

Hoare Logic

Various

May 22, 2012

Abstract

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

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

Contents

0.0.1	Derivation of the proof rules and, most importantly, the VCG tactic	12
0.0.2	References	13
0.0.3	Field access and update	13
0.1	The heap	14
0.1.1	Paths in the heap	14
0.1.2	Lists on the heap	14
0.1.3	Functional abstraction	15
0.2	Verifications	16
0.2.1	List reversal	16
0.2.2	Searching in a list	18
0.2.3	Merging two lists	19
0.2.4	Storage allocation	20
0.2.5	References	21
0.3	The heap	21
0.3.1	Paths in the heap	21
0.3.2	Non-repeating paths	22
0.3.3	Lists on the heap	22
0.3.4	Functional abstraction	24
0.3.5	Field access and update	25
0.4	Verifications	26
0.4.1	List reversal	26
0.4.2	Searching in a list	27
0.4.3	Splicing two lists	29
0.4.4	Merging two lists	29
0.4.5	Cyclic list reversal	33
0.4.6	Storage allocation	34
0.4.7	Field access and update	35
0.5	Verifications	36
0.5.1	List reversal	36
0.6	Machinery for the Schorr-Waite proof	36
0.7	The Schorr-Waite algorithm	40
0.7.1	Paths in the heap	49
0.7.2	Lists on the heap	49

```

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

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

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

abbreviation annskip (SKIP) where SKIP == Basic id

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

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

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

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

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

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

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

```

```

use hoare-syntax.ML
parse-translation << [(@{syntax-const -hoare-vars}, Hoare-Syntax.hoare-vars-tr)]
>>
print-translation << [(@{const-syntax Valid}, Hoare-Syntax.spec-tr' @ {syntax-const
-hoare})] >>

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma While-aux:
assumes Sem (WHILE b INV {i} DO c OD) s s'
shows  $\forall s s'. \text{Sem } c \text{ } s \text{ } s' \implies s \in I \wedge s \in b \implies s' \in I \implies$ 
 $s \in I \implies s' \in I \wedge s' \notin b$ 
using assms
by (induct WHILE b INV {i} DO c OD s s') auto

```

```

lemma WhileRule:
 $p \subseteq i \implies \text{Valid } (i \cap b) \text{ } c \text{ } i \implies i \cap (\neg b) \subseteq q \implies \text{Valid } p \text{ (While } b \text{ } i \text{ } c) \text{ } q$ 
apply (clarsimp simp: Valid-def)
apply (drule While-aux)
apply assumption
apply blast
apply blast
done

```

```

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

```

```

lemmas AbortRule = SkipRule — dummy version
use hoare-tac.ML

```

```

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

```

```

method-setup vcg-simp = <<

```

```

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

```

end

```

theory Arith2
imports Main
begin

```

```

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

```

```

definition gcd :: [nat, nat] => nat
  where gcd m n = (SOME x. cd x m n & (!y.(cd y m n)  $\longrightarrow$  y<=x))

```

```

primrec fac :: nat => nat
where
  fac 0 = Suc 0
| fac (Suc n) = Suc n * fac n

```

cd

```

lemma cd-nnn: 0<n ==> cd n n n
  apply (simp add: cd-def)
  done

```

```

lemma cd-le: [| cd x m n; 0<m; 0<n |] ==> x<=m & x<=n
  apply (unfold cd-def)
  apply (blast intro: dvd-imp-le)
  done

```

```

lemma cd-swap: cd x m n = cd x n m
  apply (unfold cd-def)
  apply blast
  done

```

```

lemma cd-diff-l: n<=m ==> cd x m n = cd x (m-n) n
  apply (unfold cd-def)
  apply (fastforce dest: dvd-diffD)
  done

```

```

lemma cd-diff-r: m<=n ==> cd x m n = cd x m (n-m)
  apply (unfold cd-def)
  apply (fastforce dest: dvd-diffD)
  done

```

gcd

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

lemma *gcd-swap*: $\text{gcd } m \ n = \text{gcd } n \ m$
apply (*simp add: gcd-def cd-swap*)
done

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

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

pow

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

end

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

lemma *multiply-by-add*: *VARs* $m \ s \ a \ b$
 $\{a=A \ \& \ b=B\}$
 $m := 0; \ s := 0;$

```

    WHILE  $m \sim a$ 
    INV  $\{s=m*b \ \& \ a=A \ \& \ b=B\}$ 
    DO  $s := s+b; m := m+(1::nat)$  OD
     $\{s = A*B\}$ 
by vcg-simp

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma gcd-scm: VARS  $a \ b \ x \ y$ 
 $\{0 < A \ \& \ 0 < B \ \& \ a=A \ \& \ b=B \ \& \ x=B \ \& \ y=A\}$ 
WHILE  $a \sim b$ 

```

```

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

```

```

lemma power-by-mult: VARS a b c
{a=A & b=B}
c := (1::nat);
WHILE b ~ = 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}
apply vcg-simp
apply(case-tac b)
apply simp
apply simp
done

```

```

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}
apply vcg-simp
apply(clarsimp split: nat-diff-split)
done

```

```

lemma [simp]: 1 ≤ i ⇒ fac (i - Suc 0) * i = fac i
by(induct i, simp-all)

```

```

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}
  apply vcg-simp
  apply(subgoal-tac i = Suc n)
  apply simp
  apply arith
  done

```

```

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

```

```

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

```

```

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

```

```

lemma imperative-append: VARS x y

```

```

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

```

lemma *zero-search*: VARS A i

```

{True}
i := 0;
WHILE i < length A & A!i ~ = 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))}
apply vcg-simp
apply(blast elim!: less-SucE)
done

```

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

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}

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

end

```

```

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

```

```

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

```

```

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

```

```

abbreviation annskip (SKIP) where SKIP == Basic id

```

```

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

```

```

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

```

```

where

```

```

  Sem (Basic f) None None
| Sem (Basic f) (Some s) (Some (f s))
| Sem Abort s None
| Sem c1 s s'' => 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 $Valid :: 'a\ bexp \Rightarrow 'a\ com \Rightarrow 'a\ bexp \Rightarrow bool$ **where**

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

syntax

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

syntax

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

syntax (output)

