Hoare Logic

Various

August 15, 2018

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

theory *Hoare-Logic*
imports *Main*
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*)

⟨*ML*⟩

lemma *SkipRule*: $p \subseteq q \Longrightarrow Valid\ p\ (Basic\ id)\ q$
⟨*proof*⟩

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

lemma *SeqRule*: $Valid\ P\ c1\ Q \Longrightarrow Valid\ Q\ c2\ R \Longrightarrow Valid\ P\ (c1;c2)\ R$
⟨*proof*⟩

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

lemma *While-aux*:
assumes $Sem\ (WHILE\ b\ INV\ \{i\}\ DO\ c\ OD)\ s\ s'$
shows $\forall s\ s'.\ Sem\ c\ s\ s' \longrightarrow s \in I \wedge s \in b \longrightarrow s' \in I \Longrightarrow$
$s \in I \Longrightarrow s' \in I \wedge s' \notin b$
⟨*proof*⟩

lemma *WhileRule*:
$p \subseteq i \Longrightarrow Valid\ (i \cap b)\ c\ i \Longrightarrow i \cap (-b) \subseteq q \Longrightarrow Valid\ p\ (While\ b\ i\ c)\ q$
⟨*proof*⟩

lemma *Compl-Collect*: $-(Collect\ b) = \{x.\ ^\sim(b\ x)\}$
⟨*proof*⟩

lemmas *AbortRule* = *SkipRule* — dummy version
⟨*ML*⟩

end

theory *Arith2*
imports *Main*
begin

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

definition *gcd* :: $[nat, nat] \Rightarrow nat$
where $gcd\ m\ n = (SOME\ x.\ cd\ x\ m\ n\ \&\ (\forall y.(cd\ y\ m\ n) \longrightarrow y \leq x))$

primrec *fac* :: $nat \Rightarrow nat$
where
$fac\ 0 = Suc\ 0$
| $fac\ (Suc\ n) = Suc\ n * fac\ n$

cd

lemma *cd-nnn*: $0<n \Longrightarrow cd\ n\ n\ n$
⟨*proof*⟩

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

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

lemma *cd-diff-l*: $n \leq m \Longrightarrow cd\ x\ m\ n = cd\ x\ (m-n)\ n$
⟨*proof*⟩

lemma *cd-diff-r*: $m \leq n \Longrightarrow cd\ x\ m\ n = cd\ x\ m\ (n-m)$
⟨*proof*⟩

gcd

lemma *gcd-nnn*: $0<n \Longrightarrow n = gcd\ n\ n$
⟨*proof*⟩

lemma *gcd-swap*: $gcd\ m\ n = gcd\ n\ m$
⟨*proof*⟩

lemma *gcd-diff-l*: $n \leq m \Longrightarrow gcd\ m\ n = gcd\ (m-n)\ n$
⟨*proof*⟩

lemma *gcd-diff-r*: $m \leq n \Longrightarrow gcd\ m\ n = gcd\ m\ (n-m)$
⟨*proof*⟩

pow

lemma *sq-pow-div2* [*simp*]:
$m\ mod\ 2 = 0 \Longrightarrow ((n::nat)*n)\ \hat{}\ (m\ div\ 2) = n\ \hat{}\ m$
⟨*proof*⟩

end

Chapter 1

Various examples

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

1.1 ARITHMETIC

1.1.1 multiplication by successive addition

lemma *multiply-by-add*: *VARS m s a b*
 $\{a=A \wedge b=B\}$
 $m := 0$; $s := 0$;
 WHILE $m \neq a$
 INV $\{s=m*b \wedge a=A \wedge b=B\}$
 DO $s := s+b$; $m := m+(1::nat)$ *OD*
 $\{s = A*B\}$
⟨*proof*⟩

lemma *multiply-by-add-time*: *VARS m s a b t*
 $\{a=A \wedge b=B \wedge t=0\}$
 $m := 0$; $t := t+1$; $s := 0$; $t := t+1$;
 WHILE $m \neq a$
 INV $\{s=m*b \wedge a=A \wedge b=B \wedge t = 2*m + 2\}$
 DO $s := s+b$; $t := t+1$; $m := m+(1::nat)$; $t := t+1$ *OD*
 $\{s = A*B \wedge t = 2*A + 2\}$
⟨*proof*⟩

lemma *multiply-by-add2*: *VARS M N P :: int*
 $\{m=M \wedge n=N\}$
 IF $M < 0$ *THEN* $M := -M$; $N := -N$ *ELSE SKIP FI*;
 $P := 0$;
 WHILE $0 < M$
 INV $\{0 \leq M \wedge (\exists p.\ p = (if\ m<0\ then\ -m\ else\ m)\ \&\ p*N = m*n\ \&\ P = (p-M)*N)\}$
 DO $P := P+N$; $M := M - 1$ *OD*
 $\{P = m*n\}$
⟨*proof*⟩

lemma *multiply-by-add2-time*: *VARS M N P t :: int*
$\{m{=}M \wedge n{=}N \wedge t{=}0\}$
IF M < 0 THEN M := −M; t := t+1; N := −N; t := t+1 ELSE SKIP FI;
P := 0; t := t+1;
WHILE 0 < M
INV $\{0 \leq M$ & $(\exists p.\ p = (if\ m{<}0\ then\ -m\ else\ m)$ & $p{*}N = m{*}n$ & $P = (p{-}M){*}N$ & $t \geq 0$ & $t \leq 2{*}(p{-}M){+}3)\}$
DO P := P+N; t := t+1; M := M − 1; t := t+1 OD
$\{P = m{*}n$ & $t \leq 2{*}abs\ m + 3\}$
⟨*proof*⟩

