

# Hoare Logic

Various

November 22, 2007

### **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 . . . . .	12
0.1.1	Paths in the heap . . . . .	12
0.1.2	Lists on the heap . . . . .	12
0.1.3	Functional abstraction . . . . .	13
0.2	Verifications . . . . .	14
0.2.1	List reversal . . . . .	14
0.2.2	Searching in a list . . . . .	14
0.2.3	Merging two lists . . . . .	15
0.2.4	Storage allocation . . . . .	16
0.2.5	References . . . . .	16
0.3	The heap . . . . .	17
0.3.1	Paths in the heap . . . . .	17
0.3.2	Non-repeating paths . . . . .	17
0.3.3	Lists on the heap . . . . .	18
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 . . . . .	25
0.4.6	Storage allocation . . . . .	26
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 . . . . .	28
0.7	The Schorr-Waite algorithm . . . . .	30
0.7.1	Paths in the heap . . . . .	31
0.7.2	Lists on the heap . . . . .	32

```

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)

```

$\langle ML \rangle$

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

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

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

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

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

$\langle ML \rangle$

**end**

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

**constdefs**  
 $cd \quad :: [nat, nat, nat] \Rightarrow bool$   
 $cd \ x \ m \ n \ == \ x \ dvd \ m \ \& \ x \ dvd \ n$   
  
 $gcd \quad :: [nat, nat] \Rightarrow nat$   
 $gcd \ m \ n \quad == \ @x.(cd \ x \ m \ n) \ \& \ (!y.(cd \ y \ m \ n) \dashrightarrow y \leq x)$   
  
**consts** *fac*  $:: nat \Rightarrow nat$

**primrec**

$fac\ 0 = Suc\ 0$

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

**cd**

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

**gcd**

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

$\langle proof \rangle$

**lemma** *gcd-swap*:  $gcd\ m\ n = gcd\ n\ m$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

**pow**

**lemma** *sq-pow-div2* [simp]:

$m \bmod 2 = 0 \implies ((n::nat)*n)^(m \div 2) = n^m$

$\langle proof \rangle$

**end**

**theory** *Examples* **imports** *Hoare Arith2* **begin**

**lemma** *multiply-by-add*: *VARs m s a b*  
 $\{a=A \ \& \ b=B\}$   
 $m := 0; s := 0;$   
 $WHILE \ m \neq a$   
 $INV \ \{s=m*b \ \& \ a=A \ \& \ b=B\}$   
 $DO \ s := s+b; m := m+(1::nat) \ OD$   
 $\{s = A*B\}$   
 $\langle proof \rangle$

**lemma** *VARs M N P :: int*  
 $\{m=M \ \& \ n=N\}$   
 $IF \ M < 0 \ THEN \ M := -M; N := -N \ ELSE \ SKIP \ FI;$   
 $P := 0;$   
 $WHILE \ 0 < M$   
 $INV \ \{0 \leq M \ \& \ (EX \ p. \ p = (if \ m < 0 \ then \ -m \ else \ m) \ \& \ p*N = m*n \ \& \ P = (p-M)*N)\}$   
 $DO \ P := P+N; M := M - 1 \ OD$   
 $\{P = m*n\}$   
 $\langle proof \rangle$

**lemma** *Euclid-GCD*: *VARs a b*  
 $\{0 < A \ \& \ 0 < B\}$   
 $a := A; b := B;$   
 $WHILE \ a \neq b$   
 $INV \ \{0 < a \ \& \ 0 < b \ \& \ gcd \ A \ B = gcd \ a \ b\}$   
 $DO \ IF \ a < b \ THEN \ b := b-a \ ELSE \ a := a-b \ FI \ OD$   
 $\{a = gcd \ A \ B\}$   
 $\langle proof \rangle$

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

**lemma** *gcd-scm*: *VARs a b x y*  
 $\{0 < A \ \& \ 0 < B \ \& \ a=A \ \& \ b=B \ \& \ x=B \ \& \ y=A\}$   
 $WHILE \ a \neq b$   
 $INV \ \{0 < a \ \& \ 0 < b \ \& \ gcd \ A \ B = gcd \ a \ b \ \& \ 2*A*B = a*x + b*y\}$   
 $DO \ IF \ a < b \ THEN \ (b := b-a; x := x+y) \ ELSE \ (a := a-b; y := y+x) \ FI \ OD$   
 $\{a = gcd \ A \ B \ \& \ 2*A*B = a*(x+y)\}$   
 $\langle proof \rangle$

**lemma** *power-by-mult*: *VARs a b c*  
 $\{a=A \ \& \ b=B\}$   
 $c := (1::nat);$   
 $WHILE \ b \ \sim = \ 0$   
 $INV \ \{A \wedge B = c * a^b\}$   
 $DO \ WHILE \ b \ mod \ 2 = 0$   
 $INV \ \{A \wedge B = c * a^b\}$   
 $DO \ a := a*a; \ b := b \ div \ 2 \ OD;$   
 $c := c*a; \ b := b - 1$   
 $OD$   
 $\{c = A^B\}$   
 $\langle proof \rangle$