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

use $hoare-syntax.ML$

parse-translation $\ll [(@\{syntax-const\ -hoare-abort-vars\}, Hoare-Syntax.hoare-vars-tr)]$
 \gg

print-translation

$\ll [(@\{const-syntax\ Valid\}, Hoare-Syntax.spec-tr'\ @\{syntax-const\ -hoare-abort\})]$
 \gg

lemma $SkipRule: p \subseteq q \Longrightarrow Valid\ p (Basic\ id)\ q$

by $(auto\ simp: Valid-def)$

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

by $(auto\ simp: Valid-def)$

lemma $SeqRule: Valid\ P\ c1\ Q \Longrightarrow Valid\ Q\ c2\ R \Longrightarrow Valid\ P\ (c1;c2)\ R$

by $(auto\ simp: Valid-def)$

lemma $CondRule:$

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

by $(fastforce\ simp: Valid-def\ image-def)$

lemma $While-aux:$

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

shows $\forall s\ s'. Sem\ c\ s\ s' \longrightarrow s \in Some\ ' (I \cap b) \longrightarrow s' \in Some\ ' I \Longrightarrow$
 $s \in Some\ ' I \Longrightarrow s' \in Some\ ' (I \cap \neg b)$

using $assms$

by (induct WHILE b INV {i} DO c OD s s') auto

lemma WhileRule:

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

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

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

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

use hoare-tac.ML

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

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

syntax

-guarded-com :: $\text{bool} \Rightarrow 'a \text{ com} \Rightarrow 'a \text{ com} \ ((2- \rightarrow / -) \ 71)$
 -array-update :: $'a \text{ list} \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow 'a \text{ com} \ ((2-[-] := / -) \ [70, 65] \ 61)$

translations

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

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

end

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

```

lemma VARS  $x\ y\ z::nat$ 
   $\{y = z \ \&\ z \neq 0\} \ z \neq 0 \rightarrow x := y \ \text{div}\ z \ \{x = 1\}$ 
by vcg-simp

```

```

lemma
  VARS  $a\ i\ j$ 
   $\{k \leq \text{length}\ a \ \&\ i < k \ \&\ j < k\} \ j < \text{length}\ a \rightarrow a[i] := a!j \ \{True\}$ 
by vcg-simp

```

```

lemma VARS  $(a::int\ list)\ i$ 
   $\{True\}$ 
   $i := 0;$ 
  WHILE  $i < \text{length}\ a$ 
  INV  $\{i \leq \text{length}\ a\}$ 
  DO  $a[i] := \text{?}; i := i+1$  OD
   $\{True\}$ 
by vcg-simp

```

end

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

0.0.2 References

```

class ref =
  fixes Null :: 'a

```

0.0.3 Field access and update

```

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

```

translations

```

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

```

An example due to Suzuki:

```

lemma VARS  $v\ n$ 
   $\{distinct[w,x,y,z]\}$ 
   $w \hat{\cdot} v := (1::int); w \hat{\cdot} n := x;$ 
   $x \hat{\cdot} v := 2; x \hat{\cdot} n := y;$ 
   $y \hat{\cdot} v := 3; y \hat{\cdot} n := z;$ 
   $z \hat{\cdot} v := 4; x \hat{\cdot} n := z$ 
   $\{w \hat{\cdot} n \hat{\cdot} n \hat{\cdot} v = 4\}$ 
by vcg-simp

```

0.1 The heap

0.1.1 Paths in the heap

primrec *Path* :: ('a::ref \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a list \Rightarrow 'a \Rightarrow bool
where

Path *h* *x* [] *y* = (*x* = *y*)
| *Path* *h* *x* (*a*#*as*) *y* = (*x* \neq Null \wedge *x* = *a* \wedge *Path* *h* (*h* *a*) *as* *y*)

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

apply(*case-tac* *xs*)

apply *fastforce*

apply *fastforce*

done

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

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

apply(*case-tac* *as*)

apply *fastforce*

apply *fastforce*

done

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

by(*induct* *as*, *simp*+))

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

by(*induct* *as*, *simp*, *simp* *add:eq-sym-conv*)

0.1.2 Lists on the heap

Relational abstraction

definition *List* :: ('a::ref \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a list \Rightarrow bool

where *List* *h* *x* *as* = *Path* *h* *x* *as* Null

lemma [*simp*]: *List* *h* *x* [] = (*x* = Null)

by(*simp* *add>List-def*)

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

by(*simp* *add>List-def*)

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

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

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

a \neq Null \Longrightarrow *List* *h* *a* *as* = (\exists *bs*. *as* = *a*#*bs* \wedge *List* *h* (*h* *a*) *bs*)

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

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

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

```

apply(induct as)
apply simp
apply(clarsimp simp add:fun-upd-apply)
done

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

0.1.3 Functional abstraction

```

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

```

```

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

```

```

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

```

```

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

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

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

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

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

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

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

```

0.2 Verifications

0.2.1 List reversal

A short but unreadable proof:

```

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

```

apply *fastforce*
done

A longer readable version:

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

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

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

```

apply vcg-simp
  apply clarsimp
  apply clarsimp
apply clarsimp
done

```

0.2.2 Searching in a list

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

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

```

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

```

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

```

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

```

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

```

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

```

```

  {p = X}
apply vcg-simp
apply clarsimp
apply(erule converse-rtranclE)
  apply simp
apply(simp)
apply(fastforce elim:converse-rtranclE)
done

```

0.2.3 Merging two lists

This is still a bit rough, especially the proof.

```

fun merge :: 'a list * 'a list * ('a ⇒ 'a ⇒ bool) ⇒ '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 ⟶ R) = ((P ⟶ R) ∧ (¬P ⟶ Q ⟶ R))
by blast

```

```

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

```

```

lemma VARS hd tl p q r s
  {List tl p Ps ∧ List tl q Qs ∧ set Ps ∩ set Qs = {} ∧
   (p ≠ Null ∨ q ≠ Null)}
  IF if q = Null then True else 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. Path tl r rs s ∧ List tl p ps ∧ List tl q qs ∧
       distinct(s # ps @ qs @ rs) ∧ s ≠ Null ∧
       merge(Ps,Qs,λx y. hd x ≤ hd y) =
       rs @ s # merge(ps,qs,λx y. hd x ≤ hd y) ∧
       (tl s = p ∨ tl s = q)}
  DO IF if q = Null then True else p ≠ Null ∧ p^.hd ≤ q^.hd
  THEN s^.tl := p; p := p^.tl ELSE s^.tl := q; q := q^.tl FI;
  s := s^.tl
  OD
  {List tl r (merge(Ps,Qs,λx y. hd x ≤ hd y))}
apply vcg-simp

```

```

apply (fastforce)

```

```

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

```

```

apply(rule-tac  $x = rs @ [s]$  in exI)
apply simp
apply(rule-tac  $x = bs$  in exI)
apply (fastforce simp: eq-sym-conv)

```

