Hoare Logic

Various

June 8, 2008

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

types
'a bexp = 'a set
'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)

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

translations
SKIP == Basic id

types *'a sem = 'a => 'a => bool*

consts *iter :: nat => 'a bexp => 'a sem => 'a sem*
primrec
iter 0 b S = (%s s'. s ~: 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 => '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 --> Sem c1 s s') &
(s ~: b --> Sem c2 s s'))
Sem(While b x c) s s' = (? n. iter n b (Sem c) s s')

constdefs *Valid :: 'a bexp ⇒ 'a com ⇒ 'a bexp ⇒ bool*
Valid p c q == !s s'. Sem c s s' --> s : p --> s' : q

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

⟨*ML*⟩

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 *iter-aux*: $!\ s\ s'.\ Sem\ c\ s\ s' \dashrightarrow s : I\ \&\ s : b \dashrightarrow s' : I \Longrightarrow$
$(\bigwedge s\ s'.\ s : I \Longrightarrow iter\ n\ b\ (Sem\ c)\ s\ s' \Longrightarrow s' : I\ \&\ s'\ {}^{\sim}: b)$
⟨*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*⟩

⟨*ML*⟩

end

theory *Arith2*
imports *Main*
begin

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

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

consts *fac* :: $nat \Rightarrow nat$

primrec
fac 0 = Suc 0
*fac(Suc n) = (Suc n)*fac(n)*

cd

lemma *cd-nnn*: *0<n ==> 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<=m ==> cd x m n = cd x (m−n) n*
⟨*proof*⟩

lemma *cd-diff-r*: *m<=n ==> cd x m n = cd x m (n−m)*
⟨*proof*⟩

gcd

lemma *gcd-nnn*: *0<n ==> n = gcd n n*
⟨*proof*⟩

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

lemma *gcd-diff-l*: *n<=m ==> gcd m n = gcd (m−n) n*
⟨*proof*⟩

lemma *gcd-diff-r*: *m<=n ==> gcd m n = gcd m (n−m)*
⟨*proof*⟩

pow

lemma *sq-pow-div2* [*simp*]:
*m mod 2 = 0 ==> ((n::nat)*n) ^(m div 2) = n^m*
⟨*proof*⟩

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~=a
*INV {s=m*b & a=A & b=B}*
DO s := s+b; m := m+(1::nat) OD
*{s = A*B}*
⟨*proof*⟩

lemma *VARS M N P :: int*
{m=M & n=N}
IF M < 0 THEN M := −M; N := −N ELSE SKIP FI;
P := 0;
WHILE 0 < M
*INV {0 <= M & (EX p. p = (if m<0 then −m else m) & p*N = m*n & P = (p−M)*N)}*
DO P := P+N; M := M − 1 OD
*{P = m*n}*
⟨*proof*⟩

lemma *Euclid-GCD*: *VARS a b*
{0<A & 0<B}
a := A; b := B;
WHILE a ≠ b
INV {0<a & 0<b & gcd A B = gcd a b}
DO IF a<b THEN b := b−a ELSE a := a−b FI OD
{a = gcd A B}
⟨*proof*⟩

lemmas *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 ~= b
*INV {0<a & 0<b & gcd A B = gcd a b & 2*A*B = a*x + b*y}*
DO IF a<b THEN (b := b−a; x := x+y) ELSE (a := a−b; y := y+x) FI OD
*{a = gcd A B & 2*A*B = a*(x+y)}*
⟨*proof*⟩

lemma *power-by-mult*: *VARS a b c*
$\{a{=}A \ \&\ b{=}B\}$
$c := (1{::}nat);$
WHILE $b \sim= 0$
INV $\{A\hat{}B = c * a\hat{}b\}$
DO WHILE $b \bmod 2 = 0$
INV $\{A\hat{}B = c * a\hat{}b\}$
DO $a := a{*}a;\ b := b \ div\ 2$ *OD*;
$c := c{*}a;\ b := b - 1$
OD
$\{c = A\hat{}B\}$
⟨*proof*⟩