**lemma** *factorial*: *VARs a b*  
 $\{a=A\}$   
 $b := 1;$   
 $WHILE \ a \ \sim = \ 0$   
 $INV \ \{fac \ A = b * fac \ a\}$   
 $DO \ b := b*a; \ a := a - 1 \ OD$   
 $\{b = fac \ A\}$   
 $\langle proof \rangle$

**lemma** [*simp*]:  $1 \leq i \implies fac \ (i - Suc \ 0) * i = fac \ i$   
 $\langle proof \rangle$

**lemma** *VARs i f*  
 $\{True\}$   
 $i := (1::nat); \ f := 1;$   
 $WHILE \ i \leq n \ INV \ \{f = fac(i - 1) \ \& \ 1 \leq i \ \& \ i \leq n+1\}$   
 $DO \ f := f*i; \ i := i+1 \ OD$   
 $\{f = fac \ n\}$   
 $\langle proof \rangle$

**lemma** *sqrt*: *VARs r x*  
 $\{True\}$   
 $x := X; \ r := (0::nat);$   
 $WHILE \ (r+1)*(r+1) \leq x$   
 $INV \ \{r*r \leq x \ \& \ x=X\}$   
 $DO \ r := r+1 \ OD$   
 $\{r*r \leq X \ \& \ X < (r+1)*(r+1)\}$   
 $\langle proof \rangle$