```

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

```

```

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

```

```

apply(clarsimp simp add: List-app)
done

```

0.2.4 Storage allocation

definition *new* :: 'a set \Rightarrow 'a::ref
 where *new* A = (SOME a. a \notin A & a \neq Null)

lemma *new-notin*:

```

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

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

end

```

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

0.2.5 References

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

lemma *not-Null-eq* [*iff*]: ($x \sim = \text{Null}$) = ($\exists x. x = \text{Ref } y$)
by (*induct x*) *auto*

lemma *not-Ref-eq* [*iff*]: ($\forall y. x \sim = \text{Ref } y$) = ($x = \text{Null}$)
by (*induct x*) *auto*

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

0.3 The heap

0.3.1 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* $\longleftrightarrow x = y$

| $Path\ h\ x\ (a\#\ as)\ y \iff x = Ref\ a \wedge Path\ h\ (h\ a)\ as\ y$

lemma [*iff*]: $Path\ h\ Null\ xs\ y = (xs = [] \wedge y = Null)$
apply (*case-tac xs*)
apply *fastforce*
apply *fastforce*
done

lemma [*simp*]: $Path\ h\ (Ref\ a)\ as\ z =$
 $(as = [] \wedge z = Ref\ a \vee (\exists bs. as = a\#\ bs \wedge Path\ h\ (h\ a)\ bs\ z))$
apply (*case-tac as*)
apply *fastforce*
apply *fastforce*
done

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

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

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

0.3.2 Non-repeating paths

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

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

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

0.3.3 Lists on the heap

Relational abstraction

definition *List* :: $('a \Rightarrow 'a\ ref) \Rightarrow 'a\ ref \Rightarrow 'a\ list \Rightarrow bool$

where $List\ h\ x\ as = Path\ h\ x\ as\ Null$

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

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

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

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

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

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

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

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

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

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

lemma $Path-is-List:$
 $[[Path\ h\ b\ Ps\ (Ref\ a); a \notin set\ Ps]] \implies List\ (h(a := Null))\ b\ (Ps\ @\ [a])$
apply ($induct\ Ps\ arbitrary: b$)
apply ($auto\ simp\ add:fun-upd-apply$)
done

0.3.4 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)
apply(*simp* add:*islist-def* *list-def*)
apply(*rule* *iffI*)
apply(*rule* *conjI*)
apply *blast*
apply(*subst* *some1-equality*)
apply(*erule* *List-unique1*)
apply *assumption*
apply(*rule* *refl*)
apply *simp*
apply(*rule* *someI-ex*)
apply *fast*
done

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

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

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

lemma *list-Ref-conv*[*simp*]:
islist h (h a) ⇒ *list* h (Ref a) = a # *list* h (h a)
apply(*insert* *List-Ref*[*of* h])
apply(*fastforce* *simp*:*List-conv-islist-list*)
done

lemma [*simp*]: *islist* h (h a) ⇒ a ∉ set(*list* h (h a))
apply(*insert* *List-hd-not-in-tl*[*of* h])
apply(*simp* add:*List-conv-islist-list*)
done

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

lemma *islist-upd*[*simp*]:
islist h p ⇒ y ∉ set(*list* h p) ⇒ *islist* (h(y := q)) p

```

apply(frule notin-List-update[of - - h q p])
apply(simp add:List-conv-islist-list)
done

end

```

```

theory HeapSyntax imports Hoare-Logic Heap begin

```

0.3.5 Field access and update

syntax

```

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

```

translations

```

f(r → v) == f(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^.v := 3; y^.n := z;
z^.v := 4; x^.n := z
{w^.n^.n^.v = 4}

```

by *vcg-simp*

end

```

theory Pointer-Examples imports HeapSyntax begin

```

```

axiomatization where unproven: PROP A

```

0.4 Verifications

0.4.1 List reversal

A short but unreadable proof:

```

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

apply fastforce
done

```

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)}
apply vcg-simp
apply fastforce
apply fastforce
done

```

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 vcg
  fix tl p q r
  assume List tl p Ps ∧ List tl q Qs ∧ set Ps ∩ set Qs = {}
  thus ∃ ps qs. List tl p ps ∧ List tl q qs ∧ set ps ∩ set qs = {} ∧
      rev ps @ qs = rev Ps @ Qs by fastforce
next
  fix tl p q r

```

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

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

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

0.4.2 Searching in a list

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

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

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

```

    {p = Ref X}
apply vcg-simp
    apply blast
    apply clarsimp
apply clarsimp
done

```

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

```

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

```

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

```

lemma VARs tl p
  {(p,X) ∈ {(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}
apply vcg-simp
apply clarsimp
apply(erule converse-rtranclE)
apply simp
apply(clarsimp elim:converse-rtranclE)
apply(fast elim:converse-rtranclE)
done

```

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

```

lemma VARs tl p
  {p ≠ 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}
apply vcg-simp
apply clarsimp
apply(erule converse-rtranclE)
apply simp

```

```

apply clarsimp
apply clarsimp
done

```

0.4.3 Splicing two lists

```

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

```

```

apply vcg-simp
  apply(rule-tac x = [] in exI)
  apply fastforce
  apply clarsimp
  apply(rename-tac y bs qqs)
  apply(case-tac bs) apply simp
  apply clarsimp
  apply(rename-tac x bbs)
  apply(rule-tac x = as @ [x,y] in exI)
  apply simp
  apply(rule-tac x = bbs in exI)
  apply simp
  apply(rule-tac x = qqs in exI)
  apply simp
apply (fastforce simp:List-app)
done

```

0.4.4 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)
    else y # merge(x#xs,ys,f))
  | merge(x#xs,[],f) = x # merge(xs,[],f)
  | merge([],y#ys,f) = y # merge([],ys,f)
  | merge([],[],f) = []

```

Simplifies the proof a little:

```

lemma [simp]: ( $\{\} = \text{insert } a \ A \cap B$ ) = ( $a \notin B \ \& \ \{\} = A \cap B$ )
by blast
lemma [simp]: ( $\{\} = A \cap \text{insert } b \ B$ ) = ( $b \notin A \ \& \ \{\} = A \cap B$ )
by blast
lemma [simp]: ( $\{\} = A \cap (B \cup C)$ ) = ( $\{\} = A \cap B \ \& \ \{\} = A \cap C$ )
by blast

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

apply (fastforce)

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

```

```

apply(clarsimp simp:eq-sym-conv)
apply(rule-tac x = bsa in exI)
apply(simp)
apply (fastforce intro!:Path-snoc intro:Path-upd[THEN iffD2] notin-List-update[THEN
iffD2] simp:eq-sym-conv)