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*
$\{b = fac\ A\}$
⟨*proof*⟩

lemma [*simp*]: $1 \leq i \Longrightarrow fac\ (i - Suc\ 0) * i = fac\ i$
⟨*proof*⟩

lemma *VARS i f*
$\{True\}$
$i := (1{::}nat);\ f := 1;$
WHILE $i <= n$ *INV* $\{f = fac(i - 1)\ \&\ 1 <= i\ \&\ i <= n{+}1\}$
DO $f := f{*}i;\ i := i{+}1$ *OD*
$\{f = fac\ n\}$
⟨*proof*⟩

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

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

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

lemma *imperative-append*: *VARS x y*
{*x*=*X* & *y*=*Y*}
x := *rev*(*x*);
WHILE x~=[]
INV {*rev*(*x*)@*y* = *X*@*Y*}
DO y := (*hd x* # *y*);
x := *tl x*
OD
{*y* = *X*@*Y*}
⟨*proof*⟩

lemma *zero-search*: *VARS A i*
{*True*}
i := *0*;
WHILE i < *length A* & *A*!*i* ~= *key*
INV {!*j*. *j*<*i* −−> *A*!*j* ~= *key*}
DO i := *i*+*1 OD*
{(*i* < *length A* −−> *A*!*i* = *key*) &
(*i* = *length A* −−> (!*j*. *j* < *length A* −−> *A*!*j* ~= *key*))}
⟨*proof*⟩

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

⟨*proof*⟩

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

⟨*proof*⟩

end

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

types
'a bexp = *'a set*
'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*)

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

translations
SKIP == *Basic id*

types *'a sem* = *'a option* => *'a option* => *bool*

consts *iter* :: *nat* => *'a bexp* => *'a sem* => *'a sem*
primrec
iter 0 b S = (λ*s s'*. *s* ∉ *Some* ‘ *b* ∧ *s*=*s'*)
iter (*Suc n*) *b S* =
(λ*s s'*. *s* ∈ *Some* ‘ *b* ∧ (∃ *s''*. *S s s''* ∧ *iter n b S s'' s'*))

consts *Sem* :: *'a com* => *'a sem*
primrec
Sem(*Basic f*) *s s'* = (*case s of None* ⇒ *s'* = *None* | *Some t* ⇒ *s'* = *Some*(*f t*))
Sem Abort s s' = (*s'* = *None*)
Sem(*c1*;*c2*) *s s'* = (∃ *s''*. *Sem c1 s s''* ∧ *Sem c2 s'' s'*)
Sem(*IF b THEN c1 ELSE c2 FI*) *s s'* =
(*case s of None* ⇒ *s'* = *None*
| *Some t* ⇒ ((*t* ∈ *b* ⟶ *Sem c1 s s'*) ∧ (*t* ∉ *b* ⟶ *Sem c2 s s'*)))
Sem(*While b x c*) *s s'* =
(*if s* = *None then s'* = *None else* ∃ *n*. *iter n b* (*Sem c*) *s s'*)

constdefs *Valid* :: *'a bexp* ⇒ *'a com* ⇒ *'a bexp* ⇒ *bool*
Valid p c q == ∀ *s s'*. *Sem c s s'* ⟶ *s* : *Some* ‘ *p* ⟶ *s'* : *Some* ‘ *q*

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

⟨*ML*⟩

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

lemma *BasicRule*: *p* ⊆ {*s*. *f s* ∈ *q*} ⟹ *Valid p* (*Basic f*) *q*
⟨*proof*⟩

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

lemma *CondRule*:
p ⊆ {*s*. (*s* ∈ *b* ⟶ *s* ∈ *w*) ∧ (*s* ∉ *b* ⟶ *s* ∈ *w'*)}
⟹ *Valid w c1 q* ⟹ *Valid w' c2 q* ⟹ *Valid p* (*Cond b c1 c2*) *q*
⟨*proof*⟩

lemma *iter-aux*:
$! \; s \; s'. \; Sem \; c \; s \; s' \longrightarrow s \in Some \; `` \; (I \cap b) \longrightarrow s' \in Some \; `` \; I \Longrightarrow$
$(\bigwedge s \; s'. \; s \in Some \; `` \; I \Longrightarrow iter \; n \; b \; (Sem \; c) \; s \; s' \Longrightarrow s' \in Some \; `` \; (I \cap -b))$
⟨*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*⟩