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

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

**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}

```

$\langle proof \rangle$

**end**

**theory** *HoareAbort* **imports** *Main*  
**begin**

**types**

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

**datatype**

'a com = Basic 'a  $\Rightarrow$  '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)

**syntax**

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

**translations**

SKIP == Basic id

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

**consts** *iter* :: *nat* => '*a bexp* => '*a sem* => '*a sem*  
**primrec**  
*iter* 0 *b S* = ( $\lambda s s'. s \notin \text{Some } 'b \wedge s=s'$ )  
*iter* (*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** *Sem* :: '*a com* => '*a sem*  
**primrec**  
*Sem*(*Basic* *f*) *s s'* = ( $\text{case } s \text{ of } \text{None} \Rightarrow s' = \text{None} \mid \text{Some } t \Rightarrow s' = \text{Some}(f t)$ )  
*Sem* *Abort* *s s'* = ( $s' = \text{None}$ )  
*Sem*(*c1*;*c2*) *s s'* = ( $\exists s''. \text{Sem } c1 s s'' \wedge \text{Sem } c2 s'' s'$ )  
*Sem*(*IF* *b THEN* *c1 ELSE* *c2 FI*) *s s'* =  
 $(\text{case } s \text{ of } \text{None} \Rightarrow s' = \text{None}$   
 $\mid \text{Some } t \Rightarrow ((t \in b \longrightarrow \text{Sem } c1 s s') \wedge (t \notin b \longrightarrow \text{Sem } c2 s s')))$   
*Sem*(*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** *Valid* :: '*a bexp*  $\Rightarrow$  '*a com*  $\Rightarrow$  '*a bexp*  $\Rightarrow$  *bool*  
 $\text{Valid } p c q == \forall s s'. \text{Sem } c s s' \longrightarrow s : \text{Some } 'p \longrightarrow s' : \text{Some } 'q$

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

$\langle ML \rangle$

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

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

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

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

**lemma** *iter-aux*:

$! s s'. \text{Sem } c \ s \ s' \longrightarrow s \in \text{Some } ' (I \cap b) \longrightarrow s' \in \text{Some } ' I \Longrightarrow$   
 $(\bigwedge s s'. s \in \text{Some } ' I \Longrightarrow \text{iter } n \ b \ (\text{Sem } c) \ s \ s' \Longrightarrow s' \in \text{Some } ' (I \cap -b))$   
 $\langle \text{proof} \rangle$

**lemma** *WhileRule*:

$p \subseteq i \Longrightarrow \text{Valid } (i \cap b) \ c \ i \Longrightarrow i \cap (-b) \subseteq q \Longrightarrow \text{Valid } p \ (\text{While } b \ i \ c) \ q$   
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{ML} \rangle$

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

$\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

**syntax**

*guarded-com* ::  $\text{bool} \Rightarrow 'a \ \text{com} \Rightarrow 'a \ \text{com} \ ((\text{?} \rightarrow / -) \ 71)$   
*array-update* ::  $'a \ \text{list} \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a \ \text{com} \ ((\text{?}[-] := / -) \ [70,65] \ 61)$

**translations**

$P \rightarrow c == \text{IF } P \ \text{THEN } c \ \text{ELSE } \text{Abort } FI$   
 $a[i] := v \Rightarrow (i < \text{CONST length } a) \rightarrow (a := \text{list-update } a \ i \ v)$

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** *HoareAbort* **begin**

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

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

$\langle proof \rangle$

**lemma**

*VARs*  $a\ i\ j$   
 $\{k \leq \text{length } a \ \& \ i < k \ \& \ j < k\} \ j < \text{length } a \rightarrow a[i] := a[j] \ \{True\}$   
 $\langle proof \rangle$

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

$\{True\}$   
 $i := 0;$   
*WHILE*  $i < \text{length } a$   
*INV*  $\{i \leq \text{length } a\}$   
*DO*  $a[i] := \gamma; i := i+1$  *OD*  
 $\{True\}$   
 $\langle proof \rangle$

**end**

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

## 0.0.2 References

**axclass** *ref*  $< \text{type}$   
**consts** *Null*  $:: 'a::\text{ref}$

## 0.0.3 Field access and update

**syntax**

$@fassign :: 'a::\text{ref} \Rightarrow id \Rightarrow 'v \Rightarrow 's \text{ com}$   
 $((2-\hat{\cdot} \text{.-} := / -) [70,1000,65] 61)$   
 $@faccess :: 'a::\text{ref} \Rightarrow ('a::\text{ref} \Rightarrow 'v) \Rightarrow 'v$   
 $(-\hat{\cdot} \text{.-} [65,1000] 65)$

**translations**

$p \hat{\cdot} f := e \Rightarrow f := \text{fun-upd } f \ p \ e$   
 $p \hat{\cdot} f \Rightarrow f \ p$

An example due to Suzuki:

**lemma** *VARs*  $v\ n$

$\{\text{distinct}[w,x,y,z]\}$   
 $w \hat{\cdot} v := (1::\text{int}); w \hat{\cdot} n := x;$   
 $x \hat{\cdot} v := 2; x \hat{\cdot} n := y;$   
 $y \hat{\cdot} v := 3; y \hat{\cdot} n := z;$   
 $z \hat{\cdot} v := 4; x \hat{\cdot} n := z$   