```

```

apply(clarsimp simp add:List-app)
done

```

And now with ghost variables:

```

lemma VARS elem next p q r s ps qs rs a
  {List next p Ps  $\wedge$  List next q Qs  $\wedge$  set Ps  $\cap$  set Qs = {}  $\wedge$ 
  (p  $\neq$  Null  $\vee$  q  $\neq$  Null)  $\wedge$  ps = Ps  $\wedge$  qs = Qs}
  IF cor (q = Null) (cand (p  $\neq$  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 := addr s;
  WHILE p  $\neq$  Null  $\vee$  q  $\neq$  Null
  INV {Path next r rs s  $\wedge$  List next p ps  $\wedge$  List next q qs  $\wedge$ 
  distinct(a # ps @ qs @ rs)  $\wedge$  s = Ref a  $\wedge$ 
  merge(Ps,Qs, $\lambda$ x y. elem x  $\leq$  elem y) =
  rs @ a # merge(ps,qs, $\lambda$ x y. elem x  $\leq$  elem y)  $\wedge$ 
  (next a = p  $\vee$  next a = q)}
  DO IF cor (q = Null) (cand (p  $\neq$  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 := addr s
  OD
  {List next r (merge(Ps,Qs, $\lambda$ x y. elem x  $\leq$  elem y))}
apply vcg-simp
apply (simp-all add: cand-def cor-def)

```

```

apply (fastforce)

```

```

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

```

```

apply(clarsimp simp add:List-app)
done

```

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

More of the previous proof without ghost variables can be automated,

but the runtime goes up drastically. In general it is usually more efficient to give the witness directly than to have it found by proof.

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

axiomatization

```
ispath :: ('a => 'a ref) => 'a ref => 'a ref => bool and
path  :: ('a => 'a ref) => 'a ref => 'a ref => 'a list
```

First some basic lemmas:

```
lemma [simp]: ispath f p p
by (rule unproven)
lemma [simp]: path f p p = []
by (rule unproven)
lemma [simp]: ispath f p q => a ∉ set(path f p q) => ispath (f(a := r)) p q
by (rule unproven)
lemma [simp]: ispath f p q => a ∉ set(path f p q) =>
  path (f(a := r)) p q = path f p q
by (rule unproven)
```

Some more specific lemmas needed by the example:

```
lemma [simp]: ispath (f(a := q)) p (Ref a) => ispath (f(a := q)) p q
by (rule unproven)
lemma [simp]: ispath (f(a := q)) p (Ref a) =>
  path (f(a := q)) p q = path (f(a := q)) p (Ref a) @ [a]
by (rule unproven)
lemma [simp]: ispath f p (Ref a) => f a = Ref b =>
  b ∉ set (path f p (Ref a))
by (rule unproven)
lemma [simp]: ispath f p (Ref a) => f a = Null => islist f p
by (rule unproven)
lemma [simp]: ispath f p (Ref a) => f a = Null => list f p = path f p (Ref a) @
[a]
by (rule unproven)

lemma [simp]: islist f p => distinct (list f p)
by (rule unproven)
```

```
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 @ 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
{islist tl r & list tl r = (merge(Ps,Qs,λx y. hd x ≤ hd y))}
apply vcg-simp

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

```

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

0.4.5 Cyclic list reversal

We consider two algorithms for the reversal of circular lists.

lemma *circular-list-rev-I*:

```

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

```

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

The precondition states that there exists a non-empty non-repeating path $r \# Ps$ from pointer *root* to itself, given that *root* points to location *r*. Pointers *p* and *q* are then set to *root* and the successor of *root* respectively. If $q = root$, we have circled the loop, otherwise we set the *next* pointer field

of q to point to p , and shift p and q one step forward. The invariant thus states that p and q point to two disjoint lists ps and qs , such that $Ps = rev\ ps\ @\ qs$. After the loop terminates, one extra step is needed to close the loop. As expected, the postcondition states that the *distPath* from *root* to itself is now $r\ \#\ rev\ Ps$.

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

lemma *circular-list-rev-II*:

VARs $next\ p\ q\ tmp$

$\{p = Ref\ r \wedge distPath\ next\ p\ (r\ \#\ Ps)\ p\}$

$q := Null;$

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 $tmp := p; p := p.^next; tmp.^next := q; q := tmp$ *OD*

$\{q = Ref\ r \wedge distPath\ next\ q\ (r\ \#\ rev\ Ps)\ q\}$

apply (*simp only:distPath-def*)

apply *vcg-simp*

apply *clarsimp*

apply *clarsimp*

apply (*case-tac* ($q = Null$))

apply (*fastforce intro: Path-is-List*)

apply *clarsimp*

apply (*rule-tac* $x = bs$ **in** *exI*)

apply (*rule-tac* $x = y\ \#\ qs$ **in** *exI*)

apply *clarsimp*

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

done

0.4.6 Storage allocation

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

where $new\ A = (SOME\ a. a \notin A)$

lemma *new-notin*:

$\llbracket \sim finite(UNIV :: 'a\ set); finite(A :: 'a\ set); B \subseteq A \rrbracket \Longrightarrow new\ (A) \notin B$

apply (*unfold new-def*)

apply (*rule someI2-ex*)

apply (*fast intro:ex-new-if-finite*)

apply (*fast*)

done

lemma $\sim finite(UNIV :: 'a\ set) \Longrightarrow$

```

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

```

end

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

0.4.7 Field access and update

Heap update $p^{\wedge}.h := e$ 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  $\Rightarrow$  'b)  $\Rightarrow$  'a ref  $\Rightarrow$  'b  $\Rightarrow$  ('a  $\Rightarrow$  'b)
  (-/'((-  $\rightarrow$  -)') [1000,0] 900)
-fassign :: 'a ref  $\Rightarrow$  id  $\Rightarrow$  'v  $\Rightarrow$  's com
  ((2-^.- :=/ -) [70,1000,65] 61)
-faccess :: 'a ref  $\Rightarrow$  ('a ref  $\Rightarrow$  'v)  $\Rightarrow$  'v
  (-^.- [65,1000] 65)

```

translations

```

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

```

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

An example due to Suzuki:

lemma *VARs v n*

```

{w = Ref w0 & x = Ref x0 & y = Ref y0 & z = Ref z0 &
  distinct[w0,x0,y0,z0]}
```