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

⟨*ML*⟩

lemma *Compl-Collect*: $-(Collect \; b) = \{x. \; \sim(b \; x)\}$
⟨*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
$P \rightarrow c$ == *IF P THEN c ELSE Abort FI*
$a[i] := v$ => $(i <$ *CONST length* $a) \rightarrow (a :=$ *list-update* $a \; i \; v)$

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

end

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

0.0.2 References

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

0.0.3 Field access and update

syntax
@*fassign* :: '*a*::*ref* => *id* => '*v* => '*s com*
((*2*-^.- :=/ -) [*70*,*1000*,*65*] *61*)
@*faccess* :: '*a*::*ref* => ('*a*::*ref* ⇒ '*v*) => '*v*
(-^.- [*65*,*1000*] *65*)
translations
p^.*f* := *e* => *f* := *fun-upd f p e*
p^.*f* => *f p*

An example due to Suzuki:

lemma *VARS v n*
{*distinct*[*w*,*x*,*y*,*z*]}
w^.*v* := (*1*::*int*); *w*^.*n* := *x*;
x^.*v* := *2*; *x*^.*n* := *y*;
y^.*v* := *3*; *y*^.*n* := *z*;
z^.*v* := *4*; *x*^.*n* := *z*
{*w*^.*n*^.*n*^.*v* = *4*}
⟨*proof*⟩

0.1 The heap

0.1.1 Paths in the heap

consts
Path :: (*'a*::*ref* ⇒ *'a*) ⇒ *'a* ⇒ *'a list* ⇒ *'a* ⇒ *bool*
primrec
Path h x [] *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*⟩

0.1.2 Lists on the heap

Relational abstraction

constdefs
List :: (*'a*::*ref* ⇒ *'a*) ⇒ *'a* ⇒ *'a list* ⇒ *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* ≠ *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*: $\bigwedge x\ bs.\ List\ h\ x\ as \Longrightarrow List\ h\ x\ bs \Longrightarrow as = bs$
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *list-Ref-conv*[*simp*]:
$[\![\ a \neq Null;\ islist\ h\ (h\ a)\ ]\!] \Longrightarrow list\ h\ a = a\ \#\ list\ h\ (h\ a)$
⟨*proof*⟩

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

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

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

⟨*proof*⟩

0.2 Verifications

0.2.1 List reversal

A short but unreadable proof:

lemma *VARS tl p q r*
{*List tl p Ps* ∧ *List tl q Qs* ∧ *set Ps* ∩ *set Qs* = {}}
WHILE p ≠ *Null*
INV {∃ *ps qs*. *List tl p ps* ∧ *List tl q qs* ∧ *set ps* ∩ *set qs* = {} ∧
rev ps @ *qs* = *rev Ps* @ *Qs*}
DO r := *p*; *p* := *p*^.*tl*; *r*^.*tl* := *q*; *q* := *r OD*
{*List tl q* (*rev Ps* @ *Qs*)}
⟨*proof*⟩

A longer readable version:

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

0.2.2 Searching in a list

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

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

lemma *VARS tl p*
{*List tl p Ps* ∧ *X* ∈ *set Ps*}
WHILE p ≠ *Null* ∧ *p* ≠ *X*
INV {*p* ≠ *Null* ∧ (∃ *ps*. *List tl p ps* ∧ *X* ∈ *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*
$\{(p,X) \in \{(x,y).\ y = tl\ x\ \&\ x \neq Null\}\hat{}*\}$
$WHILE\ p \neq Null \wedge p \neq X$
$INV\ \{(p,X) \in \{(x,y).\ y = tl\ x\ \&\ x \neq Null\}\hat{}*\}$
$DO\ p := p\hat{}.tl\ OD$
$\{p = X\}$
⟨*proof*⟩

