Hoare Logic

Norbert Galm
Walter Guttmann
Farhad Mehta
Tobias Nipkow
Leonor Prensa Nieto

October 24, 2026

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

1 Concrete syntax for Hoare logic, with translations for variables **5**

2 Hoare logic VCG tactic **6**

3 Hoare logic **7**
3.1 Sugared semantic embedding of Hoare logic 7
3.1.1 Concrete syntax 10
3.1.2 Proof methods: VCG 10

4 More arithmetic **11**
4.1 cd 11
4.2 gcd 12
4.3 pow 12

5 Various examples **13**
5.1 Arithmetic 13
5.1.1 Multiplication by successive addition 13
5.1.2 Euclid's algorithm for GCD 14

5.1.3 Dijkstra's extension of Euclid's algorithm for simultaneous GCD and SCM . . . 14
5.1.4 Power by iterated squaring and multiplication . . . 15
5.1.5 Factorial . . . 15
5.1.6 Square root . . . 16
5.2 Lists . . . 17
5.3 Arrays . . . 18
5.3.1 Search for a key . . . 18

6 Hoare Logic with an Abort statement for modelling run time errors 19
6.1 Concrete syntax . . . 23
6.2 Proof methods: VCG . . . 23

7 Some small examples for programs that may abort 24

8 Examples using Hoare Logic for Total Correctness 25

9 Alternative pointers 27
9.1 References . . . 27
9.2 Field access and update . . . 27
9.3 The heap . . . 28
9.3.1 Paths in the heap . . . 28
9.3.2 Lists on the heap . . . 28
9.3.3 Functional abstraction . . . 29
9.4 Verifications . . . 30
9.4.1 List reversal . . . 30
9.4.2 Searching in a list . . . 32
9.4.3 Merging two lists . . . 33
9.4.4 Storage allocation . . . 35

10 Pointers, heaps and heap abstractions 35
10.1 References . . . 35
10.2 The heap . . . 36
10.2.1 Paths in the heap . . . 36
10.2.2 Non-repeating paths . . . 36
10.2.3 Lists on the heap . . . 37
10.2.4 Functional abstraction . . . 38

11 Heap syntax 39
11.1 Field access and update . . . 39

12 Examples of verifications of pointer programs **40**
12.1 Verifications 40
12.1.1 List reversal 40
12.1.2 Searching in a list 42
12.1.3 Splicing two lists 43
12.1.4 Merging two lists 44
12.1.5 Cyclic list reversal 47
12.1.6 Storage allocation 49

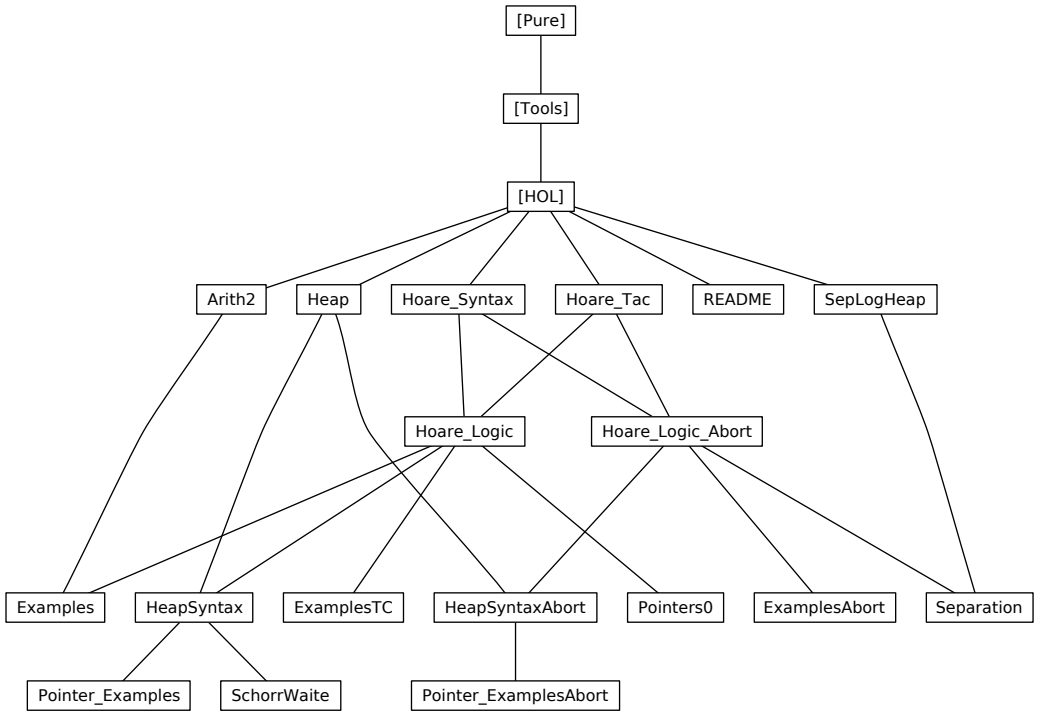
13 Heap syntax (abort) **49**
13.1 Field access and update 49

