\ x) \wedge iR = r \wedge iL = l\}$   
 $t := root; p := Null;$   
 $WHILE\ p \neq Null \vee t \neq Null \wedge \neg t.^m$   
 $INV\ \{\exists\ stack.$   
 $List\ (S\ c\ l\ r)\ p\ stack \wedge \quad (*i1*)$   
 $(\forall x \in set\ stack. m\ x) \wedge \quad (*i2*)$   
 $R = reachable\ (relS\ \{l, r\})\ \{t, p\} \wedge \quad (*i3*)$   
 $(\forall x. x \in R \wedge \neg m\ x \longrightarrow \quad (*i4*)$   
 $\quad x \in reachable\ (relS\ \{l, r\} | m)\ (\{t\} \cup set\ (map\ r\ stack))) \wedge$   
 $(\forall x. m\ x \longrightarrow x \in R) \wedge \quad (*i5*)$   
 $(\forall x. x \notin set\ stack \longrightarrow r\ x = iR\ x \wedge l\ x = iL\ x) \wedge \quad (*i6*)$   
 $(stkOk\ c\ l\ r\ iL\ iR\ t\ stack) \quad (*i7*) \}$   
 $DO\ IF\ t = Null \vee t.^m$   
 $THEN\ IF\ p.^c$

```

      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
  { (∀ x. (x ∈ R) = m x) ∧ (r = iR ∧ 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⟩

end

```

```

theory SepLogHeap
imports Main
begin

```

```

type-synonym heap = (nat ⇒ nat option)

```

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

### 0.7.1 Paths in the heap

```

primrec Path :: heap ⇒ nat ⇒ nat list ⇒ nat ⇒ bool
where

```

```

  Path h x [] y = (x = y)
| Path h x (a#as) y = (x ≠ 0 ∧ a = x ∧ (∃ b. h x = Some b ∧ Path h b as y))

```

```

lemma [iff]: Path h 0 xs y = (xs = [] ∧ y = 0)
⟨proof⟩

```

```

lemma [simp]: x ≠ 0 ⇒ Path h x as z =
  (as = [] ∧ z = x ∨ (∃ y bs. as = x#bs ∧ h x = Some y & Path h y bs z))
⟨proof⟩

```

```

lemma [simp]: ∧x. Path f x (as@bs) z = (∃ y. Path f x as y ∧ Path f y bs z)
⟨proof⟩

```

```

lemma Path-upd[simp]:
  ∧x. u ∉ set as ⇒ Path (f(u := v)) x as y = Path f x as y
⟨proof⟩

```

### 0.7.2 Lists on the heap

```

definition List :: heap ⇒ nat ⇒ nat list ⇒ bool
  where List h x as = Path h x as 0

```

```

lemma [simp]: List h x [] = (x = 0)

```

*<proof>*

**lemma** [simp]:

$List\ h\ x\ (a\#\ as) = (x\neq 0 \wedge a=x \wedge (\exists y. h\ x = Some\ y \wedge List\ h\ y\ as))$   
*<proof>*

**lemma** [simp]:  $List\ h\ 0\ as = (as = [])$

*<proof>*

**lemma** *List-non-null*:  $a\neq 0 \implies$

$List\ h\ a\ as = (\exists b\ bs. as = a\#\ bs \wedge h\ a = Some\ b \wedge List\ h\ b\ bs)$   
*<proof>*

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

$\bigwedge x. a \notin set\ as \implies List\ (h(a := y))\ x\ as = List\ h\ x\ as$   
*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *List-hd-not-in-tl*[simp]:  $List\ h\ b\ as \implies h\ a = Some\ b \implies a \notin set\ as$

*<proof>*

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

*<proof>*

**lemma** *list-in-heap*:  $\bigwedge p. List\ h\ p\ ps \implies set\ ps \subseteq dom\ h$

*<proof>*

**lemma** *list-ortho-sum1*[simp]:

$\bigwedge p. [\![\ List\ h1\ p\ ps; dom\ h1 \cap dom\ h2 = \{\}\!] \implies List\ (h1++h2)\ p\ ps$   
*<proof>*