1.1.2 Euclid's algorithm for GCD

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*⟩

lemma *Euclid-GCD-time*: *VARS a b t*
$\{0{<}A$ & $0{<}B$ & $t{=}0\}$
a := A; t := t+1; b := B; t := t+1;
WHILE $a \neq b$
INV $\{0{<}a$ & $0{<}b$ & $gcd\ A\ B = gcd\ a\ b$ & $a{\leq}A$ & $b{\leq}B$ & $t \leq max\ A\ B - max\ a\ b + 2\}$
DO IF a<b THEN b := b−a; t := t+1 ELSE a := a−b; t := t+1 FI OD
$\{a = gcd\ A\ B$ & $t \leq max\ A\ B + 2\}$
⟨*proof*⟩

1.1.3 Dijkstra's extension of Euclid's algorithm for simultaneous GCD and SCM

From E.W. Disjkstra. Selected Writings on Computing, p 98 (EWD474), where it is given without the invariant. Instead of defining *scm* explicitly we have used the theorem $scm\ x\ y = x * y\ /\ gcd\ x\ y$ and avoided division by mupltiplying with $gcd\ x\ y$.

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 $a \sim{=} b$
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*⟩

1.1.4 Power by iterated squaring and multiplication

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*⟩

1.1.5 Factorial

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 *factorial-time*: *VARS a b t*
{*a=A & t=0*}
b := *1*; *t* := *t*+1*;
WHILE a > *0*
INV {*fac A = b * fac a & a* ≤ *A & t = 2*(A*−*a)*+*1*}
DO b := *b*a*; *t* := *t*+1*; *a* := *a* − *1*; *t* := *t*+1 OD*
{*b = fac A & t = 2*A + 1*}
⟨*proof*⟩

lemma [*simp*]: *1* ≤ *i* ⟹ *fac* (*i* − *Suc 0*) * *i* = *fac i*
⟨*proof*⟩

lemma *factorial2*: *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 *factorial2-time*: *VARS i f t*
{*t=0*}

$i := (1::nat);\ t := t+1;\ f := 1;\ t := t+1;$
$WHILE\ i \leq n\ INV\ \{f = fac(i-1)\ \&\ 1 \leq i\ \&\ i \leq n+1\ \&\ t = 2*(i-1)+2\}$
$DO\ f := f*i;\ t := t+1;\ i := i+1;\ t := t+1\ OD$
$\{f = fac\ n\ \&\ t = 2*n+2\}$
⟨*proof*⟩

1.1.6 Square root

— the easy way:

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

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

1.2 LISTS

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

lemma *imperative-reverse-time*: $VARS\ y\ x\ t$
$\{x=X\ \&\ t=0\}$

y:=[]; $t := t+1$;
WHILE $x \sim= []$
INV $\{rev(x)@y = rev(X)\ \&\ t = 2*(length\ y) + 1\}$
DO $y := (hd\ x\ \#\ y)$; $t := t+1$; $x := tl\ x$; $t := t+1$ *OD*
$\{y=rev(X)\ \&\ t = 2*length\ X + 1\}$
⟨*proof*⟩

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
$\{y = X@Y\}$
⟨*proof*⟩

lemma *imperative-append-time-no-rev*: *VARS* $x\ y\ t$
$\{x=X\ \&\ y=Y\}$
$x := rev(x)$; $t := 0$;
WHILE $x\sim=[]$
INV $\{rev(x)@y = X@Y\ \&\ length\ x \leq length\ X\ \&\ t = 2 * (length\ X - length\ x)\}$
DO $y := (hd\ x\ \#\ y)$; $t := t+1$;
  $x := tl\ x$; $t := t+1$
OD
$\{y = X@Y\ \&\ t = 2 * length\ X\}$
⟨*proof*⟩

1.3 ARRAYS

1.3.1 Search for a key

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

lemma *zero-search-time*: *VARS* $A\ i\ t$
$\{t=0\}$
$i := 0$; $t := t+1$;
WHILE $i < length\ A \wedge A!i \neq key$
INV $\{(\forall j.\ j<i \longrightarrow A!j \neq key) \wedge i \leq length\ A \wedge t = i+1\}$

DO i := i+1; t := t+1 OD
{(i < length A ⟶ A!i = key) ∧
(i = length A ⟶ (∀ j. j < length A −−> A!j ≠ key)) ∧ t ≤ length A + 1}
⟨*proof*⟩

The *partition* procedure for quicksort.

- *A* is the array to be sorted (modelled as a list).
- Elements of *A* must be of class order to infer at the end that the elements between u and l are equal to pivot.