14 Examples of verifications of pointer programs **50**
14.1 Verifications 50
14.1.1 List reversal 50

15 Proof of the Schorr-Waite graph marking algorithm **51**
15.1 Machinery for the Schorr-Waite proof 51
15.2 The Schorr-Waite algorithm 55

16 Heap abstractions for Separation Logic **63**
16.1 Paths in the heap 63
16.2 Lists on the heap 63

17 Separation logic **65**


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

1 Concrete syntax for Hoare logic, with translations for variables

theory *Hoare-Syntax*
imports *Main*
begin

syntax
-assign :: *idt* ⇒ *'b* ⇒ *'com*
(‹(‹*indent notation=‹infix Hoare assignment››- :=/ -*)› [*70*, *65*] *61*)
-Seq :: *'com* ⇒ *'com* ⇒ *'com*
(‹(‹*notation=‹infix Hoare sequential composition››-;/ -*)› [*61*, *60*] *60*)
-Cond :: *'bexp* ⇒ *'com* ⇒ *'com* ⇒ *'com*
(‹(‹*notation=‹mixfix Hoare if expression›› IF -/ THEN - / ELSE -/ FI*)› [*0*, *0*, *0*] *61*)
-While :: *'bexp* ⇒ *'assn* ⇒ *'var* ⇒ *'com* ⇒ *'com*
(‹(‹*notation=‹mixfix Hoare while expression›› WHILE -/ 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*
(‹(‹*indent=1 notation=‹mixfix Hoare while expression›› WHILE -/ INV* (‹*open-block notation=‹mixfix Hoare invariant››*{-}) *//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*
(‹(‹*open-block notation=‹mixfix Hoare triple›› VARS -// (‹open-block notation=‹mixfix Hoare precondition››*{-}) *// - // (‹open-block notation=‹mixfix Hoare postcondition››*{-}))› [*0*, *0*, *55*, *0*] *50*)
-hoare-vars-tc :: [*idts*, *'assn*, *'com*, *'assn*] ⇒ *bool*
(‹(‹*open-block notation=‹mixfix Hoare triple›› VARS -// (‹open-block notation=‹mixfix Hoare precondition››*[-]) *// - // (‹open-block notation=‹mixfix Hoare postcondition››*[-]))› [*0*, *0*, *55*, *0*] *50*)
syntax (**output**)
-hoare :: [*'assn*, *'com*, *'assn*] ⇒ *bool*
(‹(‹*notation=‹mixfix Hoare triple››(‹open-block notation=‹mixfix Hoare precondition››*{-})*//-//(‹open-block notation=‹mixfix Hoare postcondition››*{-}))› [*0*, *55*, *0*] *50*)

-hoare-tc :: [*'assn*, *'com*, *'assn*] ⇒ *bool*
(‹(‹*notation*=‹*mixfix Hoare triple*››(‹*open-block notation*=‹*mixfix Hoare precondition*››[-])//-//(‹*open-block notation*=‹*mixfix Hoare postcondition*››[-]))› [*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-file ‹*hoare-syntax.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*: $-(Collect\ b) = \{x.\ \neg(b\ x)\}$
by *blast*

ML-file ‹*hoare-tac.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 ⇒ '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 ⇒ 'a sem*
where
Sem (Basic f) s (f s)
| Sem c1 s s'' ⟹ Sem c2 s'' s' ⟹ Sem (Seq c1 c2) s s'
| s ∈ b ⟹ Sem c1 s s' ⟹ Sem (Cond b c1 c2) s s'
| s ∉ b ⟹ Sem c2 s s' ⟹ Sem (Cond b c1 c2) s s'
| s ∉ b ⟹ Sem (While b c) s s
| s ∈ b ⟹ Sem c s s'' ⟹ Sem (While b c) s'' s' ⟹
Sem (While b c) s s'

