Hoare Logic

Norbert Galm
Walter Guttmann
Farhad Mehta
Tobias Nipkow
Leonor Prensa Nieto

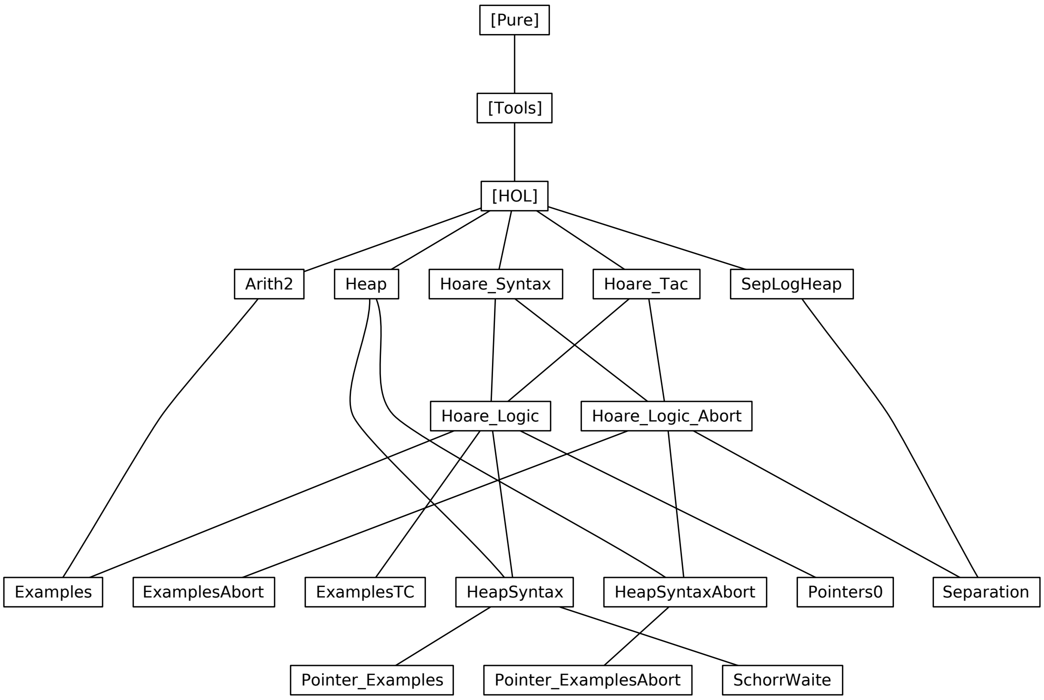
December 12, 2021

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


[Pure]
[Tools]
[HOL]
Arith2
Heap
Hoare_Syntax
Hoare_Tac
SepLogHeap
Hoare_Logic
Hoare_Logic_Abort
Examples
ExamplesAbort
ExamplesTC
HeapSyntax
HeapSyntaxAbort
Pointers0
Separation
Pointer_Examples
Pointer_ExamplesAbort
SchorrWaite

1 Concrete syntax for Hoare logic, with translations for variables

theory *Hoare-Syntax*
  imports *Main*
begin

syntax
  -assign :: *idt* ⇒ *'b* ⇒ *'com* ((*2- :=/ -*) [*70*, *65*] *61*)
  -Seq :: *'com* ⇒ *'com* ⇒ *'com* ((*-;/ -*) [*61*,*60*] *60*)
  -Cond :: *'bexp* ⇒ *'com* ⇒ *'com* ⇒ *'com* ((*1IF -/ THEN - / ELSE -/ FI*) [*0*,*0*,*0*] *61*)
  -While :: *'bexp* ⇒ *'assn* ⇒ *'var* ⇒ *'com* ⇒ *'com* ((*1WHILE -/ INV* {-} / *VAR* {-} //*DO - /OD*) [*0*,*0*,*0*,*0*] *61*)

The *VAR* {-} syntax supports two variants:

- *VAR* $\{x = t\}$ where t::*nat* is the decreasing expression, the variant, and x a variable that can be referred to from inner annotations. The x can be necessary for nested loops, e.g. to prove that the inner loops do not mess with t.
- *VAR* $\{t\}$ where the variable is omitted because it is not needed.

syntax
  -While0 :: *'bexp* ⇒ *'assn* ⇒ *'com* ⇒ *'com* ((*1WHILE -/ INV* {-} //*DO - /OD*) [*0*,*0*,*0*] *61*)

The *-While0* syntax is translated into the *-While* syntax with the trivial variant 0. This is ok because partial correctness proofs do not make use of the variant.

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

Completeness requires(?) the ability to refer to an outer variant in an inner invariant. Thus we need to abstract over a variable equated with the variant, the x in *VAR* $\{x = t\}$. But the x should only occur in invariants. To enforce this, syntax translations in *hoare-syntax.ML* separate the program from its annotations and only the latter are abstracted over over x. (Thus x can also occur in inner variants, but that neither helps nor hurts.)

datatype *'a anno* =
  Abasic |

Aseq 'a anno 'a anno |
Acond 'a anno 'a anno |
Awhile 'a set 'a ⇒ nat nat ⇒ 'a anno

⟨*ML*⟩

end

2 Hoare logic VCG tactic

theory *Hoare-Tac*
imports *Main*
begin

context
begin

qualified named-theorems *BasicRule*
qualified named-theorems *SkipRule*
qualified named-theorems *AbortRule*
qualified named-theorems *SeqRule*
qualified named-theorems *CondRule*
qualified named-theorems *WhileRule*

qualified named-theorems *BasicRuleTC*
qualified named-theorems *SkipRuleTC*
qualified named-theorems *SeqRuleTC*
qualified named-theorems *CondRuleTC*
qualified named-theorems *WhileRuleTC*

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

⟨*ML*⟩

end

end

3 Hoare logic

theory *Hoare-Logic*
imports *Hoare-Syntax Hoare-Tac*
begin

3.1 Sugared semantic embedding of Hoare logic

Strictly speaking a shallow embedding (as implemented by Norbert Galm following Mike Gordon) would suffice. Maybe the datatype com comes in useful later.