Ambiguity warnings of parser are due to := being used both for assignment and list update.

lemma *lem*: *m − Suc 0 < n ==> m < Suc n*
⟨*proof*⟩

lemma *Partition*:
[| leq == λA i. ∀ k. k<i ⟶ A!k ≤ pivot;
geq == λA 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}
— expand and delete abbreviations first
⟨*proof*⟩

end

theory *Hoare-Logic-Abort*
imports *Main*
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'' ⟹ Sem c2 s'' s' ⟹ Sem (c1;c2) s s'*
| *Sem (IF b THEN c1 ELSE c2 FI) None None*
| *s ∈ b ⟹ Sem c1 (Some s) s' ⟹ Sem (IF b THEN c1 ELSE c2 FI) (Some s) s'*
| *s ∉ b ⟹ Sem c2 (Some s) s' ⟹ Sem (IF b THEN c1 ELSE c2 FI) (Some s) s'*
| *Sem (While b x c) None None*
| *s ∉ b ⟹ Sem (While b x c) (Some s) (Some s)*
| *s ∈ b ⟹ Sem c (Some s) s'' ⟹ Sem (While b x c) s'' s' ⟹*
Sem (While b x c) (Some 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 ∈ Some ' p ⟶ s' ∈ Some ' q

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

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

⟨*ML*⟩

1.4 The proof rules

lemma *SkipRule*: *p ⊆ q ⟹ Valid p (Basic id) q*

⟨*proof*⟩

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

lemma *SeqRule*: $Valid\ P\ c1\ Q \Longrightarrow Valid\ Q\ c2\ R \Longrightarrow Valid\ P\ (c1;c2)\ R$
⟨*proof*⟩

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

lemma *While-aux*:
assumes $Sem\ (WHILE\ b\ INV\ \{i\}\ DO\ c\ OD)\ s\ s'$
shows $\forall s\ s'.\ Sem\ c\ s\ s' \longrightarrow s \in Some\ `\ (I \cap b) \longrightarrow s' \in Some\ `\ I \Longrightarrow$
$s \in Some\ `\ I \Longrightarrow s' \in Some\ `\ (I \cap -b)$
⟨*proof*⟩

lemma *WhileRule*:
$p \subseteq i \Longrightarrow Valid\ (i \cap b)\ c\ i \Longrightarrow i \cap (-b) \subseteq q \Longrightarrow Valid\ p\ (While\ b\ i\ c)\ q$
⟨*proof*⟩

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

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

lemma *Compl-Collect*: $-(Collect\ b) = \{x.\ \neg(b\ x)\}$
⟨*proof*⟩

⟨*ML*⟩
syntax
-guarded-com :: *bool* ⇒ *'a com* ⇒ *'a com* ((*2*- →/ -) *71*)
-array-update :: *'a list* ⇒ *nat* ⇒ *'a* ⇒ *'a com* ((*2*-[-] :=/ -) [*70*, *65*] *61*)
translations
$P \to c$ == *IF P THEN c ELSE CONST Abort FI*
$a[i] := v$ => $(i <$ *CONST length* $a) \to (a :=$ *CONST list-update* $a\ i\ v)$
— reverse translation not possible because of duplicate a

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

end

theory *ExamplesAbort* **imports** *Hoare-Logic-Abort* **begin**

lemma *VARS x y z*::*nat*

$\{y = z \ \& \ z \neq 0\} \ z \neq 0 \rightarrow x := y \ div \ z \ \{x = 1\}$
⟨*proof*⟩

lemma
VARS a i j
$\{k <= length \ a \ \& \ i < k \ \& \ j < k\} \ j < length \ a \rightarrow a[i] := a!j \ \{True\}$
⟨*proof*⟩

lemma *VARS* (*a*::*int list*) *i*
{*True*}
i := *0*;
WHILE i < *length a*
INV {*i* <= *length a*}
DO a[*i*] := *7*; *i* := *i*+*1 OD*
{*True*}
⟨*proof*⟩

end

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

1.4.2 References

class *ref* =
fixes *Null* :: *'a*

1.4.3 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*⟩

1.5 The heap

1.5.1 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*)
⟨*proof*⟩

lemma [*simp*]: *a* ≠ *Null* ⟹ *Path h a as z* =
 (*as* = [] ∧ *z* = *a* ∨ (∃ *bs*. *as* = *a*#*bs* ∧ *Path h* (*h a*) *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 [*simp*]: ⋀*x*. *u* ∉ *set as* ⟹ *Path* (*f*(*u* := *v*)) *x as y* = *Path f x as y*
⟨*proof*⟩

1.5.2 Lists on the heap

Relational abstraction

definition *List* :: (*'a*::*ref* ⇒ *'a*) ⇒ *'a* ⇒ *'a list* ⇒ *bool*
  where *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* ≠ *Null* ∧ *x* = *a* ∧ *List h* (*h a*) *as*)
⟨*proof*⟩

lemma [*simp*]: *List h Null as* = (*as* = [])
⟨*proof*⟩

lemma *List-Ref*[*simp*]:
 a ≠ *Null* ⟹ *List h a as* = (∃ *bs*. *as* = *a*#*bs* ∧ *List h* (*h a*) *bs*)
⟨*proof*⟩

theorem *notin-List-update*[*simp*]:
 ⋀*x*. *a* ∉ *set as* ⟹ *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*: ⋀*x bs*. *List h x as* ⟹ *List h x bs* ⟹ *as* = *bs*