 $\{w \hat{\cdot} n \hat{\cdot} n \hat{\cdot} v = 4\}$   
 $\langle proof \rangle$

## 0.1 The heap

### 0.1.1 Paths in the heap

**consts**

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

**primrec**

$Path\ h\ x\ []\ y = (x = y)$

$Path\ h\ x\ (a\#\ as)\ y = (x \neq Null \wedge x = a \wedge Path\ h\ (h\ a)\ as\ y)$

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

*<proof>*

**lemma** *[simp]*:  $a \neq Null \implies Path\ h\ a\ as\ z =$

$(as = [] \wedge z = a \vee (\exists bs. as = a\#\ bs \wedge Path\ h\ (h\ a)\ bs\ z))$

*<proof>*

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

*<proof>*

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

*<proof>*

### 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]*:  $List\ h\ x\ [] = (x = Null)$

*<proof>*

**lemma** *[simp]*:  $List\ h\ x\ (a\#\ as) = (x \neq Null \wedge x = a \wedge List\ h\ (h\ a)\ as)$

*<proof>*

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

*<proof>*

**lemma** *List-Ref* *[simp]*:

$a \neq Null \implies List\ h\ a\ as = (\exists bs. as = a\#\ bs \wedge List\ h\ (h\ a)\ bs)$

*<proof>*

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

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

*<proof>*

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

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

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

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

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

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

### 0.1.3 Functional abstraction

**constdefs**

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

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

**lemma** [simp]:  $islist \ h \ Null$   
 $\langle proof \rangle$

**lemma** [simp]:  $a \neq Null \implies islist \ h \ a = islist \ h \ (h \ a)$   
 $\langle proof \rangle$

**lemma** [simp]:  $list \ h \ Null = []$   
 $\langle proof \rangle$

**lemma** *list-Ref-conv[simp]*:  
 $\llbracket a \neq Null; islist \ h \ (h \ a) \rrbracket \implies list \ h \ a = a \# list \ h \ (h \ a)$   
 $\langle proof \rangle$

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

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

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

$\langle proof \rangle$

## 0.2 Verifications

### 0.2.1 List reversal

A short but unreadable proof:

**lemma** *VARs*  $tl\ p\ q\ r$   
 $\{List\ tl\ p\ Ps \wedge List\ tl\ q\ Qs \wedge set\ Ps \cap set\ Qs = \{\}\}$   
 $WHILE\ p \neq Null$   
 $INV\ \{\exists ps\ qs. List\ tl\ p\ ps \wedge List\ tl\ q\ qs \wedge set\ ps \cap set\ qs = \{\} \wedge$   
 $\quad rev\ ps\ @\ qs = rev\ Ps\ @\ Qs\}$   
 $DO\ r := p; p := p.^{.}tl; r.^{.}tl := q; q := r\ OD$   
 $\{List\ tl\ q\ (rev\ Ps\ @\ Qs)\}$   
 $\langle proof \rangle$

A longer readable version:

**lemma** *VARs*  $tl\ p\ q\ r$   
 $\{List\ tl\ p\ Ps \wedge List\ tl\ q\ Qs \wedge set\ Ps \cap set\ Qs = \{\}\}$   
 $WHILE\ p \neq Null$   
 $INV\ \{\exists ps\ qs. List\ tl\ p\ ps \wedge List\ tl\ q\ qs \wedge set\ ps \cap set\ qs = \{\} \wedge$   
 $\quad rev\ ps\ @\ qs = rev\ Ps\ @\ Qs\}$   
 $DO\ r := p; p := p.^{.}tl; r.^{.}tl := q; q := r\ OD$   
 $\{List\ tl\ q\ (rev\ Ps\ @\ Qs)\}$   
 $\langle proof \rangle$

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

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

### 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*  $tl\ p$   
 $\{List\ tl\ p\ Ps \wedge X \in set\ Ps\}$   
 $WHILE\ p \neq Null \wedge p \neq X$   
 $INV\ \{p \neq Null \wedge (\exists ps. List\ tl\ p\ ps \wedge X \in set\ ps)\}$



$DO\ p := p.^{tl}\ OD$   
 $\{p = X\}$   
 $\langle proof \rangle$

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.

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

### 0.2.3 Merging two lists

This is still a bit rough, especially the proof.

**consts** *merge* :: 'a list \* 'a list \* ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list

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

**lemma** *imp-disjCL*:  $(P|Q \longrightarrow R) = ((P \longrightarrow R) \wedge (\sim P \longrightarrow Q \longrightarrow R))$   
 $\langle proof \rangle$

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

**lemma** *VARs*  $hd\ tl\ p\ q\ r\ s$   
 $\{List\ tl\ p\ Ps \wedge List\ tl\ q\ Qs \wedge set\ Ps \cap set\ Qs = \{\} \wedge$   
 $\quad (p \neq Null \vee q \neq Null)\}$   
 $IF\ if\ q = Null\ then\ True\ else\ p \sim = Null \ \& \ p.^{hd} \leq q.^{hd}$   
 $THEN\ r := p; p := p.^{tl}\ ELSE\ r := q; q := q.^{tl}\ FI;$   
 $s := r;$   
 $WHILE\ p \neq Null \vee q \neq Null$   
 $INV\ \{EX\ rs\ ps\ qs. Path\ tl\ r\ rs\ s \wedge List\ tl\ p\ ps \wedge List\ tl\ q\ qs \wedge$