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

datatype $'a\ com =$
  $Basic\ 'a \Rightarrow 'a$
$|\ Seq\ 'a\ com\ 'a\ com$
$|\ Cond\ 'a\ bexp\ 'a\ com\ 'a\ com$
$|\ While\ 'a\ bexp\ 'a\ com$

abbreviation *annskip* (*SKIP*) **where** *SKIP* == *Basic id*

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

inductive $Sem :: 'a\ com \Rightarrow 'a\ sem$
where
  $Sem\ (Basic\ f)\ s\ (f\ s)$
$|\ Sem\ c1\ s\ s'' \Longrightarrow Sem\ c2\ s''\ s' \Longrightarrow Sem\ (Seq\ c1\ c2)\ s\ s'$
$|\ s \in b \Longrightarrow Sem\ c1\ s\ s' \Longrightarrow Sem\ (Cond\ b\ c1\ c2)\ s\ s'$
$|\ s \notin b \Longrightarrow Sem\ c2\ s\ s' \Longrightarrow Sem\ (Cond\ b\ c1\ c2)\ s\ s'$
$|\ s \notin b \Longrightarrow Sem\ (While\ b\ c)\ s\ s$
$|\ s \in b \Longrightarrow Sem\ c\ s\ s'' \Longrightarrow Sem\ (While\ b\ c)\ s''\ s' \Longrightarrow$
   $Sem\ (While\ b\ c)\ s\ s'$

definition $Valid :: 'a\ bexp \Rightarrow 'a\ com \Rightarrow 'a\ anno \Rightarrow 'a\ bexp \Rightarrow bool$
  where $Valid\ p\ c\ a\ q \equiv \forall s\ s'.\ Sem\ c\ s\ s' \longrightarrow s \in p \longrightarrow s' \in q$

definition $ValidTC :: 'a\ bexp \Rightarrow 'a\ com \Rightarrow 'a\ anno \Rightarrow 'a\ bexp \Rightarrow bool$
  where $ValidTC\ p\ c\ a\ q \equiv \forall s.\ s \in p \longrightarrow (\exists t.\ Sem\ c\ s\ t \wedge t \in q)$

inductive-cases [*elim*!]:
  $Sem\ (Basic\ f)\ s\ s'\ Sem\ (Seq\ c1\ c2)\ s\ s'$
  $Sem\ (Cond\ b\ c1\ c2)\ s\ s'$

lemma *Sem-deterministic*:
  assumes $Sem\ c\ s\ s1$
    and $Sem\ c\ s\ s2$
   shows $s1 = s2$
⟨*proof*⟩

lemma *tc-implies-pc*:
  $ValidTC\ p\ c\ a\ q \Longrightarrow Valid\ p\ c\ a\ q$
  ⟨*proof*⟩

lemma *tc-extract-function*:

$ValidTC\ p\ c\ a\ q \Longrightarrow \exists f\ .\ \forall s\ .\ s \in p \longrightarrow f\ s \in q$
⟨*proof*⟩

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

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

lemma *SeqRule*: $Valid\ P\ c1\ a1\ Q \Longrightarrow Valid\ Q\ c2\ a2\ R \Longrightarrow Valid\ P\ (Seq\ c1\ c2)\ (Aseq\ a1\ a2)\ R$
⟨*proof*⟩

lemma *CondRule*:
$p \subseteq \{s.\ (s \in b \longrightarrow s \in w) \wedge (s \notin b \longrightarrow s \in w')\}$
$\Longrightarrow Valid\ w\ c1\ a1\ q \Longrightarrow Valid\ w'\ c2\ a2\ q \Longrightarrow Valid\ p\ (Cond\ b\ c1\ c2)\ (Acond\ a1\ a2)\ q$
⟨*proof*⟩

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

lemma *WhileRule*:
$p \subseteq i \Longrightarrow Valid\ (i \cap b)\ c\ (A\ 0)\ i \Longrightarrow i \cap (-b) \subseteq q \Longrightarrow Valid\ p\ (While\ b\ c)\ (Awhile\ i\ v\ A)\ q$
⟨*proof*⟩

lemma *SkipRuleTC*:
assumes $p \subseteq q$
shows $ValidTC\ p\ (Basic\ id)\ a\ q$
⟨*proof*⟩

lemma *BasicRuleTC*:
assumes $p \subseteq \{s.\ f\ s \in q\}$
shows $ValidTC\ p\ (Basic\ f)\ a\ q$
⟨*proof*⟩

lemma *SeqRuleTC*:
assumes $ValidTC\ p\ c1\ a1\ q$
and $ValidTC\ q\ c2\ a2\ r$
shows $ValidTC\ p\ (Seq\ c1\ c2)\ (Aseq\ a1\ a2)\ r$
⟨*proof*⟩

lemma *CondRuleTC*:
assumes $p \subseteq \{s.\ (s \in b \longrightarrow s \in w) \wedge (s \notin b \longrightarrow s \in w')\}$

and $ValidTC\ w\ c1\ a1\ q$
and $ValidTC\ w'\ c2\ a2\ q$
shows $ValidTC\ p\ (Cond\ b\ c1\ c2)\ (Acond\ a1\ a2)\ q$
⟨*proof*⟩

lemma *WhileRuleTC*:
assumes $p \subseteq i$
and $\bigwedge n{::}nat\ .\ ValidTC\ (i \cap b \cap \{s\ .\ v\ s = n\})\ c\ (A\ n)\ (i \cap \{s\ .\ v\ s < n\})$
and $i \cap uminus\ b \subseteq q$
shows $ValidTC\ p\ (While\ b\ c)\ (Awhile\ i\ v\ (\lambda n.\ A\ n))\ q$
⟨*proof*⟩