definition *Valid :: 'a bexp ⇒ 'a com ⇒ 'a anno ⇒ 'a bexp ⇒ bool*
where *Valid p c a q ≡ ∀ s s'. Sem c s s' ⟶ s ∈ p ⟶ s' ∈ q*

definition *ValidTC :: 'a bexp ⇒ 'a com ⇒ 'a anno ⇒ 'a bexp ⇒ bool*
where *ValidTC p c a q ≡ ∀ s. s ∈ p ⟶ (∃ t. Sem c s t ∧ t ∈ 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 −
have *Sem c s s1* ⟹ (∀ *s2*. *Sem c s s2* ⟶ *s1 = s2*)
by (*induct rule*: *Sem.induct*) (*subst Sem.simps*, *blast*)+
thus *?thesis*
using *assms* **by** *simp*
qed

lemma *tc-implies-pc*:
ValidTC p c a q ⟹ *Valid p c a q*
by (*metis Sem-deterministic Valid-def ValidTC-def*)

lemma *tc-extract-function*:
ValidTC p c a q ⟹ ∃*f* . ∀ *s* . *s* ∈ *p* ⟶ *f s* ∈ *q*
by (*metis ValidTC-def*)

lemma *SkipRule*: *p* ⊆ *q* ⟹ *Valid p* (*Basic id*) *a q*
by (*auto simp*:*Valid-def*)

lemma *BasicRule*: *p* ⊆ {*s*. *f s* ∈ *q*} ⟹ *Valid p* (*Basic f*) *a q*
by (*auto simp*:*Valid-def*)

lemma *SeqRule*: *Valid P c1 a1 Q* ⟹ *Valid Q c2 a2 R* ⟹ *Valid P* (*Seq c1 c2*) (*Aseq a1 a2*) *R*
by (*auto simp*:*Valid-def*)

lemma *CondRule*:
p ⊆ {*s*. (*s* ∈ *b* ⟶ *s* ∈ *w*) ∧ (*s* ∉ *b* ⟶ *s* ∈ *w'*)}
⟹ *Valid w c1 a1 q* ⟹ *Valid w' c2 a2 q* ⟹ *Valid p* (*Cond b c1 c2*) (*Acond a1 a2*) *q*
by (*auto simp*:*Valid-def*)

lemma *While-aux*:
assumes *Sem* (*While b c*) *s s'*
shows ∀ *s s'*. *Sem c s s'* ⟶ *s* ∈ *I* ∧ *s* ∈ *b* ⟶ *s'* ∈ *I* ⟹
s ∈ *I* ⟹ *s'* ∈ *I* ∧ *s'* ∉ *b*
using *assms*
by (*induct While b c s s'*) *auto*

lemma *WhileRule*:
p ⊆ *i* ⟹ *Valid* (*i* ∩ *b*) *c* (*A 0*) *i* ⟹ *i* ∩ (−*b*) ⊆ *q* ⟹ *Valid p* (*While b c*) (*Awhile i v A*) *q*
apply (*clarsimp simp*:*Valid-def*)
apply(*drule While-aux*)
apply *assumption*

apply *blast*
apply *blast*
done

lemma *SkipRuleTC*:
assumes $p \subseteq q$
shows *ValidTC p* (*Basic id*) *a q*
by (*metis assms Sem.intros*(*1*) *ValidTC-def id-apply subsetD*)

lemma *BasicRuleTC*:
assumes $p \subseteq \{s.\ f\ s \in q\}$
shows *ValidTC p* (*Basic f*) *a q*
by (*metis assms Ball-Collect Sem.intros*(*1*) *ValidTC-def*)

lemma *SeqRuleTC*:
assumes *ValidTC p c1 a1 q*
and *ValidTC q c2 a2 r*
shows *ValidTC p* (*Seq c1 c2*) (*Aseq a1 a2*) *r*
by (*meson assms Sem.intros*(*2*) *ValidTC-def*)

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 (*unfold ValidTC-def*, *rule allI*)
fix *s*
show $s \in p \longrightarrow (\exists t\ .\ Sem\ (Cond\ b\ c1\ c2)\ s\ t \wedge t \in q)$
apply (*cases* $s \in b$)
apply (*metis* (*mono-tags*, *lifting*) *assms*(*1*,*2*) *Ball-Collect Sem.intros*(*3*) *ValidTC-def*)
by (*metis* (*mono-tags*, *lifting*) *assms*(*1*,*3*) *Ball-Collect Sem.intros*(*4*) *ValidTC-def*)
qed