**lemma** *list-ortho-sum2*[simp]:

$\bigwedge p. [\![\ List\ h2\ p\ ps; dom\ h1 \cap dom\ h2 = \{\}\!] \implies List\ (h1++h2)\ p\ ps$   
*<proof>*

**end**

**theory** *Separation* **imports** *Hoare-Logic-Abort SepLogHeap* **begin**

The semantic definition of a few connectives:

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

**definition** *is-empty* :: *heap*  $\Rightarrow$  *bool*  
**where** *is-empty*  $h \iff h = \text{empty}$

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

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

**definition** *wand*:: (*heap*  $\Rightarrow$  *bool*)  $\Rightarrow$  (*heap*  $\Rightarrow$  *bool*)  $\Rightarrow$  (*heap*  $\Rightarrow$  *bool*)  
**where** *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*  
 $\{ x \neq z \}$   
 $\langle \text{proof} \rangle$

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**  
 $\text{-emp} :: \text{bool } (\text{emp})$   
 $\text{-singl} :: \text{nat } \Rightarrow \text{nat } \Rightarrow \text{bool } ([- \mapsto -])$   
 $\text{-star} :: \text{bool } \Rightarrow \text{bool } \Rightarrow \text{bool } (\text{infixl} \ ** \ 60)$   
 $\text{-wand} :: \text{bool } \Rightarrow \text{bool } \Rightarrow \text{bool } (\text{infixl} \ -* \ 60)$

$\langle ML \rangle$

Now it looks much better:

**lemma** *VARS*  $H \ x \ y \ z \ w$   
 $\{ [x \mapsto y] \ ** \ [z \mapsto w] \}$   
*SKIP*  
 $\{ x \neq z \}$   
 $\langle \text{proof} \rangle$

**lemma** *VARS*  $H \ x \ y \ z \ w$   
 $\{ \text{emp} \ ** \ \text{emp} \}$   
*SKIP*  
 $\{ \text{emp} \}$

$\langle proof \rangle$

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

$\langle ML \rangle$

Now the intermediate proof states are also readable:

**lemma** *VARs*  $H x y z w$   
 $\{[x \mapsto y] ** [z \mapsto w]\}$   
 $y := w$   
 $\{x \neq z\}$   
 $\langle proof \rangle$

**lemma** *VARs*  $H x y z w$   
 $\{emp ** emp\}$   
*SKIP*  
 $\{emp\}$   
 $\langle proof \rangle$

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*:  $P ** Q = Q ** P$   
 $\langle proof \rangle$

**lemma** *VARs*  $H x y z w$   
 $\{P ** Q\}$   
*SKIP*  
 $\{Q ** P\}$   
 $\langle proof \rangle$

**lemma** *VARs*  $H$   
 $\{p \neq 0 \wedge [p \mapsto x] ** List\ H\ q\ qs\}$   
 $H := H(p \mapsto q)$   
 $\{List\ H\ p\ (p \# qs)\}$   
 $\langle proof \rangle$

**lemma** *VARs*  $H p q r$   
 $\{List\ H\ p\ Ps ** List\ H\ q\ Qs\}$   
*WHILE*  $p \neq 0$   
*INV*  $\{\exists ps\ qs. (List\ H\ p\ ps ** List\ H\ q\ qs) \wedge rev\ ps\ @\ qs = rev\ Ps\ @\ Qs\}$   
*DO*  $r := p; p := the(H\ p); H := H(r \mapsto q); q := r$  *OD*  
 $\{List\ H\ q\ (rev\ Ps\ @\ Qs)\}$   
 $\langle proof \rangle$

**end**

```
theory Hoare  
imports Examples ExamplesAbort Pointers0 Pointer-Examples Pointer-ExamplesAbort  
SchorrWaite Separation  
begin  
  
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.