3.1.1 Concrete syntax

⟨*ML*⟩

3.1.2 Proof methods: VCG

declare *BasicRule* [*Hoare-Tac.BasicRule*]
and *SkipRule* [*Hoare-Tac.SkipRule*]
and *SeqRule* [*Hoare-Tac.SeqRule*]
and *CondRule* [*Hoare-Tac.CondRule*]
and *WhileRule* [*Hoare-Tac.WhileRule*]

declare *BasicRuleTC* [*Hoare-Tac.BasicRuleTC*]
and *SkipRuleTC* [*Hoare-Tac.SkipRuleTC*]
and *SeqRuleTC* [*Hoare-Tac.SeqRuleTC*]
and *CondRuleTC* [*Hoare-Tac.CondRuleTC*]
and *WhileRuleTC* [*Hoare-Tac.WhileRuleTC*]

⟨*ML*⟩

end

4 More arithmetic

theory *Arith2*
imports *Main*
begin

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

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

primrec $fac :: nat \Rightarrow nat$
where
$fac\ 0 = Suc\ 0$

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

4.1 cd

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

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

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

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

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

4.2 gcd

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

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

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

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

4.3 pow

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

end

5 Various examples

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

5.1 Arithmetic

5.1.1 Multiplication by successive addition

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

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

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

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

5.1.2 Euclid's algorithm for GCD

lemma *Euclid-GCD*: *VARS a b*
 $\{0<A\ \&\ 0<B\}$
 $a := A;\ b := B;$
 WHILE $a \neq b$

INV {*0*<*a* & *0*<*b* & *gcd A B* = *gcd a b*}
DO IF a<*b THEN b* := *b*−*a ELSE a* := *a*−*b FI OD*
{*a* = *gcd A B*}
⟨*proof*⟩

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

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

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

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

lemma *gcd-scm*: *VARS a b x y*
{*0*<*A* & *0*<*B* & *a*=*A* & *b*=*B* & *x*=*B* & *y*=*A*}
WHILE a ~= *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*⟩

5.1.4 Power by iterated squaring and multiplication

lemma *power-by-mult*: *VARS a b c*
{*a*=*A* & *b*=*B*}
c := (*1*::*nat*);
WHILE b ~= *0*
INV {*A*^*B* = *c* ∗ *a*^*b*}
DO WHILE b mod 2 = *0*
    INV {*A*^*B* = *c* ∗ *a*^*b*}
    DO a := *a*∗*a*; *b* := *b div 2 OD*;
    c := *c*∗*a*; *b* := *b* − *1*
OD
{*c* = *A*^*B*}
⟨*proof*⟩

5.1.5 Factorial

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

lemma *factorial-time*: *VARS a b t*
$\{a=A\ \&\ t=0\}$
$b := 1;\ t := t+1;$
$WHILE\ a > 0$
$INV\ \{fac\ A = b * fac\ a\ \&\ a \le A\ \&\ t = 2{*}(A{-}a){+}1\}$
$DO\ b := b{*}a;\ t := t+1;\ a := a - 1;\ t := t+1\ OD$
$\{b = fac\ A\ \&\ t = 2{*}A + 1\}$
$\langle proof\rangle$

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

lemma *factorial2*: *VARS i f*
$\{True\}$
$i := (1{::}nat);\ f := 1;$
$WHILE\ i <= n\ INV\ \{f = fac(i - 1)\ \&\ 1 <= i\ \&\ i <= n{+}1\}$
$DO\ f := f{*}i;\ i := i{+}1\ OD$
$\{f = fac\ n\}$
$\langle proof\rangle$

lemma *factorial2-time*: *VARS i f t*
$\{t=0\}$
$i := (1{::}nat);\ t := t+1;\ f := 1;\ t := t+1;$
$WHILE\ i \le n\ INV\ \{f = fac(i - 1)\ \&\ 1 \le i\ \&\ i \le n{+}1\ \&\ t = 2{*}(i{-}1){+}2\}$
$DO\ f := f{*}i;\ t := t+1;\ i := i+1;\ t := t+1\ OD$
$\{f = fac\ n\ \&\ t = 2{*}n{+}2\}$
$\langle proof\rangle$

5.1.6 Square root

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

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

5.2 Lists

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-reverse-time*: *VARS y x t*
{*x=X* & *t=0*}
y:=[]; *t* := *t+1*;
WHILE x ~= []
INV {*rev*(*x*)@*y* = *rev*(*X*) & *t* = *2*∗(*length y*) + *1*}
DO y := (*hd x* # *y*); *t* := *t+1*; *x* := *tl x*; *t* := *t+1 OD*
{*y=rev*(*X*) & *t* = *2*∗*length X* + *1*}
⟨*proof*⟩

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

lemma *imperative-append-time-no-rev*: *VARS x y t*

$\{x=X\ \&\ y=Y\}$
$x := rev(x);\ t := 0;$
$WHILE\ x \tilde{\ } =[]$
$INV\ \{rev(x)@y = X@Y\ \&\ length\ x \leq length\ X\ \&\ t = 2 * (length\ X - length\ x)\}$
$DO\ y := (hd\ x\ \#\ y);\ t := t+1;$
$\quad x := tl\ x;\ t := t+1$
OD
$\{y = X@Y\ \&\ t = 2 * length\ X\}$
⟨*proof*⟩

5.3 Arrays

5.3.1 Search for a key

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

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

The *partition* procedure for quicksort.

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

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

lemma *lem*: $m - Suc\ 0 < n ==> m < Suc\ n$
⟨*proof*⟩

lemma *Partition*:
$[|\ leq == \lambda A\ i.\ \forall k.\ k<i \longrightarrow A!k \leq pivot;$
$\quad geq == \lambda A\ i.\ \forall k.\ i<k \wedge k<length\ A \longrightarrow pivot \leq A!k\ |] ==>$