⟨*proof*⟩

lemma *List-unique1*: *List h p as* ⟹ ∃!*as*. *List h p as*
⟨*proof*⟩

lemma *List-app*: ⋀*x*. *List h x* (*as*@*bs*) = (∃ *y*. *Path h x as y* ∧ *List h y bs*)
⟨*proof*⟩

lemma *List-hd-not-in-tl*[*simp*]: *List h* (*h a*) *as* ⟹ *a* ∉ *set as*
⟨*proof*⟩

lemma *List-distinct*[*simp*]: ⋀*x*. *List h x as* ⟹ *distinct as*
⟨*proof*⟩

1.5.3 Functional abstraction

definition *islist* :: (′*a*::*ref* ⇒ ′*a*) ⇒ ′*a* ⇒ *bool*
where *islist h p* ⟷ (∃ *as*. *List h p as*)

definition *list* :: (′*a*::*ref* ⇒ ′*a*) ⇒ ′*a* ⇒ ′*a list*
where *list h p* = (*SOME as*. *List h p as*)

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

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

lemma [*simp*]: *a* ≠ *Null* ⟹ *islist h a* = *islist h* (*h a*)
⟨*proof*⟩

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

lemma *list-Ref-conv*[*simp*]:
⟦ *a* ≠ *Null*; *islist h* (*h a*) ⟧ ⟹ *list h a* = *a* # *list h* (*h a*)
⟨*proof*⟩

lemma [*simp*]: *islist h* (*h a*) ⟹ *a* ∉ *set*(*list h* (*h a*))
⟨*proof*⟩

lemma *list-upd-conv*[*simp*]:
islist h p ⟹ *y* ∉ *set*(*list h p*) ⟹ *list* (*h*(*y* := *q*)) *p* = *list h p*
⟨*proof*⟩

lemma *islist-upd*[*simp*]:
islist h p ⟹ *y* ∉ *set*(*list h p*) ⟹ *islist* (*h*(*y* := *q*)) *p*
⟨*proof*⟩

1.6 Verifications

1.6.1 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 $r := p;\ p := p\hat{}.tl;\ r\hat{}.tl := q;\ q := r$ *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 $r := p;\ p := p\hat{}.tl;\ r\hat{}.tl := q;\ q := r$ *OD*
$\{List\ tl\ q\ (rev\ Ps\ @\ Qs)\}$
⟨*proof*⟩

Finaly, 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 $r := p;\ p := p\hat{}.tl;\ r\hat{}.tl := q;\ q := r$ *OD*
$\{islist\ tl\ q \wedge list\ tl\ q = rev\ Ps\ @\ Qs\}$
⟨*proof*⟩

1.6.2 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 $p := p\hat{}.tl$ *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 := $p\hat{}$*.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 := $p\hat{}$*.tl OD*
{$p = X$}
⟨*proof*⟩

1.6.3 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* & $p\hat{}$*.hd* $\leq$ $q\hat{}$*.hd*
THEN r := *p*; *p* := $p\hat{}$*.tl ELSE r* := *q*; *q* := $q\hat{}$*.tl FI*;
s := *r*;
WHILE $p \neq Null \vee q \neq Null$
INV {$\exists$ *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 ≠ 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*⟩

1.6.4 Storage allocation

definition *new :: 'a set ⇒ 'a::ref*
where *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**

1.6.5 References

datatype *'a ref = Null | Ref 'a*

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

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

primrec *addr :: 'a ref ⇒ 'a* **where**
addr (Ref a) = a

1.7 The heap

1.7.1 Paths in the heap

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

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

lemma [*simp*]: $Path\ h\ (Ref\ a)\ as\ z =$
$(as = [] \wedge z = Ref\ a\ \vee\ (\exists\, bs.\ as = a\#bs \wedge Path\ h\ (h\ a)\ bs\ z))$
⟨*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 *Path-upd*[*simp*]:
$\bigwedge x.\ u \notin set\ as \Longrightarrow Path\ (f(u := v))\ x\ as\ y = Path\ f\ x\ as\ y$
⟨*proof*⟩

lemma *Path-snoc*:
$Path\ (f(a := q))\ p\ as\ (Ref\ a) \Longrightarrow Path\ (f(a := q))\ p\ (as\ @\ [a])\ q$
⟨*proof*⟩

1.7.2 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 *neq-dP*: $p \neq q \Longrightarrow Path\ h\ p\ Ps\ q \Longrightarrow distinct\ Ps \Longrightarrow$
$\exists\, a\ Qs.\ p = Ref\ a \wedge Ps = a\#Qs \wedge a \notin set\ Qs$
⟨*proof*⟩

lemma *neq-dP-disp*: $[\![\ p \neq q;\ distPath\ h\ p\ Ps\ q\ ]\!] \Longrightarrow$
$\exists\, a\ Qs.\ p = Ref\ a \wedge Ps = a\#Qs \wedge a \notin set\ Qs$
⟨*proof*⟩