```

distinct(s # ps @ qs @ rs) ∧ s ≠ Null ∧
merge(Ps, Qs, λx y. hd x ≤ hd y) =
rs @ s # merge(ps, qs, λx y. hd x ≤ hd y) ∧
(tl s = p ∨ tl s = q)
DO IF if q = Null then True else 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⟩

```

### 0.2.4 Storage allocation

```

constdefs new :: 'a set ⇒ 'a::ref
new A == SOME a. a ∉ A & a ≠ Null

```

**lemma** new-notin:

```

[[ ~finite(UNIV::('a::ref)set); finite(A::'a set); B ⊆ A ]] ⇒
new (A) ∉ B & new A ≠ Null
⟨proof⟩

```

**lemma** ~finite(UNIV::('a::ref)set) ⇒

```

VARs xs elem next alloc p q
{Xs = xs ∧ p = (Null::'a)}
WHILE xs ≠ []
INV {islist next p ∧ set(list next p) ⊆ set alloc ∧
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}
⟨proof⟩

```

**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)  
⟨proof⟩

**lemma** not-Ref-eq [iff]: (ALL y. x ~ = Ref y) = (x = Null)  
⟨proof⟩

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

## 0.3 The heap

### 0.3.1 Paths in the heap

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

**lemma**  $[iff]: Path\ h\ Null\ xs\ y = (xs = [] \wedge y = Null)$   
 $\langle proof \rangle$

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

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

### 0.3.2 Non-repeating paths

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

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**  $neq\text{-}dP: p \neq q \implies Path\ h\ p\ Ps\ q \implies distinct\ Ps \implies$   
 $EX\ a\ Qs. p = Ref\ a \ \&\ Ps = a \# Qs \ \&\ a \notin set\ Qs$   
 $\langle proof \rangle$

**lemma**  $neq\text{-}dP\text{-}disp: [p \neq q; distPath\ h\ p\ Ps\ q] \implies$   
 $EX\ a\ Qs. p = Ref\ a \wedge Ps = a \# Qs \wedge a \notin set\ Qs$

$\langle proof \rangle$

### 0.3.3 Lists on the heap

#### Relational abstraction

##### constdefs

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

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

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

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

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

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

##### lemma $Path\text{-}is\text{-}List$ :

$\llbracket Path\ h\ b\ Ps\ (Ref\ a); a \notin set\ Ps \rrbracket \implies List\ (h(a := Null))\ b\ (Ps @ [a])$   
 $\langle proof \rangle$

### 0.3.4 Functional abstraction

##### constdefs

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

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

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

**lemma** *[simp]*:  $islist\ h\ Null$   
 $\langle proof \rangle$

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

**lemma** *[simp]*:  $list\ h\ Null = []$   
 $\langle proof \rangle$

**lemma** *list-Ref-conv[simp]*:  
 $islist\ h\ (h\ a) \Longrightarrow list\ h\ (Ref\ a) = a \# list\ h\ (h\ a)$   
 $\langle proof \rangle$

**lemma** *[simp]*:  $islist\ h\ (h\ a) \Longrightarrow a \notin set(list\ h\ (h\ a))$   
 $\langle proof \rangle$

**lemma** *list-upd-conv[simp]*:  
 $islist\ h\ p \Longrightarrow y \notin set(list\ h\ p) \Longrightarrow list\ (h(y := q))\ p = list\ h\ p$   
 $\langle proof \rangle$

**lemma** *islist-upd[simp]*:  
 $islist\ h\ p \Longrightarrow y \notin set(list\ h\ p) \Longrightarrow islist\ (h(y := q))\ p$   
 $\langle proof \rangle$

**end**

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

### 0.3.5 Field access and update

**syntax**

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

**translations**

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

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

An example due to Suzuki:

**lemma** *VARs* *v n*  
 $\{w = \text{Ref } w0 \ \& \ x = \text{Ref } x0 \ \& \ y = \text{Ref } y0 \ \& \ z = \text{Ref } z0 \ \& \text{distinct}[w0, x0, y0, z0]\}$   
 $w^\wedge.v := (1::\text{int}); w^\wedge.n := x;$   
 $x^\wedge.v := 2; x^\wedge.n := y;$   
 $y^\wedge.v := 3; y^\wedge.n := z;$   
 $z^\wedge.v := 4; x^\wedge.n := z$   
 $\{w^\wedge.n^\wedge.n^\wedge.v = 4\}$   
 $\langle \text{proof} \rangle$   
**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*  
 $\{\text{List } tl \ p \ Ps \wedge \text{List } tl \ q \ Qs \wedge \text{set } Ps \cap \text{set } Qs = \{\}\}$   
 $\text{WHILE } p \neq \text{Null}$   
 $\text{INV } \{\exists ps \ qs. \text{List } tl \ p \ ps \wedge \text{List } tl \ q \ qs \wedge \text{set } ps \cap \text{set } qs = \{\} \wedge$   
 $\text{rev } ps @ qs = \text{rev } Ps @ Qs\}$   
 $\text{DO } r := p; p := p^\wedge.tl; r^\wedge.tl := q; q := r \text{ OD}$   
 $\{\text{List } tl \ q \ (\text{rev } Ps @ Qs)\}$   
 $\langle \text{proof} \rangle$

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

**lemma** *VARs* *next p ps q qs r*  
 $\{\text{List } next \ p \ Ps \wedge \text{List } next \ q \ Qs \wedge \text{set } Ps \cap \text{set } Qs = \{\} \wedge$   
 $ps = Ps \wedge qs = Qs\}$   
 $\text{WHILE } p \neq \text{Null}$   
 $\text{INV } \{\text{List } next \ p \ ps \wedge \text{List } next \ q \ qs \wedge \text{set } ps \cap \text{set } qs = \{\} \wedge$   
 $\text{rev } ps @ qs = \text{rev } Ps @ Qs\}$   
 $\text{DO } r := p; p := p^\wedge.next; r^\wedge.next := q; q := r;$   
 $qs := (\text{hd } ps) \# qs; ps := tl \ ps \text{ OD}$   
 $\{\text{List } next \ q \ (\text{rev } Ps @ Qs)\}$   
 $\langle \text{proof} \rangle$

A longer readable version:

**lemma** *VARs*  $tl\ p\ q\ r$   
 $\{List\ tl\ p\ Ps \wedge List\ tl\ q\ Qs \wedge set\ Ps \cap set\ Qs = \{\}\}$   