$VARS\ A\ u\ l$
$\{0 < length(A{::}('a{::}order)list)\}$
$l := 0;\ u := length\ A - Suc\ 0;$
$WHILE\ l \le u$
$INV\ \{leq\ A\ l \wedge geq\ A\ u \wedge u{<}length\ A \wedge l{\le}length\ A\}$
$DO\ WHILE\ l < length\ A \wedge A!l \le pivot$
$INV\ \{leq\ A\ l\ \&\ geq\ A\ u \wedge u{<}length\ A \wedge l{\le}length\ A\}$
$DO\ l := l{+}1\ OD;$
$WHILE\ 0 < u\ \&\ pivot \le A!u$
$INV\ \{leq\ A\ l\ \&\ geq\ A\ u \wedge u{<}length\ A \wedge l{\le}length\ A\}$
$DO\ u := u - 1\ OD;$
$IF\ l \le u\ THEN\ A := A[l := A!u,\ u := A!l]\ ELSE\ SKIP\ FI$
OD
$\{leq\ A\ u\ \&\ (\forall k.\ u{<}k \wedge k{<}l \ {-}{-}{>}\ A!k = pivot) \wedge geq\ A\ l\}$
— expand and delete abbreviations first
⟨*proof*⟩

end

6 Hoare Logic with an Abort statement for modelling run time errors

theory *Hoare-Logic-Abort*
imports *Hoare-Syntax Hoare-Tac*
begin

type-synonym *'a bexp = 'a set*
type-synonym *'a assn = 'a set*
type-synonym *'a var = 'a ⇒ nat*

datatype *'a com =*
Basic 'a ⇒ 'a
| *Abort*
| *Seq 'a com 'a com*
| *Cond 'a bexp 'a com 'a com*
| *While 'a bexp 'a com*

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* (*Seq c1 c2*) *s s'*
| *Sem* (*Cond b c1 c2*) *None None*

| $s \in b \Longrightarrow Sem\ c1\ (Some\ s)\ s' \Longrightarrow Sem\ (Cond\ b\ c1\ c2)\ (Some\ s)\ s'$
| $s \notin b \Longrightarrow Sem\ c2\ (Some\ s)\ s' \Longrightarrow Sem\ (Cond\ b\ c1\ c2)\ (Some\ s)\ s'$
| $Sem\ (While\ b\ c)\ None\ None$
| $s \notin b \Longrightarrow Sem\ (While\ b\ c)\ (Some\ s)\ (Some\ s)$
| $s \in b \Longrightarrow Sem\ c\ (Some\ s)\ s'' \Longrightarrow Sem\ (While\ b\ c)\ s''\ s' \Longrightarrow$
$Sem\ (While\ b\ c)\ (Some\ s)\ s'$

inductive-cases [*elim*!]:
$Sem\ (Basic\ f)\ s\ s'$ $Sem\ (Seq\ c1\ c2)\ s\ s'$
$Sem\ (Cond\ b\ c1\ c2)\ s\ s'$

lemma *Sem-deterministic*:
assumes $Sem\ c\ s\ s1$
and $Sem\ c\ s\ s2$
shows $s1 = s2$
⟨*proof*⟩

definition $Valid :: 'a\ bexp \Rightarrow 'a\ com \Rightarrow 'a\ anno \Rightarrow 'a\ bexp \Rightarrow bool$
where $Valid\ p\ c\ a\ q \equiv \forall s\ s'.\ Sem\ c\ s\ s' \longrightarrow s \in Some\ `\ p \longrightarrow s' \in Some\ `\ q$

definition $ValidTC :: 'a\ bexp \Rightarrow 'a\ com \Rightarrow 'a\ anno \Rightarrow 'a\ bexp \Rightarrow bool$
where $ValidTC\ p\ c\ a\ q \equiv \forall s\ .\ s \in p \longrightarrow (\exists t\ .\ Sem\ c\ (Some\ s)\ (Some\ t) \wedge t \in q)$

lemma *tc-implies-pc*:
$ValidTC\ p\ c\ a\ q \Longrightarrow Valid\ p\ c\ a\ q$
⟨*proof*⟩

lemma *tc-extract-function*:
$ValidTC\ p\ c\ a\ q \Longrightarrow \exists f\ .\ \forall s\ .\ s \in p \longrightarrow f\ s \in q$
⟨*proof*⟩

The proof rules for partial correctness

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

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

lemma *SeqRule*: $Valid\ P\ c1\ a1\ Q \Longrightarrow Valid\ Q\ c2\ a2\ R \Longrightarrow Valid\ P\ (Seq\ c1\ c2)\ (Aseq\ a1\ a2)\ R$
⟨*proof*⟩

lemma *CondRule*:
$p \subseteq \{s.\ (s \in b \longrightarrow s \in w) \wedge (s \notin b \longrightarrow s \in w')\}$
$\Longrightarrow Valid\ w\ c1\ a1\ q \Longrightarrow Valid\ w'\ c2\ a2\ q \Longrightarrow Valid\ p\ (Cond\ b\ c1\ c2)\ (Acond\ a1\ a2)\ q$
⟨*proof*⟩

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

lemma *WhileRule*:
$p \subseteq i \Longrightarrow Valid\ (i \cap b)\ c\ (A\ 0)\ i \Longrightarrow i \cap (-b) \subseteq q \Longrightarrow Valid\ p\ (While\ b\ c)$
$(Awhile\ i\ v\ A)\ q$
⟨*proof*⟩

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

The proof rules for total correctness

lemma *SkipRuleTC*:
assumes $p \subseteq q$
shows *ValidTC p* (*Basic id*) *a q*
⟨*proof*⟩

lemma *BasicRuleTC*:
assumes $p \subseteq \{s.\ f\ s \in q\}$
shows *ValidTC p* (*Basic f*) *a q*
⟨*proof*⟩

lemma *SeqRuleTC*:
assumes *ValidTC p c1 a1 q*
and *ValidTC q c2 a2 r*
shows *ValidTC p* (*Seq c1 c2*) (*Aseq a1 a2*) *r*
⟨*proof*⟩

lemma *CondRuleTC*:
assumes $p \subseteq \{s.\ (s \in b \longrightarrow s \in w) \land (s \notin b \longrightarrow s \in w')\}$
and *ValidTC w c1 a1 q*
and *ValidTC w' c2 a2 q*
shows *ValidTC p* (*Cond b c1 c2*) (*Acons a1 a2*) *q*
⟨*proof*⟩

lemma *WhileRuleTC*:
assumes $p \subseteq i$
and $\bigwedge n{::}nat\ .\ ValidTC\ (i \cap b \cap \{s\ .\ v\ s = n\})\ c\ (A\ n)\ (i \cap \{s\ .\ v\ s < n\})$
and $i \cap uminus\ b \subseteq q$
shows *ValidTC p* (*While b c*) (*Awhile i v A*) *q*
⟨*proof*⟩