1.7.3 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*)
⟨*proof*⟩

lemma [*simp*]: *List h x* (*a*#*as*) = (*x* = *Ref a* ∧ *List h* (*h a*) *as*)
⟨*proof*⟩

lemma [*simp*]: *List h Null as* = (*as* = [])
⟨*proof*⟩

lemma *List-Ref*[*simp*]: *List h* (*Ref a*) *as* = (∃ *bs*. *as* = *a*#*bs* ∧ *List h* (*h a*) *bs*)
⟨*proof*⟩

theorem *notin-List-update*[*simp*]:
⋀*x*. *a* ∉ *set as* ⟹ *List* (*h*(*a* := *y*)) *x as* = *List h x as*
⟨*proof*⟩

lemma *List-unique*: ⋀*x bs*. *List h x as* ⟹ *List h x bs* ⟹ *as* = *bs*
⟨*proof*⟩

lemma *List-unique1*: *List h p as* ⟹ ∃!*as*. *List h p as*
⟨*proof*⟩

lemma *List-app*: ⋀*x*. *List h x* (*as*@*bs*) = (∃ *y*. *Path h x as y* ∧ *List h y bs*)
⟨*proof*⟩

lemma *List-hd-not-in-tl*[*simp*]: *List h* (*h a*) *as* ⟹ *a* ∉ *set as*
⟨*proof*⟩

lemma *List-distinct*[*simp*]: ⋀*x*. *List h x as* ⟹ *distinct as*
⟨*proof*⟩

lemma *Path-is-List*:
⟦*Path h b Ps* (*Ref a*); *a* ∉ *set Ps*⟧ ⟹ *List* (*h*(*a* := *Null*)) *b* (*Ps* @ [*a*])
⟨*proof*⟩

1.7.4 Functional abstraction

definition *islist* :: ('*a* ⇒ '*a ref*) ⇒ '*a ref* ⇒ *bool*
where *islist h p* ⟷ (∃ *as*. *List h p as*)

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

lemma *List-conv-islist-list*: *List h p as* = (*islist h p* ∧ *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) ⟹ list h (Ref a) = a # list h (h a)
⟨*proof*⟩

lemma [*simp*]: *islist h (h a) ⟹ a ∉ set(list h (h a))*
⟨*proof*⟩

lemma *list-upd-conv*[*simp*]:
islist h p ⟹ y ∉ set(list h p) ⟹ list (h(y := q)) p = list h p
⟨*proof*⟩

lemma *islist-upd*[*simp*]:
islist h p ⟹ y ∉ set(list h p) ⟹ islist (h(y := q)) p
⟨*proof*⟩

end

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

1.7.5 Field access and update

syntax
-refupdate :: ('a ⇒ 'b) ⇒ 'a ref ⇒ 'b ⇒ ('a ⇒ 'b)
(-/'((- → -)') [1000,0] 900)
-fassign :: 'a ref => id => 'v => 's com
((2-^.- :=/ -) [70,1000,65] 61)
-faccess :: 'a ref => ('a ref ⇒ 'v) => 'v
(-^.- [65,1000] 65)
translations
f(r → v) == f(CONST addr r := v)
p^.f := e => f := f(p → e)
p^.f => 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^.v := (1::int); w^.n := x;
x^.v := 2; x^.n := y;

$y\hat{}.v := 3;\ y\hat{}.n := z;$
$z\hat{}.v := 4;\ x\hat{}.n := z$
$\{w\hat{}.n\hat{}.n\hat{}.v = 4\}$
⟨*proof*⟩

end

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

axiomatization where *unproven*: *PROP A*

1.8 Verifications

1.8.1 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 $r := p;\ p := p\hat{}.tl;\ r\hat{}.tl := q;\ q := r$ *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 $r := p;\ p := p\hat{}.next;\ r\hat{}.next := q;\ q := r;$
$qs := (hd\ ps)\ \#\ qs;\ ps := tl\ ps$ *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 $r := p;\ p := p\hat{}.tl;\ r\hat{}.tl := q;\ q := r$ *OD*
$\{List\ tl\ q\ (rev\ Ps\ @\ Qs)\}$
⟨*proof*⟩

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

lemma *VARS tl p q r*
{*islist tl p* ∧ *islist tl q* ∧
Ps = *list tl p* ∧ *Qs* = *list tl q* ∧ *set Ps* ∩ *set Qs* = {}}
WHILE p ≠ *Null*
INV {*islist tl p* ∧ *islist tl q* ∧
set(*list tl p*) ∩ *set*(*list tl q*) = {} ∧
rev(*list tl p*) @ (*list tl q*) = *rev Ps* @ *Qs*}
DO r := *p*; *p* := *p*ˆ.*tl*; *r*ˆ.*tl* := *q*; *q* := *r OD*
{*islist tl q* ∧ *list tl q* = *rev Ps* @ *Qs*}
⟨*proof*⟩

1.8.2 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* ∧ *X* ∈ *set Ps*}
WHILE p ≠ *Null* ∧ *p* ≠ *Ref X*
INV {∃ *ps*. *List tl p ps* ∧ *X* ∈ *set ps*}
DO p := *p*ˆ.*tl OD*
{*p* = *Ref 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 ≠ *Null* ∧ *p* ≠ *X*
INV {∃ *ps*. *Path tl p ps X*}
DO p := *p*ˆ.*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. The first version uses a relation on *'a ref*:

lemma *VARS tl p*
{(*p*,*X*) ∈ {(*Ref x*,*tl x*) |*x*. *True*}*}
WHILE p ≠ *Null* ∧ *p* ≠ *X*
INV {(*p*,*X*) ∈ {(*Ref x*,*tl x*) |*x*. *True*}*}
DO p := *p*ˆ.*tl OD*
{*p* = *X*}
⟨*proof*⟩

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

lemma *VARS tl p*

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

1.8.3 Splicing two lists

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

1.8.4 Merging two lists

This is still a bit rough, especially the proof.