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

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

$distinct(s \# ps \mathbin{@} qs \mathbin{@} rs) \wedge s \neq Null \wedge$
$merge(Ps,Qs,\lambda x\ y.\ hd\ x \leq hd\ y) =$
$rs \mathbin{@} s \# merge(ps,qs,\lambda x\ y.\ hd\ x \leq hd\ y) \wedge$
$(tl\ s = p \vee tl\ s = q)\}$
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
$\{List\ tl\ r\ (merge(Ps,Qs,\lambda x\ y.\ hd\ x \leq hd\ y))\}$
⟨*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 ref* ⇒ *'a*
primrec *addr*(*Ref a*) = *a*

0.3 The heap

0.3.1 Paths in the heap

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

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

lemma [*simp*]: *Path h* (*Ref a*) *as z* =
(*as* = [] ∧ *z* = *Ref 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 *Path-upd*[*simp*]:
⋀*x*. *u* ∉ *set as* ⟹ *Path* (*f*(*u* := *v*)) *x as y* = *Path f x as y*
⟨*proof*⟩

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

0.3.2 Non-repeating paths

constdefs
distPath :: (*'a* ⇒ *'a ref*) ⇒ *'a ref* ⇒ *'a list* ⇒ *'a ref* ⇒ *bool*
distPath h x as y ≡ *Path h x as y* ∧ *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* ≠ *q* ⟹ *Path h p Ps q* ⟹ *distinct Ps* ⟹
EX a Qs. *p* = *Ref a* & *Ps* = *a*#*Qs* & *a* ∉ *set Qs*
⟨*proof*⟩

lemma *neq-dP-disp*: ⟦ *p* ≠ *q*; *distPath h p Ps q* ⟧ ⟹
EX a Qs. *p* = *Ref a* ∧ *Ps* = *a*#*Qs* ∧ *a* ∉ *set Qs*

⟨*proof*⟩

0.3.3 Lists on the heap

Relational abstraction

constdefs
List :: (*'a* ⇒ *'a ref*) ⇒ *'a ref* ⇒ *'a list* ⇒ *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* = *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*⟩

0.3.4 Functional abstraction

constdefs
islist :: (*'a* ⇒ *'a ref*) ⇒ *'a ref* ⇒ *bool*
islist h p == ∃ *as*. *List h p as*

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

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

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

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

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

end

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

0.3.5 Field access and update

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

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-Examples* **imports** *HeapSyntax* **begin**

axiomatization where *unproven*: *PROP A*

0.4 Verifications

0.4.1 List reversal

A short but unreadable proof:

lemma *VARS tl p q r*
{*List tl p Ps ∧ List tl q Qs ∧ set Ps ∩ set Qs = {}*}
WHILE p ≠ Null
INV {*∃ps qs. List tl p ps ∧ List tl q qs ∧ set ps ∩ set qs = {} ∧*
rev ps @ qs = rev Ps @ Qs}
DO r := p; p := p^.tl; r^.tl := q; q := r OD
{*List tl q (rev Ps @ Qs)*}
⟨*proof*⟩

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

lemma *VARS next p ps q qs r*
{*List next p Ps ∧ List next q Qs ∧ set Ps ∩ set Qs = {} ∧*
ps = Ps ∧ qs = Qs}
WHILE p ≠ Null
INV {*List next p ps ∧ List next q qs ∧ set ps ∩ set qs = {} ∧*
rev ps @ qs = rev Ps @ Qs}
DO r := p; p := p^.next; r^.next := q; q := r;
qs := (hd ps) # qs; ps := tl ps OD
{*List next q (rev Ps @ Qs)*}
⟨*proof*⟩

A longer readable version:

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

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* ∧ *X* ∈ *set Ps*}
WHILE p ≠ *Null* ∧ *p* ≠ *Ref X*
INV {∃ *ps*. *List tl p ps* ∧ *X* ∈ *set ps*}
DO p := *p*^.*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*^.*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^.tl OD
{p = X}
⟨*proof*⟩