```

  w^.v := (1::int); w^.n := x;
  x^.v := 2; x^.n := y;
  y^.v := 3; y^.n := z;
  z^.v := 4; x^.n := z
  {w^.n^.n^.v = 4}
by vcg-simp

end

```

```

theory Pointer-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)}
apply vcg-simp
  apply fastforce
  apply (fastforce intro:notin-List-update[THEN iffD2])
  apply fastforce
done

end

```

```

theory SchorrWaite imports HeapSyntax begin

```

0.6 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

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

definition

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

lemmas $rel-defs = relS-def \ rel-def$

Rewrite rules for relations induced by a mapping

lemma $self-reachable: b \in B \Longrightarrow b \in R^* \text{ `` } B$
apply $blast$
done

lemma $oneStep-reachable: b \in R \text{ `` } B \Longrightarrow b \in R^* \text{ `` } B$
apply $blast$
done

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

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

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

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

lemma $reachable-union: (reachable \ mS \ aS \cup reachable \ mS \ bS) = reachable \ mS \ (aS \cup bS)$
apply $(simp \ add: reachable-def \ rel-defs \ addr s-def)$
apply $blast$

done

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

apply (*simp add: reachable-def rel-defs addrs-def*)
apply *blast*
done

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

apply (*rule classical*)
apply (*simp add:rel-defs fun-upd-apply*)
done

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

apply (*rule classical*)
apply (*simp add:rel-defs fun-upd-apply*)
done

definition

— Restriction of a relation

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

where *restr* r m = $\{(x,y). (x,y) \in r \wedge \neg m\ x\}$

Rewrite rules for the restriction of a relation

lemma *restr-identity*[*simp*]:

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

by (*auto simp add:restr-def*)

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

by (*auto simp add:restr-def elim:converse-rtranclE*)

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

by (*auto simp add:restr-def elim:converse-rtranclE*)

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

apply (*auto simp:restr-def rel-def fun-upd-apply*)

apply (*rename-tac a b*)

apply (*case-tac a=q*)

apply *auto*

done

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

by (*auto simp add:restr-def*)

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

apply (*rule classical*)

apply (*simp add:restr-def fun-upd-apply*)

done

definition

— A short form for the stack mapping function for List

$S :: ('a \Rightarrow \text{bool}) \Rightarrow ('a \Rightarrow 'a \text{ ref}) \Rightarrow ('a \Rightarrow 'a \text{ ref}) \Rightarrow ('a \Rightarrow 'a \text{ ref})$
where $S \ c \ l \ r = (\lambda x. \text{if } c \ x \ \text{then } r \ x \ \text{else } l \ x)$

Rewrite rules for Lists using S as their mapping

lemma [rule-format,simp]:

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

apply(*induct-tac stack*)

apply(*simp add:fun-upd-apply S-def*)+

done

lemma [rule-format,simp]:

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

apply(*induct-tac stack*)

apply(*simp add:fun-upd-apply S-def*)+

done

lemma [rule-format,simp]:

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

apply(*induct-tac stack*)

apply(*simp add:fun-upd-apply S-def*)+

done

lemma [rule-format,simp]:

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

apply(*induct-tac stack*)

apply(*simp add:fun-upd-apply S-def*)+

done

primrec

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

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

where

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

| *stkOk-cons*:

$\text{stkOk } c \ l \ r \ iL \ iR \ t \ (p\#\text{stk}) = (\text{stkOk } c \ l \ r \ iL \ iR \ (\text{Ref } p) \ (\text{stk})) \wedge$

$iL \ p = (\text{if } c \ p \ \text{then } l \ p \ \text{else } t) \wedge$

$iR \ p = (\text{if } c \ p \ \text{then } t \ \text{else } r \ p)$

Rewrite rules for stkOk

lemma [simp]: $\bigwedge t. [] \notin \text{set } xs; \text{Ref } x \neq t \implies$

$\text{stkOk } (c(x := f)) \ l \ r \ iL \ iR \ t \ xs = \text{stkOk } c \ l \ r \ iL \ iR \ t \ xs$

apply (*induct xs*)

apply (*auto simp:eq-sym-conv*)

done

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

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

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

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

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

0.7 The Schorr-Waite algorithm

theorem *SchorrWaiteAlgorithm*:

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

```

    THEN IF  $p^{\wedge}.c$ 
      THEN  $q := t; t := p; p := p^{\wedge}.r; t^{\wedge}.r := q$  (*pop*)
      ELSE  $q := t; t := p^{\wedge}.r; p^{\wedge}.r := p^{\wedge}.l;$  (*swing*)
            $p^{\wedge}.l := q; p^{\wedge}.c := \text{True}$  FI
    ELSE  $q := p; p := t; t := t^{\wedge}.l; p^{\wedge}.l := q;$  (*push*)
          $p^{\wedge}.m := \text{True}; p^{\wedge}.c := \text{False}$  FI OD
   $\{(\forall x. (x \in R) = m x) \wedge (r = iR \wedge l = iL)\}$ 
  (is VARS  $c m l r t p q \text{root}$   $\{?Pre\ c\ m\ l\ r\ \text{root}\}$   $(?c1; ?c2; ?c3)$   $\{?Post\ c\ m\ l\ r\}$ )
proof (vcg)
  let While  $\{(c, m, l, r, t, p, q, \text{root}). ?whileB\ m\ t\ p\}$ 
     $\{(c, m, l, r, t, p, q, \text{root}). ?inv\ c\ m\ l\ r\ t\ p\}$   $?body = ?c3$ 
  {

    fix  $c\ m\ l\ r\ t\ p\ q\ \text{root}$ 
    assume  $?Pre\ c\ m\ l\ r\ \text{root}$ 
    thus  $?inv\ c\ m\ l\ r\ \text{root}\ \text{Null}$  by (auto simp add: reachable-def adrs-def)
  next

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

  next

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

    assume  $(\exists\ \text{stack}. ?Inv\ \text{stack}) \wedge (p \neq \text{Null} \vee t \neq \text{Null} \wedge \neg t^{\wedge}.m)$  (is -  $\wedge$ 
     $?whileB$ )
    then obtain stack where  $inv: ?Inv\ \text{stack}$  and  $whileB: ?whileB$  by blast
    let  $?I1 \wedge ?I2 \wedge ?I3 \wedge ?I4 \wedge ?I5 \wedge ?I6 \wedge ?I7 = ?Inv\ \text{stack}$ 
    from inv have  $i1: ?I1$  and  $i2: ?I2$  and  $i3: ?I3$  and  $i4: ?I4$ 
      and  $i5: ?I5$  and  $i6: ?I6$  and  $i7: ?I7$  by simp+
    have  $stackDist: \text{distinct}\ (\text{stack})$  using i1 by (rule List-distinct)

```