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 −
have $s \in i \wedge v\ s = n \longrightarrow (\exists t\ .\ Sem\ (While\ b\ c)\ s\ t \wedge t \in q)$ **for** *s n*
proof (*induction n arbitrary*: *s rule*: *less-induct*)
fix *n* :: *nat*
fix *s* :: *′a*
assume *1*: $\bigwedge(m{::}nat)\ s{::}'a\ .\ m < n \Longrightarrow s \in i \wedge v\ s = m \longrightarrow (\exists t\ .\ Sem\ (While\ b\ c)\ s\ t \wedge t \in q)$
show $s \in i \wedge v\ s = n \longrightarrow (\exists t\ .\ Sem\ (While\ b\ c)\ s\ t \wedge t \in q)$
proof (*rule impI*, *cases* $s \in b$)
assume *2*: $s \in b$ **and** $s \in i \wedge v\ s = n$
hence $s \in i \cap b \cap \{s\ .\ v\ s = n\}$

using *assms(1)* **by** *auto*
hence $\exists t\ .\ Sem\ c\ s\ t \wedge t \in i \cap \{s\ .\ v\ s < n\}$
by (*metis assms(2) ValidTC-def*)
from *this* **obtain** *t* **where** *3*: $Sem\ c\ s\ t \wedge t \in i \cap \{s\ .\ v\ s < n\}$
by *auto*
hence $\exists u\ .\ Sem\ (While\ b\ c)\ t\ u \wedge u \in q$
using *1* **by** *auto*
thus $\exists t\ .\ Sem\ (While\ b\ c)\ s\ t \wedge t \in q$
using *2 3 Sem.intros(6)* **by** *force*
next
assume $s \notin b$ **and** $s \in i \wedge v\ s = n$
thus $\exists t\ .\ Sem\ (While\ b\ c)\ s\ t \wedge t \in q$
using *Sem.intros(5) assms(3)* **by** *fastforce*
qed
qed
thus *?thesis*
using *assms(1) ValidTC-def* **by** *force*
qed

3.1.1 Concrete syntax

setup ‹
Hoare-Syntax.setup
{*Basic* = ***const-syntax*** ‹*Basic*›,
Skip = ***const-syntax*** ‹*annskip*›,
Seq = ***const-syntax*** ‹*Seq*›,
Cond = ***const-syntax*** ‹*Cond*›,
While = ***const-syntax*** ‹*While*›,
Valid = ***const-syntax*** ‹*Valid*›,
ValidTC = ***const-syntax*** ‹*ValidTC*›}
›

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*]

method-setup *vcg* = ‹
Scan.succeed (*fn ctxt => SIMPLE-METHOD'* (*Hoare-Tac.hoare-tac ctxt* (*K all-tac*)))›
verification condition generator

method-setup *vcg-simp* = ‹
  Scan.succeed (*fn ctxt* =>
    SIMPLE-METHOD' (*Hoare-Tac.hoare-tac ctxt* (*asm-full-simp-tac ctxt*)))›
  verification condition generator plus simplification

method-setup *vcg-tc* = ‹
  Scan.succeed (*fn ctxt* => *SIMPLE-METHOD'* (*Hoare-Tac.hoare-tc-tac ctxt* (*K all-tac*)))›
  verification condition generator

method-setup *vcg-tc-simp* = ‹
  Scan.succeed (*fn ctxt* =>
    SIMPLE-METHOD' (*Hoare-Tac.hoare-tc-tac ctxt* (*asm-full-simp-tac ctxt*)))›
  verification condition generator plus simplification

end

4 More arithmetic

theory *Arith2*
  imports *Main*
begin

definition *cd* :: [*nat*, *nat*, *nat*] ⇒ *bool*
  where *cd x m n* ⟷ *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*) ⟶ *y*≤*x*))

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

4.1 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*: $n \leq m \Longrightarrow cd\ x\ m\ n = cd\ x\ (m-n)\ n$
apply (*unfold cd-def*)
apply (*fastforce dest*: *dvd-diffD*)
done

lemma *cd-diff-r*: $m \leq n \Longrightarrow cd\ x\ m\ n = cd\ x\ m\ (n-m)$
apply (*unfold cd-def*)
apply (*fastforce dest*: *dvd-diffD*)
done