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

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

0.4.3 Splicing two lists

lemma *VARS tl p q pp qq*
{List tl p Ps ∧ List tl q Qs ∧ set Ps ∩ set Qs = {} ∧ size Qs ≤ size Ps}
pp := p;
WHILE q ≠ Null
INV {∃ as bs qs.
distinct as ∧ Path tl p as pp ∧ List tl pp bs ∧ List tl q qs ∧
set bs ∩ set qs = {} ∧ set as ∩ (set bs ∪ set qs) = {} ∧
size qs ≤ size bs ∧ 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)}
⟨*proof*⟩

0.4.4 Merging two lists

This is still a bit rough, especially the proof.

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

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

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

Simplifies the proof a little:

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

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

And now with ghost variables:

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

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* ⟹ *a* ∉ *set*(*path f p q*) ⟹ *ispath* (*f*(*a* := *r*)) *p q*
⟨*proof*⟩
lemma [*simp*]: *ispath f p q* ⟹ *a* ∉ *set*(*path f p q*) ⟹
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*) ⟹ *ispath* (*f*(*a* := *q*)) *p q*
⟨*proof*⟩
lemma [*simp*]: *ispath* (*f*(*a* := *q*)) *p* (*Ref a*) ⟹
path (*f*(*a* := *q*)) *p q* = *path* (*f*(*a* := *q*)) *p* (*Ref a*) @ [*a*]
⟨*proof*⟩
lemma [*simp*]: *ispath f p* (*Ref a*) ⟹ *f a* = *Ref b* ⟹
b ∉ *set* (*path f p* (*Ref a*))
⟨*proof*⟩
lemma [*simp*]: *ispath f p* (*Ref a*) ⟹ *f a* = *Null* ⟹ *islist f p*
⟨*proof*⟩
lemma [*simp*]: *ispath f p* (*Ref a*) ⟹ *f a* = *Null* ⟹ *list f p* = *path f p* (*Ref a*) @
[*a*]
⟨*proof*⟩

lemma [*simp*]: *islist f p* ⟹ *distinct* (*list f p*)
⟨*proof*⟩

lemma *VARS hd tl p q r s*
{*islist tl p* & *Ps* = *list tl p* ∧ *islist tl q* & *Qs* = *list tl q* ∧
set Ps ∩ *set Qs* = {} ∧
(*p* ≠ *Null* ∨ *q* ≠ *Null*)}
IF cor (*q* = *Null*) (*cand* (*p* ≠ *Null*) (*p*^.*hd* ≤ *q*^.*hd*))
THEN r := *p*; *p* := *p*^.*tl ELSE r* := *q*; *q* := *q*^.*tl FI*;
s := *r*;
WHILE p ≠ *Null* ∨ *q* ≠ *Null*
INV {*EX rs ps qs a*. *ispath tl r s* & *rs* = *path tl r s* ∧
islist tl p & *ps* = *list tl p* ∧ *islist tl q* & *qs* = *list tl q* ∧

$distinct(a \# ps \mathbin{@} qs \mathbin{@} rs) \wedge s = Ref\ a \wedge$
$merge(Ps,Qs,\lambda x\ y.\ hd\ x \leq hd\ y) =$
$rs \mathbin{@} a \# merge(ps,qs,\lambda x\ y.\ hd\ x \leq hd\ y) \wedge$
$(tl\ a = p \vee tl\ a = q)\}$
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
$\{islist\ tl\ r\ \&\ list\ tl\ r = (merge(Ps,Qs,\lambda x\ y.\ hd\ x \leq hd\ y))\}$
⟨*proof*⟩

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

0.4.5 Cyclic list reversal

We consider two algorithms for the reversal of circular lists.