```

show (?ifB1  $\longrightarrow$  (?ifB2  $\longrightarrow$  ( $\exists$  stack. ?popInv stack))  $\wedge$ 
      ( $\neg$ ?ifB2  $\longrightarrow$  ( $\exists$  stack. ?swInv stack)))  $\wedge$ 
      ( $\neg$ ?ifB1  $\longrightarrow$  ( $\exists$  stack. ?puInv stack))
proof –
  {
    assume ifB1:  $t = \text{Null} \vee t.^m$  and ifB2:  $p.^c$ 
    from ifB1 whileB have pNotNull:  $p \neq \text{Null}$  by auto
    then obtain addr-p where addr-p-eq:  $p = \text{Ref } \text{addr-p}$  by auto
    with i1 obtain stack-tl where stack-eq:  $\text{stack} = (\text{addr } p) \# \text{stack-tl}$ 
      by auto
    with i2 have m-addr-p:  $p.^m$  by auto
    have stackDist: distinct (stack) using i1 by (rule List-distinct)
    from stack-eq stackDist have p-notin-stack-tl:  $\text{addr } p \notin \text{set } \text{stack-tl}$  by
simp
let ?poI1  $\wedge$  ?poI2  $\wedge$  ?poI3  $\wedge$  ?poI4  $\wedge$  ?poI5  $\wedge$  ?poI6  $\wedge$  ?poI7 = ?popInv stack-tl
have ?popInv stack-tl
proof –

    – List property is maintained:
    from i1 p-notin-stack-tl ifB2
    have poI1: List (S c l (r(p  $\rightarrow$  t))) (p.^r) stack-tl
      by(simp add: addr-p-eq stack-eq, simp add: S-def)

    moreover
    – Everything on the stack is marked:
    from i2 have poI2:  $\forall x \in \text{set } \text{stack-tl}. m \ x$  by (simp add:stack-eq)
    moreover

    – Everything is still reachable:
    let (R = reachable ?Ra ?A) = ?I3
    let ?Rb = (relS {l, r(p  $\rightarrow$  t)})
    let ?B = {p, p.^r}
    – Our goal is  $R = \text{reachable } ?Rb \ ?B$ .
    have ?Ra* “ addrs ?A = ?Rb* “ addrs ?B (is ?L = ?R)
    proof
      show ?L  $\subseteq$  ?R
      proof (rule still-reachable)
        show addrs ?A  $\subseteq$  ?Rb* “ addrs ?B by(fastforce simp:addr-def
relS-def rel-def addr-p-eq
      intro:oneStep-reachable Image-iff[THEN iffD2])
        show  $\forall (x,y) \in ?Ra - ?Rb. y \in (?Rb^* \text{ “ } \text{addrs } ?B)$  by (clarsimp
simp:relS-def)
          (fastforce simp add:rel-def Image-iff addr-def dest:rel-upd1)
      qed
      show ?R  $\subseteq$  ?L
      proof (rule still-reachable)
        show addrs ?B  $\subseteq$  ?Ra* “ addrs ?A
          by(fastforce simp:addr-def rel-defs addr-p-eq

```


— If it is marked, then it is reachable
from *i5* **have** *poI5*: $\forall x. m\ x \longrightarrow x \in R$.
moreover

— If it is not on the stack, then its *l* and *r* fields are unchanged
from *i7* *i6* *ifB2*
have *poI6*: $\forall x. x \notin \text{set } \text{stack-tl} \longrightarrow (r(p \rightarrow t))\ x = iR\ x \wedge l\ x = iL\ x$
by (*auto simp: addr-p-eq stack-eq fun-upd-apply*)

moreover

— If it is on the stack, then its *l* and *r* fields can be reconstructed
from *p-notin-stack-tl* *i7* **have** *poI7*: *stkOk c l (r(p → t)) iL iR p stack-tl*
by (*clarsimp simp:stack-eq addr-p-eq*)

ultimately show *?popInv stack-tl* **by** *simp*
qed
hence $\exists \text{stack}. ?\text{popInv } \text{stack} \dots$
}
moreover

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

— Swing arm
assume *ifB1*: *?ifB1* **and** *nifB2*: $\neg ?\text{ifB2}$
from *ifB1* *whileB* **have** *pNotNull*: $p \neq \text{Null}$ **by** *clarsimp*
then obtain *addr-p* **where** *addr-p-eq*: $p = \text{Ref } \text{addr-p}$ **by** *clarsimp*
with *i1* **obtain** *stack-tl* **where** *stack-eq*: $\text{stack} = (\text{addr } p) \# \text{stack-tl}$ **by**
clarsimp
with *i2* **have** *m-addr-p*: $p \hat{=} m$ **by** *clarsimp*
from *stack-eq stackDist* **have** *p-notin-stack-tl*: $(\text{addr } p) \notin \text{set } \text{stack-tl}$
by *simp*
let $?swI1 \wedge ?swI2 \wedge ?swI3 \wedge ?swI4 \wedge ?swI5 \wedge ?swI6 \wedge ?swI7 = ?swInv\ \text{stack}$
have *?swInv stack*
proof —

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

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

— Everything is still reachable:

```

let  $R = \text{reachable } ?Ra \ ?A = ?I3$ 
let  $R = \text{reachable } ?Rb \ ?B = ?swI3$ 
have  $?Ra^* \text{ `` } \text{addrs } ?A = ?Rb^* \text{ `` } \text{addrs } ?B$ 
proof (rule still-reachable-eq)
  show  $\text{addrs } ?A \subseteq ?Rb^* \text{ `` } \text{addrs } ?B$ 
  by (fastforce simp:addrs-def rel-defs addr-p-eq intro:oneStep-reachable
Image-iff [THEN iffD2])
  next
  show  $\text{addrs } ?B \subseteq ?Ra^* \text{ `` } \text{addrs } ?A$ 
  by (fastforce simp:addrs-def rel-defs addr-p-eq intro:oneStep-reachable
Image-iff [THEN iffD2])
  next
  show  $\forall (x, y) \in ?Ra - ?Rb. y \in (?Rb^* \text{ `` } \text{addrs } ?B)$ 
  by (clarsimp simp:relS-def) (fastforce simp add:rel-def Image-iff
addrs-def fun-upd-apply dest:rel-upd1)
  next
  show  $\forall (x, y) \in ?Rb - ?Ra. y \in (?Ra^* \text{ `` } \text{addrs } ?A)$ 
  by (clarsimp simp:relS-def) (fastforce simp add:rel-def Image-iff
addrs-def fun-upd-apply dest:rel-upd2)
qed
with  $i3$ 
have  $swI3: ?swI3$  by (simp add:reachable-def)
moreover

```

— If it is reachable and not marked, it is still reachable using...