6.1 Concrete syntax

⟨*ML*⟩
syntax
-guarded-com :: *bool* ⇒ *'a com* ⇒ *'a com* ((*2- →/ -*) *71*)

$\textit{-array-update} :: 'a\ list \Rightarrow nat \Rightarrow 'a \Rightarrow 'a\ com\ \ ((2\text{-}[\text{-}] :=/\ \text{-})\ [70,\ 65]\ 61)$
translations
$P \rightarrow c \rightleftharpoons IF\ P\ THEN\ c\ ELSE\ CONST\ Abort\ FI$
$a[i] := v \rightharpoonup (i < CONST\ length\ a) \rightarrow (a := CONST\ list\text{-}update\ a\ i\ v)$
— reverse translation not possible because of duplicate a

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

6.2 Proof methods: VCG

declare *BasicRule* [*Hoare-Tac.BasicRule*]
and *SkipRule* [*Hoare-Tac.SkipRule*]
and *AbortRule* [*Hoare-Tac.AbortRule*]
and *SeqRule* [*Hoare-Tac.SeqRule*]
and *CondRule* [*Hoare-Tac.CondRule*]
and *WhileRule* [*Hoare-Tac.WhileRule*]

declare *BasicRuleTC* [*Hoare-Tac.BasicRuleTC*]
and *SkipRuleTC* [*Hoare-Tac.SkipRuleTC*]
and *SeqRuleTC* [*Hoare-Tac.SeqRuleTC*]
and *CondRuleTC* [*Hoare-Tac.CondRuleTC*]
and *WhileRuleTC* [*Hoare-Tac.WhileRuleTC*]

$\langle ML \rangle$

end

7 Some small examples for programs that may abort

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

lemma $VARS\ x\ y\ z::nat$
$\{y = z\ \&\ z \neq 0\}\ z \neq 0 \rightarrow x := y\ div\ z\ \{x = 1\}$
$\langle proof \rangle$

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

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*
$\{True\}$
$\langle proof \rangle$

end

8 Examples using Hoare Logic for Total Correctness

theory *ExamplesTC*
imports *Hoare-Logic*
begin

This theory demonstrates a few simple partial- and total-correctness proofs. The first example is taken from HOL/Hoare/Examples.thy written by N. Galm. We have added the invariant $m \leq a$.

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

Here is the total-correctness proof for the same program. It needs the additional invariant $m \leq a$.

lemma *multiply-by-add-tc*: *VARS m s a b*
$[a=A \wedge b=B]$
$m := 0;\ s := 0;$
WHILE $m \neq a$
INV $\{s=m*b \wedge a=A \wedge b=B \wedge m \leq a\}$
VAR $\{a-m\}$
DO $s := s+b;\ m := m+(1::nat)$ *OD*
$[s = A*B]$
$\langle proof \rangle$

Next, we prove partial correctness of a program that computes powers.

lemma *power*: *VARS* $(p::int)\ i$
$\{\ True\ \}$
$p := 1;$
$i := 0;$
WHILE $i < n$
  INV $\{\ p = x\hat{}i \wedge i \leq n\ \}$
   DO $p := p * x;$
     $i := i + 1$
   OD

$\{ \ p = x \hat{\ } n \ \}$
$\langle proof \rangle$

Here is its total-correctness proof.

lemma *power-tc*: *VARS* (*p*::*int*) *i*
$[\ True\ ]$
$p := 1;$
$i := 0;$
$WHILE\ i < n$
$INV\ \{\ p = x \hat{\ } i \wedge i \leq n\ \}$
$VAR\ \{\ n - i\ \}$
$DO\ p := p * x;$
$i := i + 1$
OD
$[\ p = x \hat{\ } n\ ]$
$\langle proof \rangle$

The last example is again taken from HOL/Hoare/Examples.thy. We have modified it to integers so it requires precondition $0 \leq x$.

lemma *sqrt-tc*: *VARS r*
$[0 \leq (x{::}int)]$
$r := 0;$
$WHILE\ (r+1)*(r+1) <= x$
$INV\ \{r*r \leq x\}$
$VAR\ \{\ nat\ (x-r)\}$
$DO\ r := r+1\ OD$
$[r*r \leq x \wedge x < (r+1)*(r+1)]$
$\langle proof \rangle$

A total-correctness proof allows us to extract a function for further use. For every input satisfying the precondition the function returns an output satisfying the postcondition.

lemma *sqrt-exists*:
$0 \leq (x{::}int) \Longrightarrow \exists\, r'\ .\ r'*r' \leq x \wedge x < (r'+1)*(r'+1)$
$\langle proof \rangle$

definition *sqrt* $(x{::}int) \equiv (SOME\ r'\ .\ r'*r' \leq x \wedge x < (r'+1)*(r'+1))$

lemma *sqrt-function*:
assumes $0 \leq (x{::}int)$
and $r' = sqrt\ x$
shows $r'*r' \leq x \wedge x < (r'+1)*(r'+1)$
$\langle proof \rangle$

Nested loops!

lemma *VARS* (*i*::*nat*) *j*
$[\ True\ ]$
$WHILE\ 0 < i$
$INV\ \{\ True\ \}$

```
    VAR { z = i }
     DO i := i − 1; j := i;
        WHILE 0 < j
        INV { z = i+1 }
        VAR { j }
        DO j := j − 1 OD
     OD
  [ i ≤ 0 ]
  ⟨proof⟩
```

end

9 Alternative pointers

```
theory Pointers0
  imports Hoare-Logic
begin
```

9.1 References

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

9.2 Field access and update

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

An example due to Suzuki:

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

9.3 The heap

9.3.1 Paths in the heap

```
primrec Path :: ('a::ref ⇒ 'a) ⇒ 'a ⇒ 'a list ⇒ 'a ⇒ bool
```

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

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

lemma [*simp*]: $a \neq Null \Longrightarrow Path\ h\ a\ as\ z =$
 $(as = [] \wedge z = a \ \vee\ (\exists\, bs.\ as = a\#bs \wedge Path\ h\ (h\ a)\ bs\ z))$
⟨*proof*⟩