 $WHILE\ p \neq Null$   
 $INV\ \{\exists ps\ qs. List\ tl\ p\ ps \wedge List\ tl\ q\ qs \wedge set\ ps \cap set\ qs = \{\} \wedge$   
 $\quad rev\ ps\ @\ qs = rev\ Ps\ @\ Qs\}$   
 $DO\ r := p; p := p.^{.}tl; r.^{.}tl := q; q := r\ OD$   
 $\{List\ tl\ q\ (rev\ Ps\ @\ Qs)\}$   
 $\langle proof \rangle$

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

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

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

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

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

### 0.4.3 Splicing two lists

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

### 0.4.4 Merging two lists

This is still a bit rough, especially the proof.

**constdefs**

$cor :: bool \Rightarrow bool \Rightarrow bool$   
 $cor\ P\ Q == if\ P\ then\ True\ else\ Q$   
 $cand :: bool \Rightarrow bool \Rightarrow bool$   
 $cand\ P\ Q == if\ P\ then\ Q\ else\ False$

**consts**  $merge :: 'a\ list * 'a\ list * ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list$

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

Simplifies the proof a little:



**lemma** [simp]:  $(\{\} = \text{insert } a \ A \cap B) = (a \notin B \ \& \ \{\} = A \cap B)$   
 $\langle \text{proof} \rangle$   
**lemma** [simp]:  $(\{\} = A \cap \text{insert } b \ B) = (b \notin A \ \& \ \{\} = A \cap B)$   
 $\langle \text{proof} \rangle$   
**lemma** [simp]:  $(\{\} = A \cap (B \cup C)) = (\{\} = A \cap B \ \& \ \{\} = A \cap C)$   
 $\langle \text{proof} \rangle$

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

And now with ghost variables:

**lemma** VARS *elem next p q r s ps qs rs a*  
 $\{ \text{List next } p \ Ps \wedge \text{List next } q \ Qs \wedge \text{set } Ps \cap \text{set } Qs = \{\} \wedge$   
 $(p \neq \text{Null} \vee q \neq \text{Null}) \wedge ps = Ps \wedge qs = Qs \}$   
 $\text{IF } \text{cor } (q = \text{Null}) \ (\text{cand } (p \neq \text{Null}) \ (p.^{elem} \leq q.^{elem}))$   
 $\text{THEN } r := p; p := p.^{next}; ps := tl \ ps$   
 $\text{ELSE } r := q; q := q.^{next}; qs := tl \ qs \text{ FI};$   
 $s := r; rs := []; a := \text{addr } s;$   
 $\text{WHILE } p \neq \text{Null} \vee q \neq \text{Null}$   
 $\text{INV } \{ \text{Path next } r \ rs \ s \wedge \text{List next } p \ ps \wedge \text{List next } q \ qs \wedge$   
 $\text{distinct}(a \ \# \ ps \ @ \ qs \ @ \ rs) \wedge s = \text{Ref } a \wedge$   
 $\text{merge}(Ps, Qs, \lambda x \ y. \text{elem } x \leq \text{elem } y) =$   
 $rs \ @ \ a \ \# \ \text{merge}(ps, qs, \lambda x \ y. \text{elem } x \leq \text{elem } y) \wedge$   
 $(\text{next } a = p \vee \text{next } a = q) \}$   
 $\text{DO IF } \text{cor } (q = \text{Null}) \ (\text{cand } (p \neq \text{Null}) \ (p.^{elem} \leq q.^{elem}))$   
 $\text{THEN } s.^{next} := p; p := p.^{next}; ps := tl \ ps$   
 $\text{ELSE } s.^{next} := q; q := q.^{next}; qs := tl \ qs \text{ FI};$   
 $rs := rs \ @ \ [a]; s := s.^{next}; a := \text{addr } s$   
 $\text{OD}$   
 $\{ \text{List next } r \ (\text{merge}(Ps, Qs, \lambda x \ y. \text{elem } x \leq \text{elem } y)) \}$   
 $\langle \text{proof} \rangle$

The proof is a LOT simpler because it does not need instantiations any-more, 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  $\Rightarrow$  'a ref)  $\Rightarrow$  'a ref  $\Rightarrow$  'a ref  $\Rightarrow$  bool  
           *path*:: ('a  $\Rightarrow$  'a ref)  $\Rightarrow$  'a ref  $\Rightarrow$  'a ref  $\Rightarrow$  '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*  $\Longrightarrow$  *a*  $\notin$  *set*(*path* *f* *p* *q*)  $\Longrightarrow$  *ispath* (*f*(*a* := *r*)) *p* *q*  
 <proof>  
**lemma** [*simp*]: *ispath* *f* *p* *q*  $\Longrightarrow$  *a*  $\notin$  *set*(*path* *f* *p* *q*)  $\Longrightarrow$   
           *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*)  $\Longrightarrow$  *ispath* (*f*(*a* := *q*)) *p* *q*  
 <proof>  
**lemma** [*simp*]: *ispath* (*f*(*a* := *q*)) *p* (*Ref* *a*)  $\Longrightarrow$   
           *path* (*f*(*a* := *q*)) *p* *q* = *path* (*f*(*a* := *q*)) *p* (*Ref* *a*) @ [*a*]  
 <proof>  
**lemma** [*simp*]: *ispath* *f* *p* (*Ref* *a*)  $\Longrightarrow$  *f* *a* = *Ref* *b*  $\Longrightarrow$   
           *b*  $\notin$  *set* (*path* *f* *p* (*Ref* *a*))  
 <proof>  
**lemma** [*simp*]: *ispath* *f* *p* (*Ref* *a*)  $\Longrightarrow$  *f* *a* = *Null*  $\Longrightarrow$  *islist* *f* *p*  
 <proof>  
**lemma** [*simp*]: *ispath* *f* *p* (*Ref* *a*)  $\Longrightarrow$  *f* *a* = *Null*  $\Longrightarrow$  *list* *f* *p* = *path* *f* *p* (*Ref* *a*) @ [*a*]  
 <proof>

**lemma** [*simp*]: *islist* *f* *p*  $\Longrightarrow$  *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( $a \# ps @ qs @ rs$ )  $\wedge s = \text{Ref } a \wedge$ 
merge( $Ps, Qs, \lambda x y. \text{hd } x \leq \text{hd } y$ ) =
 $rs @ a \# \text{merge}(ps, qs, \lambda x y. \text{hd } x \leq \text{hd } y) \wedge$ 
( $\text{tl } a = p \vee \text{tl } a = q$ )
DO IF cor ( $q = \text{Null}$ ) (cand ( $p \neq \text{Null}$ ) ( $p.^{\wedge}.\text{hd} \leq q.^{\wedge}.\text{hd}$ ))
THEN  $s.^{\wedge}.\text{tl} := p$ ;  $p := p.^{\wedge}.\text{tl}$  ELSE  $s.^{\wedge}.\text{tl} := q$ ;  $q := q.^{\wedge}.\text{tl}$  FI;
 $s := s.^{\wedge}.\text{tl}$ 
OD
{islist  $\text{tl } r$  & list  $\text{tl } r = (\text{merge}(Ps, Qs, \lambda x y. \text{hd } x \leq \text{hd } y))$ }
<proof>