definition *cor* :: $bool \Rightarrow bool \Rightarrow bool$
where $cor\ P\ Q \longleftrightarrow (if\ P\ then\ True\ else\ Q)$

definition *cand* :: $bool \Rightarrow bool \Rightarrow bool$
where $cand\ P\ Q \longleftrightarrow (if\ P\ then\ Q\ else\ False)$

fun *merge* :: $'a\ list * 'a\ list * ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ list$
where
$merge(x\#xs,y\#ys,f) = (if\ f\ x\ y\ then\ x\ \#\ merge(xs,y\#ys,f)$
$\quad else\ y\ \#\ merge(x\#xs,ys,f))$
$| merge(x\#xs,[],f) = x\ \#\ merge(xs,[],f)$
$| merge([],y\#ys,f) = y\ \#\ merge([],ys,f)$
$| merge([],[],f) = []$

Simplifies the proof a little:

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

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

$path :: ('a \Rightarrow 'a\ ref) \Rightarrow 'a\ ref \Rightarrow 'a\ ref \Rightarrow 'a\ list$

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

$\{islist\ tl\ r\ \&\ list\ tl\ r = (merge(Ps,Qs,\lambda x\ y.\ hd\ x \leq hd\ y))\}$
$\langle proof \rangle$

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

1.8.5 Cyclic list reversal

We consider two algorithms for the reversal of circular lists.

lemma *circular-list-rev-I*:
VARS next root p q tmp
$\{root = Ref\ r \wedge distPath\ next\ root\ (r\#Ps)\ root\}$
$p := root;\ q := root\hat{}.next;$
WHILE $q \neq root$
INV $\{\exists\ ps\ qs.\ distPath\ next\ p\ ps\ root \wedge distPath\ next\ q\ qs\ root \wedge$
$root = Ref\ r \wedge r \notin set\ Ps\ \wedge set\ ps \cap set\ qs = \{\} \wedge$
$Ps = (rev\ ps)\ @\ qs\ \}$
DO $tmp := q;\ q := q\hat{}.next;\ tmp\hat{}.next := p;\ p{:=}tmp$ *OD*;
$root\hat{}.next := p$
$\{\ root = Ref\ r \wedge distPath\ next\ root\ (r\#rev\ Ps)\ root\}$
$\langle proof \rangle$

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

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

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

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

1.8.6 Storage allocation

definition *new* :: *'a set* ⇒ *'a*
where *new A* = (*SOME a*. *a* ∉ *A*)

lemma *new-notin*:
⟦ ~*finite*(*UNIV*::*'a set*); *finite*(*A*::*'a set*); *B* ⊆ *A* ⟧ ⟹ *new* (*A*) ∉ *B*
⟨*proof*⟩

lemma ~*finite*(*UNIV*::*'a set*) ⟹
VARS xs elem next alloc p q
{*Xs* = *xs* ∧ *p* = (*Null*::*'a ref*)}
WHILE xs ≠ []
INV {*islist next p* ∧ *set*(*list next p*) ⊆ *set alloc* ∧
map elem (*rev*(*list next p*)) @ *xs* = *Xs*}
DO q := *Ref*(*new*(*set alloc*)); *alloc* := (*addr q*)#*alloc*;
q^.*next* := *p*; *q*^.*elem* := *hd xs*; *xs* := *tl xs*; *p* := *q*
OD
{*islist next p* ∧ *map elem* (*rev*(*list next p*)) = *Xs*}
⟨*proof*⟩

end

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

1.8.7 Field access and update

Heap update *p*^.*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 so-called 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*(*CONST addr r* := *v*)
p^.*f* := *e* => (*p* ≠ *CONST Null*) → (*f* := *-refupdate f p e*)
p^.*f* => *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*ˆ.*v* := (*1*::*int*); *w*ˆ.*n* := *x*;
*x*ˆ.*v* := *2*; *x*ˆ.*n* := *y*;
*y*ˆ.*v* := *3*; *y*ˆ.*n* := *z*;
*z*ˆ.*v* := *4*; *x*ˆ.*n* := *z*
{*w*ˆ.*n*ˆ.*n*ˆ.*v* = *4*}
⟨*proof*⟩

end

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

1.9 Verifications

1.9.1 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*ˆ.*tl*); *r*ˆ.*tl* := *q*; *q* := *r OD*
{*List tl q* (*rev Ps* @ *Qs*)}
⟨*proof*⟩

end

theory *SchorrWaite* **imports** *HeapSyntax* **begin**

1.10 Machinery for the Schorr-Waite proof

definition
— Relations induced by a mapping
rel :: ('*a* ⇒ '*a ref*) ⇒ ('*a* × '*a*) *set*
where *rel m* = {(*x*,*y*). *m x* = *Ref y*}

definition
relS :: ('*a* ⇒ '*a ref*) *set* ⇒ ('*a* × '*a*) *set*

where *relS M* = (⋃ *m* ∈ *M*. *rel m*)

definition
addrs :: *'a ref set* ⇒ *'a set*
where *addrs P* = {*a*. *Ref a* ∈ *P*}

definition
reachable :: (*'a* × *'a*) *set* ⇒ *'a ref set* ⇒ *'a set*
where *reachable r P* = (*r** `` *addrs P*)