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

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
$\{ ((q = Null) \longrightarrow (\exists\, ps.\ distPath\ next\ p\ (ps)\ (Ref\ r) \wedge ps = r\#Ps)) \wedge$
$\quad ((q \neq Null) \longrightarrow (\exists\, ps\ qs.\ distPath\ next\ q\ (qs)\ (Ref\ r) \wedge List\ next\ p\ ps\ \wedge$
$\qquad\qquad set\ ps \cap set\ qs = \{\} \wedge rev\ qs\ @\ ps = Ps@[r])) \wedge$
$\quad \neg\ (p = Null \wedge q = Null)\ \}$
DO $tmp := p;\ p := p\hat{}.next;\ tmp\hat{}.next := q;\ q{:=}tmp$ *OD*
$\{q = Ref\ r \wedge distPath\ next\ q\ (r\ \#\ rev\ Ps)\ q\}$
⟨*proof*⟩

0.4.6 Storage allocation

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

lemma *new-notin*:
$[\![\ \sim finite(UNIV{::}'a\ set);\ finite(A{::}'a\ set);\ B \subseteq A\ ]\!] \Longrightarrow new\ (A) \notin B$
⟨*proof*⟩

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

end

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

0.4.7 Field access and update

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

syntax

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


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

An example due to Suzuki:

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

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

Rewrite rules for relations induced by a mapping

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

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

lemma *still-reachable*: $[\![B \subseteq Ra^*``A;\ \forall\ (x,y) \in Rb-Ra.\ y \in (Ra^*``A)]\!] \Longrightarrow Rb^*\ ``\ B \subseteq Ra^*\ ``\ A$
⟨*proof*⟩

lemma *still-reachable-eq*: $[\![\ A \subseteq Rb^*``B;\ B \subseteq Ra^*``A;\ \forall\ (x,y) \in Ra-Rb.\ y \in (Rb^*``B);\ \forall\ (x,y) \in Rb-Ra.\ y \in (Ra^*``A)]\!] \Longrightarrow Ra^*``A = Rb^*``B$
⟨*proof*⟩

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

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

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

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

lemma *rel-upd1*: $(a,b) \notin rel\ (r(q:=t)) \Longrightarrow (a,b) \in rel\ r \Longrightarrow a=q$
⟨*proof*⟩

lemma *rel-upd2*: $(a,b) \notin rel\ r \Longrightarrow (a,b) \in rel\ (r(q:=t)) \Longrightarrow a=q$
⟨*proof*⟩

constdefs
— Restriction of a relation
restr ::$('a \times 'a)\ set \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \times 'a)\ set$ ((-/ | -) [*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*]:
$(\forall x.\ \neg\ m\ x) \Longrightarrow (R\ |m) = R$
⟨*proof*⟩

lemma *restr-rtrancl*[*simp*]: $[\![m\ l]\!] \Longrightarrow (R\ |\ m)^*\ ``\ \{l\} = \{l\}$
⟨*proof*⟩

lemma [*simp*]: $[\![m\ l]\!] \Longrightarrow (l,x) \in (R\ |\ m)^* = (l=x)$
⟨*proof*⟩

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

⟨*proof*⟩

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

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

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

Rewrite rules for Lists using S as their mapping

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

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

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

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

consts
— Recursive definition of what is means for a the graph/stack structure to be reconstructible
stkOk :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a\ ref) \Rightarrow ('a \Rightarrow 'a\ ref) \Rightarrow ('a \Rightarrow 'a\ ref) \Rightarrow ('a \Rightarrow 'a\ ref) \Rightarrow 'a\ ref \Rightarrow 'a\ list \Rightarrow bool$
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*]: $\bigwedge t.\ [\![\ x \notin set\ xs;\ Ref\ x \neq t\ ]\!] \Longrightarrow$
$stkOk\ (c(x := f))\ l\ r\ iL\ iR\ t\ xs = stkOk\ c\ l\ r\ iL\ iR\ t\ xs$
⟨*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*⟩

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 $\{\exists stack.$
$List\ (S\ c\ l\ r)\ p\ stack \wedge$ $(*i1*)$