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

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

9.3.2 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*]: $List\ h\ x\ [] = (x = Null)$
⟨*proof*⟩

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

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

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

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

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

9.4 Verifications

9.4.1 List reversal

A short but unreadable proof:

lemma *VARS tl p q r*

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

A longer readable version:

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

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

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

9.4.2 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 $VARS\ tl\ p$
$\{List\ tl\ p\ Ps \wedge X \in set\ Ps\}$
$WHILE\ p \neq Null \wedge p \neq X$
$INV\ \{p \neq Null \wedge (\exists ps.\ List\ tl\ p\ ps \wedge X \in set\ ps)\}$
$DO\ p := p\hat{}.tl\ OD$
$\{p = X\}$
$\langle proof \rangle$

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

lemma $VARS\ tl\ p$
$\{Path\ tl\ p\ Ps\ X\}$

$WHILE\ p \neq Null \wedge p \neq X$
$INV\ \{\exists ps.\ Path\ tl\ p\ ps\ X\}$
$DO\ p := p\hat{}.tl\ OD$
$\{p = X\}$
$\langle proof \rangle$

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

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

9.4.3 Merging two lists

This is still a bit rough, especially the proof.

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

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

declare $disj\text{-}not1[simp\ del]\ imp\text{-}disjL[simp\ del]\ imp\text{-}disjCL[simp]$

lemma $VARS\ hd\ tl\ p\ q\ r\ s$
$\{List\ tl\ p\ Ps \wedge List\ tl\ q\ Qs \wedge set\ Ps \cap set\ Qs = \{\} \wedge$
$(p \neq Null \vee q \neq Null)\}$
$IF\ if\ q = Null\ then\ True\ else\ p\ {\sim}{=}\ Null\ \&\ p\hat{}.hd \leq q\hat{}.hd$
$THEN\ r := p;\ p := p\hat{}.tl\ ELSE\ r := q;\ q := q\hat{}.tl\ FI;$
$s := r;$
$WHILE\ p \neq Null \vee q \neq Null$
$INV\ \{\exists rs\ ps\ qs.\ Path\ tl\ r\ rs\ s \wedge List\ tl\ p\ ps \wedge List\ tl\ q\ qs \wedge$
$distinct(s\ \#\ ps\ @\ qs\ @\ rs) \wedge s \neq Null \wedge$
$merge(Ps,Qs,\lambda x\ y.\ hd\ x \leq hd\ y) =$
$rs\ @\ s\ \#\ merge(ps,qs,\lambda x\ y.\ hd\ x \leq hd\ y) \wedge$
$(tl\ s = p \vee tl\ s = q)\}$
$DO\ IF\ if\ q = Null\ then\ True\ else\ p \neq Null \wedge p\hat{}.hd \leq q\hat{}.hd$
$THEN\ s\hat{}.tl := p;\ p := p\hat{}.tl\ ELSE\ s\hat{}.tl := q;\ q := q\hat{}.tl\ FI;$
$s := s\hat{}.tl$
OD
$\{List\ tl\ r\ (merge(Ps,Qs,\lambda x\ y.\ hd\ x \leq hd\ y))\}$
$\langle proof \rangle$

9.4.4 Storage allocation

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

lemma *new-notin*:
$[\![$ ~*finite*(*UNIV*::($'a$::*ref*)*set*); *finite*(A::$'a$ *set*); $B \subseteq A$ $]\!] \Longrightarrow$
  new (A) $\notin B$ & *new* $A \neq$ *Null*
⟨*proof*⟩

lemma ~*finite*(*UNIV*::($'a$::*ref*)*set*) $\Longrightarrow$
  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*}
⟨*proof*⟩

end

10 Pointers, heaps and heap abstractions

See the paper by Mehta and Nipkow.

theory *Heap*
  imports *Main*
begin

10.1 References

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

lemma *not-Null-eq* [*iff*]: ($x \neq$ *Null*) = ($\exists$ y. x = *Ref* y)
  ⟨*proof*⟩

lemma *not-Ref-eq* [*iff*]: ($\forall$ y. $x \neq$ *Ref* y) = (x = *Null*)
  ⟨*proof*⟩

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

10.2 The heap

10.2.1 Paths in the heap

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

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

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

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

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

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

10.2.2 Non-repeating paths

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

The term $distPath\ h\ x\ as\ y$ expresses the fact that a non-repeating path as connects location x to location y by means of the h field. In the case where $x = y$, and there is a cycle from x to itself, as can be both [] and the non-repeating list of nodes in the cycle.

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

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

10.2.3 Lists on the heap

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

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

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

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

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

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

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

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

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

10.2.4 Functional abstraction

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

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

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

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

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

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

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

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

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

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

end

11 Heap syntax

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

11.1 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* &

$distinct[w0,x0,y0,z0]$}
$w\hat{}.v := (1::int)$; $w\hat{}.n := x$;
$x\hat{}.v := 2$; $x\hat{}.n := y$;
$y\hat{}.v := 3$; $y\hat{}.n := z$;
$z\hat{}.v := 4$; $x\hat{}.n := z$
{$w\hat{}.n\hat{}.n\hat{}.v = 4$}
⟨*proof*⟩

end

12 Examples of verifications of pointer programs

theory *Pointer-Examples*
imports *HeapSyntax*
begin

axiomatization where *unproven*: *PROP A*

12.1 Verifications

12.1.1 List reversal

A short but unreadable proof:

lemma *VARS tl p q r*
{$List\ tl\ p\ Ps \wedge List\ tl\ q\ Qs \wedge set\ Ps \cap set\ Qs = \{\}$}
$WHILE\ p \neq Null$
INV {$\exists ps\ qs.\ List\ tl\ p\ ps \wedge List\ tl\ q\ qs \wedge set\ ps \cap set\ qs = \{\} \wedge$
$rev\ ps\ @\ qs = rev\ Ps\ @\ Qs$}
$DO\ r := p;\ p := p\hat{}.tl;\ r\hat{}.tl := q;\ q := r\ OD$
{$List\ tl\ q\ (rev\ Ps\ @\ Qs)$}
⟨*proof*⟩