```

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  $\wedge$  distPath next root ( $r \# Ps$ ) root}
p := root; q := root.^{\wedge}.next;
WHILE q  $\neq$  root
INV { $\exists ps qs. \text{distPath next } p \text{ } ps \text{ root} \wedge \text{distPath next } q \text{ } qs \text{ root} \wedge$ 
root = 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}.next; tmp.^{\wedge}.next := p; p:=tmp OD;
root.^{\wedge}.next := p
{ root = Ref r  $\wedge$  distPath next root ( $r \# \text{rev } Ps$ ) root }
<proof>

```

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 = Ref r  $\wedge$  distPath next p ( $r \# Ps$ ) p}
q:=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**  $tmp := p; p := p.^{\text{next}}; tmp.^{\text{next}} := q; q := tmp$  **OD**  
 $\{ q = \text{Ref } r \wedge \text{distPath next } q (r \# \text{rev } Ps) \}$   
 $\langle \text{proof} \rangle$

#### 0.4.6 Storage allocation

**constdefs**  $\text{new} :: 'a \text{ set} \Rightarrow 'a$   
 $\text{new } A == \text{SOME } a. a \notin A$

**lemma** *new-notin*:

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

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

$\text{VARs } xs \text{ elem next alloc } p \ q$   
 $\{ Xs = xs \wedge p = (\text{Null} :: 'a \text{ ref}) \}$   
**WHILE**  $xs \neq []$   
**INV**  $\{ \text{islist next } p \wedge \text{set}(\text{list next } p) \subseteq \text{set alloc} \wedge$   
 $\text{map elem } (\text{rev}(\text{list next } p)) @ xs = Xs \}$   
**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**  
 $\{ \text{islist next } p \wedge \text{map elem } (\text{rev}(\text{list next } p)) = Xs \}$   
 $\langle \text{proof} \rangle$

**end**

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

#### 0.4.7 Field access and update

Heap update  $p.^{\text{h}} := e$  is now guarded against  $p$  being  $\text{Null}$ . 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^.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

### **constdefs**

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

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

Rewrite rules for relations induced by a mapping

**lemma** *self-reachable*:  $b \in B \implies b \in R^* \text{ `` } B$   
 $\langle proof \rangle$

**lemma** *oneStep-reachable*:  $b \in R \text{ `` } B \implies b \in R^* \text{ `` } B$   
 $\langle proof \rangle$

**lemma** *still-reachable*:  $\llbracket B \subseteq Ra^* \text{ `` } A; \forall (x,y) \in Rb - Ra. y \in (Ra^* \text{ `` } A) \rrbracket \implies Rb^* \text{ `` } B \subseteq Ra^* \text{ `` } A$   
 $\langle proof \rangle$

**lemma** *still-reachable-eq*:  $\llbracket A \subseteq Rb^* \text{ `` } B; B \subseteq Ra^* \text{ `` } A; \forall (x,y) \in Ra - Rb. y \in (Rb^* \text{ `` } B); \forall (x,y) \in Rb - Ra. y \in (Ra^* \text{ `` } A) \rrbracket \implies Ra^* \text{ `` } A = Rb^* \text{ `` } B$   
 $\langle proof \rangle$

**lemma** *reachable-null*:  $reachable\ mS\ \{Null\} = \{\}$   
 $\langle proof \rangle$

**lemma** *reachable-empty*:  $reachable\ mS\ \{\} = \{\}$   
 $\langle proof \rangle$

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

**lemma** *reachable-union-sym*:  $reachable\ r\ (insert\ a\ aS) = (r^* \text{ `` } addrs\ \{a\}) \cup reachable\ r\ aS$   
 $\langle proof \rangle$

**lemma** *rel-upd1*:  $(a,b) \notin rel\ (r(q:=t)) \implies (a,b) \in rel\ r \implies a=q$   
 $\langle proof \rangle$

**lemma** *rel-upd2*:  $(a,b) \notin \text{rel } r \implies (a,b) \in \text{rel } (r(q:=t)) \implies a=q$   
 $\langle \text{proof} \rangle$

**constdefs**

— Restriction of a relation

*restr* ::  $('a \times 'a) \text{ set} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow ('a \times 'a) \text{ set}$        $((-/ | -) [50, 51] 50)$

*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*]:

$(\forall x. \neg m\ x) \implies (R | m) = R$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**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*]:

$\forall p. a \notin \text{set } \text{stack} \longrightarrow \text{List } (S\ c\ l\ r)\ p\ \text{stack} = \text{List } (S\ (c(a:=x))\ (l(a:=y))\ (r(a:=z)))\ p\ \text{stack}$

$\langle \text{proof} \rangle$

**lemma** [*rule-format,simp*]:

$\forall p. a \notin \text{set } \text{stack} \longrightarrow \text{List } (S\ c\ l\ (r(a:=z)))\ p\ \text{stack} = \text{List } (S\ c\ l\ r)\ p\ \text{stack}$

$\langle \text{proof} \rangle$

**lemma** [*rule-format,simp*]:

$\forall p. a \notin \text{set } \text{stack} \longrightarrow \text{List } (S\ c\ (l(a:=z))\ r)\ p\ \text{stack} = \text{List } (S\ c\ l\ r)\ p\ \text{stack}$

$\langle \text{proof} \rangle$

**lemma**  $[rule-format, simp]$ :

$\forall p. a \notin set\ stack \longrightarrow List\ (S\ (c(a:=z))\ l\ r)\ p\ stack = List\ (S\ c\ l\ r)\ p\ stack$   
 $\langle proof \rangle$

**consts**

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

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

**primrec**

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

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

Rewrite rules for  $stkOk$

**lemma**  $[simp]$ :  $\bigwedge t. [x \notin set\ xs; Ref\ x \neq t] \Longrightarrow$

$stkOk\ (c(x := f))\ l\ r\ iL\ iR\ t\ xs = stkOk\ c\ l\ r\ iL\ iR\ t\ xs$   
 $\langle proof \rangle$

**lemma**  $[simp]$ :  $\bigwedge t. [x \notin set\ xs; Ref\ x \neq t] \Longrightarrow$

$stkOk\ c\ (l(x := g))\ r\ iL\ iR\ t\ xs = stkOk\ c\ l\ r\ iL\ iR\ t\ xs$   
 $\langle proof \rangle$

**lemma**  $[simp]$ :  $\bigwedge t. [x \notin set\ xs; Ref\ x \neq t] \Longrightarrow$

$stkOk\ c\ l\ (r(x := g))\ iL\ iR\ t\ xs = stkOk\ c\ l\ r\ iL\ iR\ t\ xs$   
 $\langle proof \rangle$

**lemma**  $stkOk-r-rewrite\ [simp]$ :  $\bigwedge x. x \notin set\ xs \Longrightarrow$

$stkOk\ c\ l\ (r(x := g))\ iL\ iR\ (Ref\ x)\ xs = stkOk\ c\ l\ r\ iL\ iR\ (Ref\ x)\ xs$   
 $\langle proof \rangle$

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

$stkOk\ c\ (l(x := g))\ r\ iL\ iR\ (Ref\ x)\ xs = stkOk\ c\ l\ r\ iL\ iR\ (Ref\ x)\ xs$   
 $\langle proof \rangle$

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

$stkOk\ (c(x := g))\ l\ r\ iL\ iR\ (Ref\ x)\ xs = stkOk\ c\ l\ r\ iL\ iR\ (Ref\ x)\ xs$   
 $\langle proof \rangle$

## 0.7 The Schorr-Waite algorithm

**theorem** *SchorrWaiteAlgorithm*:

$VARs\ c\ m\ l\ r\ t\ p\ q\ root$

$\{R = reachable\ (relS\ \{l, r\})\ \{root\} \wedge (\forall x. \neg m\ x) \wedge iR = r \wedge iL = l\}$

$t := root; p := Null;$

$WHILE\ p \neq Null \vee t \neq Null \wedge \neg t \hat{=} m$

$INV\ \{\exists\ stack.$

$List\ (S\ c\ l\ r)\ p\ stack \wedge$

$(*i1*)$