$(\forall\, x \in set\ stack.\ m\ x) \wedge$ $(*i2*)$
$R = reachable\ (relS\{l,\ r\})\ \{t,p\} \wedge$ $(*i3*)$
$(\forall\, x.\ x \in R \wedge \neg m\ x \longrightarrow$ $(*i4*)$
$\quad x \in reachable\ (relS\{l,r\}|m)\ (\{t\} \cup set(map\ r\ stack))) \wedge$
$(\forall\, x.\ m\ x \longrightarrow x \in R) \wedge$ $(*i5*)$
$(\forall\, x.\ x \notin set\ stack \longrightarrow r\ x = iR\ x \wedge l\ x = iL\ x) \wedge$ $(*i6*)$
$(stkOk\ c\ l\ r\ iL\ iR\ t\ stack)$ $(*i7*)\ \}$
DO IF $t = Null \vee t\hat{}.m$
THEN IF $p\hat{}.c$
THEN $q := t;\ t := p;\ p := p\hat{}.r;\ t\hat{}.r := q$ $(*pop*)$
ELSE $q := t;\ t := p\hat{}.r;\ p\hat{}.r := p\hat{}.l;$ $(*swing*)$
$p\hat{}.l := q;\ p\hat{}.c := True$ *FI*
ELSE $q := p;\ p := t;\ t := t\hat{}.l;\ p\hat{}.l := q;$ $(*push*)$
$p\hat{}.m := True;\ p\hat{}.c := False$ *FI* *OD*
$\{(\forall\, x.\ (x \in R) = m\ x) \wedge (r = iR \wedge l = iL)\ \}$
(**is** *VARS c m l r t p q root* $\{?Pre\ c\ m\ l\ r\ root\}$ $(?c1;\ ?c2;\ ?c3)$ $\{?Post\ c\ m\ l\ r\}$)
⟨*proof*⟩

end

theory *SepLogHeap*
imports *Main*
begin

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

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

0.7.1 Paths in the heap

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

lemma [*iff*]: $Path\ h\ 0\ xs\ y = (xs = [] \wedge y = 0)$
⟨*proof*⟩

lemma [*simp*]: $x \neq 0 \Longrightarrow Path\ h\ x\ as\ z =$
$(as = [] \wedge z = x \ \vee\ (\exists\, y\ bs.\ as = x\#bs \wedge h\ x = Some\ y\ \&\ Path\ h\ y\ bs\ z))$
⟨*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*⟩

0.7.2 Lists on the heap

constdefs
List :: *heap* ⇒ *nat* ⇒ *nat list* ⇒ *bool*
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*]:
$\bigwedge x.\ a \notin set\ as \Longrightarrow List\ (h(a := y))\ x\ as = List\ h\ x\ as$
⟨*proof*⟩

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

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

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

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

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

lemma *list-in-heap*: $\bigwedge p.\ List\ h\ p\ ps \Longrightarrow set\ ps \subseteq dom\ h$
⟨*proof*⟩

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

⟨*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** *HoareAbort SepLogHeap* **begin**

The semantic definition of a few connectives:

constdefs
ortho:: *heap* ⇒ *heap* ⇒ *bool* (**infix** ⊥ *55*)
h1 ⊥ *h2* == *dom h1* ∩ *dom h2* = {}

is-empty :: *heap* ⇒ *bool*
is-empty h == *h* = *empty*

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

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

wand:: (*heap* ⇒ *bool*) ⇒ (*heap* ⇒ *bool*) ⇒ (*heap* ⇒ *bool*)
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*↦*y*] ** [*z*↦*w*]}
y := *w*
{*x* ≠ *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.

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*≠*0* ∧ [*p* ↦ *x*] ** *List H q qs*}
H := *H*(*p* ↦ *q*)
{*List H p* (*p*#*qs*)}

⟨*proof*⟩

lemma *VARS H p q r*
  {*List H p Ps* ∗∗ *List H q Qs*}
  WHILE p ≠ *0*
  INV {∃ *ps qs*. (*List H p ps* ∗∗ *List H q qs*) ∧ *rev ps* @ *qs* = *rev Ps* @ *Qs*}
  DO r := *p*; *p* := *the*(*H p*); *H* := *H*(*r* ↦ *q*); *q* := *r OD*
  {*List H q* (*rev Ps* @ *Qs*)}
⟨*proof*⟩

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.