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

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

A longer readable version:

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

$rev\ ps\ @\ qs = rev\ Ps\ @\ Qs\}$
$DO\ r := p;\ p := p\hat{}.tl;\ r\hat{}.tl := q;\ q := r\ OD$
$\{List\ tl\ q\ (rev\ Ps\ @\ Qs)\}$
$\langle proof \rangle$

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

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

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

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

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

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

lemma $VARS\ tl\ p$
$\{(p,X) \in \{(Ref\ x,tl\ x)\ |x.\ True\}^*\}$
$WHILE\ p \neq Null \wedge p \neq X$
$INV\ \{(p,X) \in \{(Ref\ x,tl\ x)\ |x.\ True\}^*\}$

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

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

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

12.1.3 Splicing two lists

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

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

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

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

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
$\{List\ next\ r\ (merge(Ps,Qs,\lambda x\ y.\ elem\ x \leq elem\ y))\}$
$\langle proof \rangle$

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
$ispath :: ('a \Rightarrow 'a\ ref) \Rightarrow 'a\ ref \Rightarrow 'a\ ref \Rightarrow bool$ **and**
$path :: ('a \Rightarrow 'a\ ref) \Rightarrow 'a\ ref \Rightarrow 'a\ ref \Rightarrow 'a\ list$

First some basic lemmas:

lemma [*simp*]: $ispath\ f\ p\ p$
⟨*proof*⟩
lemma [*simp*]: $path\ f\ p\ p = []$
⟨*proof*⟩
lemma [*simp*]: $ispath\ f\ p\ q \Longrightarrow a \notin set(path\ f\ p\ q) \Longrightarrow ispath\ (f(a := r))\ p\ q$
⟨*proof*⟩
lemma [*simp*]: $ispath\ f\ p\ q \Longrightarrow a \notin set(path\ f\ p\ q) \Longrightarrow$
$path\ (f(a := r))\ p\ q = path\ f\ p\ q$
⟨*proof*⟩

Some more specific lemmas needed by the example:

lemma [*simp*]: $ispath\ (f(a := q))\ p\ (Ref\ a) \Longrightarrow ispath\ (f(a := q))\ p\ q$
⟨*proof*⟩
lemma [*simp*]: $ispath\ (f(a := q))\ p\ (Ref\ a) \Longrightarrow$
$path\ (f(a := q))\ p\ q = path\ (f(a := q))\ p\ (Ref\ a)\ @\ [a]$
⟨*proof*⟩
lemma [*simp*]: $ispath\ f\ p\ (Ref\ a) \Longrightarrow f\ a = Ref\ b \Longrightarrow$
$b \notin set\ (path\ f\ p\ (Ref\ a))$
⟨*proof*⟩
lemma [*simp*]: $ispath\ f\ p\ (Ref\ a) \Longrightarrow f\ a = Null \Longrightarrow islist\ f\ p$
⟨*proof*⟩
lemma [*simp*]: $ispath\ f\ p\ (Ref\ a) \Longrightarrow f\ a = Null \Longrightarrow list\ f\ p = path\ f\ p\ (Ref\ a)\ @\ [a]$
⟨*proof*⟩

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

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

$(tl\ a = p \vee tl\ a = q)\}$
$DO\ IF\ cor\ (q = Null)\ (cand\ (p \neq Null)\ (p\hat{}.hd \leq q\hat{}.hd))$
$THEN\ s\hat{}.tl := p;\ p := p\hat{}.tl\ ELSE\ s\hat{}.tl := q;\ q := q\hat{}.tl\ FI;$
$s := s\hat{}.tl$
OD
$\{islist\ tl\ r\ \&\ list\ tl\ r = (merge(Ps,Qs,\lambda x\ y.\ hd\ x \leq hd\ y))\}$
$\langle proof\rangle$

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

12.1.5 Cyclic list reversal

We consider two algorithms for the reversal of circular lists.

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

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

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

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

lemma *circular-list-rev-II*:
$VARS\ next\ p\ q\ tmp$
$\{p = Ref\ r \wedge distPath\ next\ p\ (r\#Ps)\ p\}$
$q{:=}Null;$
$WHILE\ p \neq Null$
INV
$\{\ ((q = Null) \longrightarrow (\exists ps.\ distPath\ next\ p\ (ps)\ (Ref\ r) \wedge ps = r\#Ps)) \wedge$

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

12.1.6 Storage allocation

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

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

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

end

13 Heap syntax (abort)

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

13.1 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(CONST addr r := v)
p^.f := e => (p ≠ CONST Null) → (f := -refupdate f p e)
p^.f => f(CONST addr p)

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

An example due to Suzuki:

lemma *VARS v n*
{w = Ref w0 & x = Ref x0 & y = Ref y0 & z = Ref z0 &
 distinct[w0,x0,y0,z0]}
w^.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

14 Examples of verifications of pointer programs

theory *Pointer-ExamplesAbort*
imports *HeapSyntaxAbort*
begin

14.1 Verifications

14.1.1 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

15 Proof of the Schorr-Waite graph marking algorithm

theory *SchorrWaite*
imports *HeapSyntax*
begin

15.1 Machinery for the Schorr-Waite proof

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

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

definition
$addrs :: 'a\ ref\ set \Rightarrow 'a\ set$
where $addrs\ P = \{a.\ Ref\ a \in P\}$

definition
$reachable :: ('a \times 'a)\ set \Rightarrow 'a\ ref\ set \Rightarrow 'a\ set$
where $reachable\ r\ P = (r^*\ ``\ addrs\ P)$

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

Rewrite rules for relations induced by a mapping

lemma *self-reachable*: $b \in B \Longrightarrow b \in R^*\ ``\ B$
⟨*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*: $(reachable\ mS\ aS \cup reachable\ mS\ bS) = reachable\ mS\ (aS \cup bS)$
$\langle proof \rangle$