```

let  $\forall x. x \in R \wedge \neg m \ x \longrightarrow x \in \text{reachable } ?Ra \ ?A = ?I4$ 
let  $\forall x. x \in R \wedge \neg m \ x \longrightarrow x \in \text{reachable } ?Rb \ ?B = ?swI4$ 
let  $?T = \{t\}$ 
have  $?Ra^* \text{ `` } \text{addrs } ?A \subseteq ?Rb^* \text{ `` } (\text{addrs } ?B \cup \text{addrs } ?T)$ 
proof (rule still-reachable)
  have rewrite:  $(\forall s \in \text{set } \text{stack-tl}. (r(\text{addr } p := l(\text{addr } p))) \ s = r \ s)$ 
  by (auto simp add:p-notin-stack-tl intro:fun-upd-other)
  show  $\text{addrs } ?A \subseteq ?Rb^* \text{ `` } (\text{addrs } ?B \cup \text{addrs } ?T)$ 
  by (fastforce cong:map-cong simp:stack-eq addrs-def rewrite in-
tro:self-reachable)
  next
  show  $\forall (x, y) \in ?Ra - ?Rb. y \in (?Rb^* \text{ `` } (\text{addrs } ?B \cup \text{addrs } ?T))$ 
  by (clarsimp simp:relS-def restr-def) (fastforce simp add:rel-def
Image-iff addrs-def fun-upd-apply dest:rel-upd1)
qed
then have subset:  $?Ra^* \text{ `` } \text{addrs } ?A - ?Rb^* \text{ `` } \text{addrs } ?T \subseteq ?Rb^* \text{ `` } \text{addrs } ?B$ 
  by blast
have  $?swI4$ 
proof (rule allI, rule impI)
  fix  $x$ 
  assume  $a: x \in R \wedge \neg m \ x$ 
  with  $i4$  addr-p-eq stack-eq have inc:  $x \in ?Ra^* \text{ `` } \text{addrs } ?A$ 

```

```

    by (simp only:reachable-def, clarsimp)
  with ifB1 a
  have exc:  $x \notin ?Rb^*$  “  $addrs ?T$ 
    by (auto simp add:addrs-def)
  from inc exc subset show  $x \in reachable ?Rb ?B$ 
    by (auto simp add:reachable-def)
qed
moreover

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

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

— If it is on the stack, then its  $l$  and  $r$  fields can be reconstructed
from stackDist i7 nifB2
have ?swI7
  by (clarsimp simp:addr-p-eq stack-eq)

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

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

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

```

— Everything on the stack is marked:

```

from  $i2$ 
have  $puI2: ?puI2$ 
  by ( $simp\ add:new-stack-eq\ fun-upd-apply$ )
moreover

```

— Everything is still reachable:

```

let  $R = reachable\ ?Ra\ ?A = ?I3$ 
let  $R = reachable\ ?Rb\ ?B = ?puI3$ 
have  $?Ra^* \text{ `` } \textit{addrs}\ ?A = ?Rb^* \text{ `` } \textit{addrs}\ ?B$ 
proof ( $rule\ still-reachable-eq$ )
  show  $\textit{addrs}\ ?A \subseteq ?Rb^* \text{ `` } \textit{addrs}\ ?B$ 
    by ( $fastforce\ simp:addrs-def\ rel-defs\ addr-t-eq\ intro:oneStep-reachable$ 
 $Image-iff[THEN\ iffD2]$ )
  next
    show  $\textit{addrs}\ ?B \subseteq ?Ra^* \text{ `` } \textit{addrs}\ ?A$ 
    by ( $fastforce\ simp:addrs-def\ rel-defs\ addr-t-eq\ intro:oneStep-reachable$ 
 $Image-iff[THEN\ iffD2]$ )
  next
    show  $\forall (x, y) \in ?Ra - ?Rb. y \in (?Rb^* \text{ `` } \textit{addrs}\ ?B)$ 
    by ( $clarsimp\ simp:relS-def$ ) ( $fastforce\ simp\ add:rel-def\ Image-iff$ 
 $addrs-def\ dest:rel-upd1$ )
  next
    show  $\forall (x, y) \in ?Rb - ?Ra. y \in (?Ra^* \text{ `` } \textit{addrs}\ ?A)$ 
    by ( $clarsimp\ simp:relS-def$ ) ( $fastforce\ simp\ add:rel-def\ Image-iff$ 
 $addrs-def\ fun-upd-apply\ dest:rel-upd2$ )
qed
with  $i3$ 
have  $puI3: ?puI3$  by ( $simp\ add:reachable-def$ )
moreover

```

— If it is reachable and not marked, it is still reachable using...

```

let  $\forall x. x \in R \wedge \neg m\ x \longrightarrow x \in reachable\ ?Ra\ ?A = ?I4$ 
let  $\forall x. x \in R \wedge \neg ?new-m\ x \longrightarrow x \in reachable\ ?Rb\ ?B = ?puI4$ 
let  $?T = \{t\}$ 
have  $?Ra^* \text{ `` } \textit{addrs}\ ?A \subseteq ?Rb^* \text{ `` } (\textit{addrs}\ ?B \cup \textit{addrs}\ ?T)$ 
proof ( $rule\ still-reachable$ )
  show  $\textit{addrs}\ ?A \subseteq ?Rb^* \text{ `` } (\textit{addrs}\ ?B \cup \textit{addrs}\ ?T)$ 
    by ( $fastforce\ simp:new-stack-eq\ addrs-def\ intro:self-reachable$ )
  next
    show  $\forall (x, y) \in ?Ra - ?Rb. y \in (?Rb^* \text{ `` } (\textit{addrs}\ ?B \cup \textit{addrs}\ ?T))$ 
    by ( $clarsimp\ simp:relS-def\ new-stack-eq\ restr-un\ restr-upd$ )
    ( $fastforce\ simp\ add:rel-def\ Image-iff\ restr-def\ addrs-def\ fun-upd-apply$ 
 $addr-t-eq\ dest:rel-upd3$ )
qed
then have  $subset: ?Ra^* \text{ `` } \textit{addrs}\ ?A - ?Rb^* \text{ `` } \textit{addrs}\ ?T \subseteq ?Rb^* \text{ `` } \textit{addrs}\ ?B$ 
  by  $blast$ 
have  $puI4$ 

```