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

Rewrite rules for relations induced by a mapping

lemma *self-reachable*: *b* ∈ *B* ⟹ *b* ∈ *R** `` *B*
⟨*proof*⟩

lemma *oneStep-reachable*: *b* ∈ *R*``*B* ⟹ *b* ∈ *R*\* `` *B*
⟨*proof*⟩

lemma *still-reachable*: ⟦*B*⊆*Ra** ``*A*; ∀ (*x*,*y*) ∈ *Rb*−*Ra*. *y*∈ (*Ra*\* ``*A*)⟧ ⟹ *Rb** `` *B* ⊆ *Ra*\* `` *A*
⟨*proof*⟩

lemma *still-reachable-eq*: ⟦ *A*⊆*Rb** ``*B*; *B*⊆*Ra*\* ``*A*; ∀ (*x*,*y*) ∈ *Ra*−*Rb*. *y* ∈(*Rb** ``*B*); ∀ (*x*,*y*) ∈ *Rb*−*Ra*. *y*∈ (*Ra*\* ``*A*)⟧ ⟹ *Ra** ``*A* = *Rb*\* ``*B*
⟨*proof*⟩

lemma *reachable-null*: *reachable mS* {*Null*} = {}
⟨*proof*⟩

lemma *reachable-empty*: *reachable mS* {} = {}
⟨*proof*⟩

lemma *reachable-union*: (*reachable mS aS* ∪ *reachable mS bS*) = *reachable mS* (*aS* ∪ *bS*)
⟨*proof*⟩

lemma *reachable-union-sym*: *reachable r* (*insert a aS*) = (*r** `` *addrs* {*a*}) ∪ *reachable r aS*
⟨*proof*⟩

lemma *rel-upd1*: (*a*,*b*) ∉ *rel* (*r*(*q*:=*t*)) ⟹ (*a*,*b*) ∈ *rel r* ⟹ *a*=*q*
⟨*proof*⟩

lemma *rel-upd2*: (*a*,*b*) ∉ *rel r* ⟹ (*a*,*b*) ∈ *rel* (*r*(*q*:=*t*)) ⟹ *a*=*q*
⟨*proof*⟩

definition
— Restriction of a relation

restr ::('*a* × '*a*) *set* ⇒ ('*a* ⇒ *bool*) ⇒ ('*a* × '*a*) *set* ((-/ | -) [*50*, *51*] *50*)
where *restr r m* = {(*x*,*y*). (*x*,*y*) ∈ *r* ∧ ¬ *m x*}

Rewrite rules for the restriction of a relation

lemma *restr-identity*[*simp*]:
(∀ *x*. ¬ *m x*) ⟹ (*R* |*m*) = *R*
⟨*proof*⟩

lemma *restr-rtrancl*[*simp*]: ⟦*m l*⟧ ⟹ (*R* | *m*)* `` {*l*} = {*l*}
⟨*proof*⟩

lemma [*simp*]: ⟦*m l*⟧ ⟹ (*l*,*x*) ∈ (*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* ∪ *s*)|*m*) = (*r*|*m*) ∪ (*s*|*m*)
⟨*proof*⟩

lemma *rel-upd3*: (*a*, *b*) ∉ (*r*|(*m*(*q* := *t*))) ⟹ (*a*,*b*) ∈ (*r*|*m*) ⟹ *a* = *q*
⟨*proof*⟩

definition
— A short form for the stack mapping function for List
S :: ('*a* ⇒ *bool*) ⇒ ('*a* ⇒ '*a ref*) ⇒ ('*a* ⇒ '*a ref*) ⇒ ('*a* ⇒ '*a ref*)
where *S c l r* = (λ*x*. *if c x then r x else l x*)

Rewrite rules for Lists using S as their mapping

lemma [*rule-format*,*simp*]:
∀ *p*. *a* ∉ *set stack* ⟶ *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*]:
∀ *p*. *a* ∉ *set stack* ⟶ *List* (*S c l* (*r*(*a*:=*z*))) *p stack* = *List* (*S c l r*) *p stack*
⟨*proof*⟩

lemma [*rule-format*,*simp*]:
∀ *p*. *a* ∉ *set stack* ⟶ *List* (*S c* (*l*(*a*:=*z*)) *r*) *p stack* = *List* (*S c l r*) *p stack*
⟨*proof*⟩

lemma [*rule-format*,*simp*]:
∀ *p*. *a* ∉ *set stack* ⟶ *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-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*]: $\bigwedge t.\ [\![\ x \notin set\ xs;\ Ref\ x \neq t\ ]\!] \Longrightarrow$
$stkOk\ (c(x := f))\ l\ r\ iL\ iR\ t\ xs = stkOk\ c\ l\ r\ iL\ iR\ t\ xs$
⟨*proof*⟩

lemma [*simp*]: $\bigwedge t.\ [\![\ x \notin set\ xs;\ Ref\ x \neq t\ ]\!] \Longrightarrow$
$stkOk\ c\ (l(x := g))\ r\ iL\ iR\ t\ xs = stkOk\ c\ l\ r\ iL\ iR\ t\ xs$
⟨*proof*⟩

lemma [*simp*]: $\bigwedge t.\ [\![\ x \notin set\ xs;\ Ref\ x \neq t\ ]\!] \Longrightarrow$
$stkOk\ c\ l\ (r(x := g))\ iL\ iR\ t\ xs = stkOk\ c\ l\ r\ iL\ iR\ t\ xs$
⟨*proof*⟩

lemma *stkOk-r-rewrite* [*simp*]: $\bigwedge x.\ x \notin set\ xs \Longrightarrow$
$stkOk\ c\ l\ (r(x := g))\ iL\ iR\ (Ref\ x)\ xs = stkOk\ c\ l\ r\ iL\ iR\ (Ref\ x)\ xs$
⟨*proof*⟩