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

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

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

definition
— Restriction of a relation
$restr :: ('a \times 'a)\ set \Rightarrow ('a \Rightarrow bool) \Rightarrow ('a \times 'a)\ set$ $\quad ((\text{-/ } \mid \text{ -})\ [50,\ 51]\ 50)$
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*]:
$(\forall x.\ \neg\ m\ x) \Longrightarrow (R\ |m) = R$
$\langle proof \rangle$

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

lemma [*simp*]: $[\![m\ l]\!] \Longrightarrow (l,x) \in (R \mid m)^* = (l=x)$
$\langle proof \rangle$

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

$\langle proof \rangle$

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

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

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

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*⟩

primrec
— Recursive definition of what is means for a the graph/stack structure to be reconstructible
$stkOk :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a\ ref) \Rightarrow ('a \Rightarrow 'a\ ref) \Rightarrow ('a \Rightarrow 'a\ ref) \Rightarrow ('a \Rightarrow 'a\ ref) \Rightarrow 'a\ ref \Rightarrow 'a\ list \Rightarrow\ bool$
where
stkOk-nil: $stkOk\ c\ l\ r\ iL\ iR\ t\ [] = True$
| *stkOk-cons*:
$stkOk\ c\ l\ r\ iL\ iR\ t\ (p\#stk) = (stkOk\ c\ l\ r\ iL\ iR\ (Ref\ p)\ (stk) \wedge$
$iL\ p = (if\ c\ p\ then\ l\ p\ else\ t) \wedge$
$iR\ p = (if\ c\ p\ then\ t\ else\ r\ p))$

Rewrite rules for stkOk

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

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

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

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

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

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

⟨*proof*⟩

15.2 The Schorr-Waite algorithm

theorem *SchorrWaiteAlgorithm*:
VARS c m l r t p q root
$\{R = reachable\ (relS\ \{l, r\})\ \{root\} \wedge (\forall x.\ \neg\ m\ x) \wedge iR = r \wedge iL = l\}$
$t := root;\ p := Null;$
$WHILE\ p \neq Null \vee t \neq Null \wedge \neg\ t\hat{}.m$
$INV\ \{\exists stack.$
$\quad List\ (S\ c\ l\ r)\ p\ stack \wedge$ — *i1*
$\quad (\forall x \in set\ stack.\ m\ x) \wedge$ — *i2*
$\quad R = reachable\ (relS\{l, r\})\ \{t,p\} \wedge$ — *i3*
$\quad (\forall x.\ x \in R \wedge \neg m\ x \longrightarrow$ — *i4*
$\qquad x \in reachable\ (relS\{l,r\}|m)\ (\{t\} \cup set(map\ r\ stack))) \wedge$
$\quad (\forall x.\ m\ x \longrightarrow x \in R) \wedge$ — *i5*
$\quad (\forall x.\ x \notin set\ stack \longrightarrow r\ x = iR\ x \wedge l\ x = iL\ x) \wedge$ — *i6*
$\quad (stkOk\ c\ l\ r\ iL\ iR\ t\ stack)$ — *i7*$\}$
$DO\ IF\ t = Null \vee t\hat{}.m$
$\quad THEN\ IF\ p\hat{}.c$
$\qquad THEN\ q := t;\ t := p;\ p := p\hat{}.r;\ t\hat{}.r := q$ — *pop*
$\qquad ELSE\ q := t;\ t := p\hat{}.r;\ p\hat{}.r := p\hat{}.l;$ — *swing*
$\qquad\quad p\hat{}.l := q;\ p\hat{}.c := True \quad FI$
$\quad ELSE\ q := p;\ p := t;\ t := t\hat{}.l;\ p\hat{}.l := q;$ — *push*
$\qquad p\hat{}.m := True;\ p\hat{}.c := False \quad FI \quad OD$
$\{(\forall x.\ (x \in R) = m\ x) \wedge (r = iR \wedge l = iL)\ \}$
(**is** *Valid*
$\quad \{(c, m, l, r, t, p, q, root).\ ?Pre\ c\ m\ l\ r\ root\}$
$\quad (Seq\ \text{-}\ (Seq\ \text{-}\ (While\ \{(c, m, l, r, t, p, q, root).\ ?whileB\ m\ t\ p\}\ \text{-})))$
$\quad (Aseq\ \text{-}\ (Aseq\ \text{-}\ (Awhile\ \{(c, m, l, r, t, p, q, root).\ ?inv\ c\ m\ l\ r\ t\ p\}\ \text{-}\ \text{-})))\ \text{-})$
⟨*proof*⟩

end

16 Heap abstractions for Separation Logic

(at the moment only Path and List)

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.

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

16.2 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*]: $List\ h\ x\ [] = (x = 0)$
⟨*proof*⟩

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

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

lemma *List-non-null*: $a{\neq}0 \Longrightarrow$
$List\ h\ a\ as = (\exists\, b\ bs.\ as = a\#bs \wedge h\ a = Some\ b \wedge List\ h\ b\ bs)$
⟨*proof*⟩

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

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

lemma *List-unique1*: $List\ h\ p\ as \Longrightarrow \exists !as.\ List\ h\ p\ as$
⟨*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*]: $List\ h\ b\ as \Longrightarrow h\ a = Some\ b \Longrightarrow a \notin set\ as$
⟨*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*]:
$\bigwedge p.\ [\![\ List\ h2\ p\ ps;\ dom\ h1 \cap dom\ h2 = \{\}]\!] \Longrightarrow List\ (h1{+}{+}h2)\ p\ ps$
⟨*proof*⟩

end

17 Separation logic

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 $is\text{-}empty\ h \longleftrightarrow h = Map.empty$

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
$\{x \neq z\}$
⟨*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*
$\{[x \mapsto y] ** [z \mapsto w]\}$
SKIP
$\{x \neq z\}$
⟨*proof*⟩

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

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

⟨*ML*⟩

Now the intermediate proof states are also readable:

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

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

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

lemma *star-comm*: $P ** Q = Q ** P$

$\langle proof \rangle$

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

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

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

end

References

[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.