```

      (∀x ∈ set stack. m x) ∧ (∗i2∗)
      R = reachable (relS{l, r}) {t,p} ∧ (∗i3∗)
      (∀x. x ∈ R ∧ ¬m x → (∗i4∗)
        x ∈ reachable (relS{l,r}|m) ({t} ∪ set(map r stack))) ∧
      (∀x. m x → x ∈ R) ∧ (∗i5∗)
      (∀x. x ∉ set stack → r x = iR x ∧ l x = iL x) ∧ (∗i6∗)
      (stkOk c l r iL iR t stack) (∗i7∗) }
DO IF t = Null ∨ 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

```

```

types heap = (nat ⇒ 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 ⇒ nat ⇒ nat list ⇒ nat ⇒ bool

```

```

primrec

```

```

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

## 0.7.2 Lists on the heap

**constdefs**

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

**lemma** *[simp]*:  $\text{List } h \ x \ [] = (x = 0)$   
 $\langle \text{proof} \rangle$

**lemma** *[simp]*:  
 $\text{List } h \ x \ (a \# as) = (x \neq 0 \wedge a = x \wedge (\exists y. h \ x = \text{Some } y \wedge \text{List } h \ y \ as))$   
 $\langle \text{proof} \rangle$

**lemma** *[simp]*:  $\text{List } h \ 0 \ as = (as = [])$   
 $\langle \text{proof} \rangle$

**lemma** *List-non-null*:  $a \neq 0 \implies$   
 $\text{List } h \ a \ as = (\exists b \ bs. as = a \# bs \wedge h \ a = \text{Some } b \wedge \text{List } h \ b \ bs)$   
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

**lemma** *list-in-heap*:  $\bigwedge p. \text{List } h \ p \ ps \implies \text{set } ps \subseteq \text{dom } h$   
 $\langle \text{proof} \rangle$

**lemma** *list-ortho-sum1[simp]*:  
 $\bigwedge p. [\![ \text{List } h1 \ p \ ps; \text{dom } h1 \cap \text{dom } h2 = \{\} ]\!] \implies \text{List } (h1 ++ h2) \ p \ ps$

$\langle proof \rangle$

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

$\bigwedge p. \llbracket List\ h2\ p\ ps; dom\ h1 \cap dom\ h2 = \{\} \rrbracket \implies List\ (h1 ++ h2)\ p\ ps$   
 $\langle proof \rangle$

**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*

$\{star\ (\%h. singl\ h\ x\ y)\ (\%h. singl\ h\ z\ w)\ h\}$   
*SKIP*  
 $\{x \neq z\}$   
 $\langle proof \rangle$

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**

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

$\langle ML \rangle$

Now it looks much better:

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

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

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**

# 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.