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

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

1.11 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*
$(\forall x \in set\ stack.\ m\ x) \wedge$ — *i2*
$R = reachable\ (relS\{l, r\})\ \{t,p\} \wedge$ — *i3*
$(\forall x.\ x \in R \wedge \neg m\ x \longrightarrow$ — *i4*
$x \in reachable\ (relS\{l,r\}|m)\ (\{t\} \cup set(map\ r\ stack))) \wedge$
$(\forall x.\ m\ x \longrightarrow x \in R) \wedge$ — *i5*
$(\forall x.\ x \notin set\ stack \longrightarrow r\ x = iR\ x \wedge l\ x = iL\ x) \wedge$ — *i6*

```
        (stkOk c l r iL iR t stack)                                  — i7}
 DO IF t = Null ∨ 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.

1.11.1 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⟩
```

1.11.2 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*≠*0* ∧ *a*=*x* ∧ (∃ *y*. *h x* = *Some y* ∧ *List h y as*))
⟨*proof*⟩

lemma [*simp*]: *List h 0 as* = (*as* = [])
⟨*proof*⟩

lemma *List-non-null*: *a*≠*0* ⟹
List h a as = (∃ *b bs*. *as* = *a*#*bs* ∧ *h a* = *Some b* ∧ *List h b bs*)
⟨*proof*⟩

theorem *notin-List-update*[*simp*]:
⋀*x*. *a* ∉ *set as* ⟹ *List* (*h*(*a* := *y*)) *x as* = *List h x as*
⟨*proof*⟩

lemma *List-unique*: ⋀*x bs*. *List h x as* ⟹ *List h x bs* ⟹ *as* = *bs*
⟨*proof*⟩

lemma *List-unique1*: *List h p as* ⟹ ∃!*as*. *List h p as*
⟨*proof*⟩

lemma *List-app*: ⋀*x*. *List h x* (*as*@*bs*) = (∃ *y*. *Path h x as y* ∧ *List h y bs*)
⟨*proof*⟩

lemma *List-hd-not-in-tl*[*simp*]: *List h b as* ⟹ *h a* = *Some b* ⟹ *a* ∉ *set as*
⟨*proof*⟩

lemma *List-distinct*[*simp*]: ⋀*x*. *List h x as* ⟹ *distinct as*
⟨*proof*⟩

lemma *list-in-heap*: ⋀*p*. *List h p ps* ⟹ *set ps* ⊆ *dom h*
⟨*proof*⟩

lemma *list-ortho-sum1*[*simp*]:
⋀*p*. ⟦ *List h1 p ps*; *dom h1* ∩ *dom h2* = {}⟧ ⟹ *List* (*h1*++*h2*) *p ps*
⟨*proof*⟩

lemma *list-ortho-sum2*[*simp*]:
⋀*p*. ⟦ *List h2 p ps*; *dom h1* ∩ *dom h2* = {}⟧ ⟹ *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* ⇒ *heap* ⇒ *bool* (**infix** ⊥ *55*)
where *h1* ⊥ *h2* ⟷ *dom h1* ∩ *dom h2* = {}

definition *is-empty* :: *heap* ⇒ *bool*
where *is-empty h* ⟷ *h* = *Map.empty*

definition *singl*:: *heap* ⇒ *nat* ⇒ *nat* ⇒ *bool*
where *singl h x y* ⟷ *dom h* = {*x*} & *h x* = *Some y*

definition *star*:: (*heap* ⇒ *bool*) ⇒ (*heap* ⇒ *bool*) ⇒ (*heap* ⇒ *bool*)
where *star P Q* = (λ*h*. ∃ *h1 h2*. *h* = *h1*++*h2* ∧ *h1* ⊥ *h2* ∧ *P h1* ∧ *Q h2*)

definition *wand*:: (*heap* ⇒ *bool*) ⇒ (*heap* ⇒ *bool*) ⇒ (*heap* ⇒ *bool*)
where *wand P Q* = (λ*h*. ∀ *h'*. *h'* ⊥ *h* ∧ *P h'* ⟶ *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* ≠ *z*}
⟨*proof*⟩

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

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

⟨*ML*⟩

Now it looks much better:

lemma *VARS H x y z w*
{[*x*↦*y*] ** [*z*↦*w*]}
SKIP
{*x* ≠ *z*}
⟨*proof*⟩

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

⟨*proof*⟩

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

⟨*ML*⟩

Now the intermediate proof states are also readable:

lemma *VARS H x y z w*
{[$x \mapsto y$] ** [$z \mapsto w$]}
$y := w$
{$x \neq z$}
⟨*proof*⟩

lemma *VARS H x y z w*
{*emp* ** *emp*}
SKIP
{*emp*}
⟨*proof*⟩

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.

— a law of separation logic
lemma *star-comm*: P ** $Q = Q$ ** P
⟨*proof*⟩

lemma *VARS H x y z w*
{P ** Q}
SKIP
{Q ** P}
⟨*proof*⟩

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

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*)}
⟨*proof*⟩

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.