```

proof (rule allI, rule impI)
  fix x
  assume a:  $x \in R \wedge \neg ?new\text{-}m\ x$ 
  have xDisj:  $x = (\text{addr } t) \vee x \neq (\text{addr } t)$  by simp
  with i4 a have inc:  $x \in ?Ra^* \text{ `` } \text{addrs } ?A$ 
  by (fastforce simp:addr-t-eq addrs-def reachable-def intro:self-reachable)
  have exc:  $x \notin ?Rb^* \text{ `` } \text{addrs } ?T$ 
  using xDisj a n-m-addr-t
  by (clarsimp simp add:addrs-def addr-t-eq)
  from inc exc subset show  $x \in \text{reachable } ?Rb\ ?B$ 
  by (auto simp add:reachable-def)
qed
moreover

— If it is marked, then it is reachable
from i5
have ?puI5
  by (auto simp:addrs-def i3 reachable-def addr-t-eq fun-upd-apply
intro:self-reachable)
moreover

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

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

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

```

```

theory SepLogHeap
imports Main
begin

```

type-synonym $heap = (nat \Rightarrow nat\ option)$

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

0.7.1 Paths in the heap

primrec $Path :: heap \Rightarrow nat \Rightarrow nat\ list \Rightarrow nat \Rightarrow bool$

where

$Path\ h\ x\ []\ y = (x = y)$
 $| Path\ h\ x\ (a\#\ as)\ y = (x \neq 0 \wedge a = x \wedge (\exists b. h\ x = Some\ b \wedge Path\ h\ b\ as\ y))$

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

by $(cases\ xs)\ simp\ all$

lemma $[simp]$: $x \neq 0 \implies Path\ h\ x\ as\ z =$

$(as = [] \wedge z = x \vee (\exists y\ bs. as = x\#\ bs \wedge h\ x = Some\ y \ \&\ Path\ h\ y\ bs\ z))$

by $(cases\ as)\ auto$

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

by $(induct\ as)\ auto$

lemma $Path\ upd[simp]$:

$\bigwedge x. u \notin set\ as \implies Path\ (f(u := v))\ x\ as\ y = Path\ f\ x\ as\ y$

by $(induct\ as)\ simp\ all$

0.7.2 Lists on the heap

definition $List :: heap \Rightarrow nat \Rightarrow nat\ list \Rightarrow bool$

where $List\ h\ x\ as = Path\ h\ x\ as\ 0$

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

by $(simp\ add:\ List\ def)$

lemma $[simp]$:

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

by $(simp\ add:\ List\ def)$

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

by $(cases\ as)\ simp\ all$

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

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

by $(cases\ as)\ simp\ all$

theorem $notin\ List\ update[simp]$:

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

by $(induct\ as)\ simp\ all$

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

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

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

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

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

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

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

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

end

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

The semantic definition of a few connectives:

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

definition *is-empty* :: $heap \Rightarrow bool$
where $is_empty\ h \longleftrightarrow h = empty$

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

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

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

This is what assertions look like without any syntactic sugar:

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

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

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

ML $\langle\langle$

(* *free-tr* takes care of free vars in the scope of sep. logic connectives:
they are implicitly applied to the heap *)

fun free-tr(*t as Free -*) = *t* \$ *Syntax.free H*

(*
| *free-tr*((*list as Free(List,-)*)\$ *p* \$ *ps*) = *list* \$ *Syntax.free H* \$ *p* \$ *ps*
*)
| *free-tr t* = *t*

fun emp-tr [] = *Syntax.const* @{*const-syntax is-empty*} \$ *Syntax.free H*
| *emp-tr ts* = *raise TERM* (*emp-tr*, *ts*);

fun singl-tr [*p*, *q*] = *Syntax.const* @{*const-syntax singl*} \$ *Syntax.free H* \$ *p* \$ *q*
| *singl-tr ts* = *raise TERM* (*singl-tr*, *ts*);

fun star-tr [*P,Q*] = *Syntax.const* @{*const-syntax star*} \$
absfree (*H*, *dummyT*) (*free-tr P*) \$ *absfree* (*H*, *dummyT*) (*free-tr Q*) \$
Syntax.free H

| *star-tr ts* = *raise TERM* (*star-tr*, *ts*);

fun wand-tr [*P, Q*] = *Syntax.const* @{*const-syntax wand*} \$
absfree (*H*, *dummyT*) *P* \$ *absfree* (*H*, *dummyT*) *Q* \$ *Syntax.free H*

| *wand-tr ts* = *raise TERM* (*wand-tr*, *ts*);

$\rangle\rangle$

parse-translation $\langle\langle$

(@{*syntax-const -emp*}, *emp-tr*),
(@{*syntax-const -singl*}, *singl-tr*),

```

  (@{syntax-const -star}, star-tr),
  (@{syntax-const -wand}, wand-tr)]
>>

```

Now it looks much better:

```

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

```

```

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

```

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

```

ML <<
  local

  fun strip (Abs(-,-,(t as Const(-free,-) $ Free -) $ Bound 0)) = t
    | strip (Abs(-,-,(t as Free -) $ Bound 0)) = t
  (*
  | strip (Abs(-,-,(list as Const(List,-))$ Bound 0 $ p $ ps))) = list$ps$ps
  *)
  | strip (Abs(-,-,(t as Const(-var,-) $ Var -) $ Bound 0)) = t
  | strip (Abs(-,-,P)) = P
  | strip (Const(@{const-syntax is-empty},-)) = Syntax.const @{syntax-const -emp}
  | strip t = t;

  in

  fun is-empty-tr' [-] = Syntax.const @{syntax-const -emp}
  fun singl-tr' [-,p,q] = Syntax.const @{syntax-const -singl} $ p $ q
  fun star-tr' [P,Q,-] = Syntax.const @{syntax-const -star} $ strip P $ strip Q
  fun wand-tr' [P,Q,-] = Syntax.const @{syntax-const -wand} $ strip P $ strip Q

  end
>>

print-translation <<
  [(@{const-syntax is-empty}, is-empty-tr'),
  (@{const-syntax singl}, singl-tr'),

```

```

  (@{const-syntax star}, star-tr'),
  (@{const-syntax wand}, wand-tr')
>>

```

Now the intermediate proof states are also readable:

```

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

```

```

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

```

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

```

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

```

```

lemma VARS  $H$   $x$   $y$   $z$   $w$ 
  { $P ** Q$ }
  SKIP
  { $Q ** P$ }
apply vcg
apply(simp add: star-comm)
done

```

```

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

```

```

lemma VARS  $H$   $p$   $q$   $r$ 
  {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  $Ps$  @  $Qs$ )}
apply vcg
apply(simp-all add: star-def ortho-def singl-def)

```

```

apply fastforce

```

```

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

```

```

apply(fastforce)
done

```

```

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.