4.2 gcd

lemma *gcd-nnn*: $0 < n \Longrightarrow n = gcd\ n\ n$
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*: $gcd\ m\ n = gcd\ n\ m$
apply (*simp add*: *gcd-def cd-swap*)
done

lemma *gcd-diff-l*: $n \leq m \Longrightarrow gcd\ m\ n = gcd\ (m-n)\ n$
apply (*unfold gcd-def*)
apply (*subgoal-tac* $n \leq m \Longrightarrow \forall x.\ cd\ x\ m\ n = cd\ x\ (m-n)\ n$)
apply *simp*
apply (*rule allI*)
apply (*erule cd-diff-l*)
done

lemma *gcd-diff-r*: $m \leq n \Longrightarrow gcd\ m\ n = gcd\ m\ (n-m)$
apply (*unfold gcd-def*)
apply (*subgoal-tac* $m \leq n \Longrightarrow \forall x.\ cd\ x\ m\ n = cd\ x\ m\ (n-m)$)
apply *simp*
apply (*rule allI*)
apply (*erule cd-diff-r*)
done

4.3 pow

lemma *sq-pow-div2* [*simp*]:
$m\ mod\ 2 = 0 \Longrightarrow ((n::nat)*n)\hat{}(m\ div\ 2) = n\hat{}m$
apply (*simp add*: *power2-eq-square* [*symmetric*] *power-mult* [*symmetric*] *minus-mod-eq-mult-div* [*symmetric*])
done

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 ∧ b=B}
m := 0; s := 0;
WHILE m≠a
INV {s=m∗b ∧ a=A ∧ b=B}
DO s := s+b; m := m+(1::nat) OD
{s = A∗B}
by *vcg-simp*

lemma *multiply-by-add-time*: *VARS m s a b t*
{a=A ∧ b=B ∧ t=0}
m := 0; t := t+1; s := 0; t := t+1;
WHILE m≠a
INV {s=m∗b ∧ a=A ∧ b=B ∧ t = 2∗m + 2}
DO s := s+b; t := t+1; m := m+(1::nat); t := t+1 OD
{s = A∗B ∧ t = 2∗A + 2}
by *vcg-simp*

lemma *multiply-by-add2*: *VARS M N P :: int*
{m=M ∧ n=N}
IF M < 0 THEN M := −M; N := −N ELSE SKIP FI;
P := 0;
WHILE 0 < M
INV {0 ≤ M ∧ (∃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 (*auto simp add:int-distrib*)
done

lemma *multiply-by-add2-time*: *VARS M N P t :: int*
{m=M ∧ n=N ∧ 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 ≤ M & (∃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*
$\{P = m*n$ & $t \leq 2*abs\ m + 3\}$
apply *vcg-simp*
apply (*auto simp add*:*int-distrib*)
done

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*$\}$
apply *vcg*
— Now prove the verification conditions
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

lemma *Euclid-GCD-time*: *VARS a b t*
$\{0<A$ & $0<B$ & $t=0\}$
$a := A$; $t := t+1$; $b := B$; $t := t+1$;
WHILE $a \neq b$
INV $\{0<a$ & $0<b$ & *gcd A B* = *gcd a b* & $a \leq A$ & $b \leq B$ & $t \leq max\ A\ B - max\ a\ b + 2\}$
DO IF $a<b$ *THEN* $b := b-a$; $t := t+1$ *ELSE* $a := a-b$; $t := t+1$ *FI OD*
$\{a =$ *gcd A B* & $t \leq max\ A\ B + 2\}$
apply *vcg*
— Now prove the verification conditions
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

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)}
apply *vcg*
apply *simp*
apply(*simp add*: *distribs gcd-diff-r linorder-not-less gcd-diff-l*)
apply(*simp add*: *distribs gcd-nnn*)
done

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}
apply *vcg-simp*
apply(*case-tac b*)
apply *simp*
apply *simp*
done

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

lemma *factorial-time*: *VARS a b t*
{a=A & t=0}
b := 1; t := t+1;
WHILE a > 0
INV {fac A = b ∗ fac a & a ≤ A & t = 2∗(A−a)+1}

DO $b := b*a$; $t := t+1$; $a := a - 1$; $t := t+1$ *OD*
$\{b = fac\ A\ \&\ t = 2*A + 1\}$
apply *vcg-simp*
apply(*clarsimp split*: *nat-diff-split*)
done

lemma [*simp*]: $1 \leq i \Longrightarrow fac\ (i - Suc\ 0) * i = fac\ i$
by(*induct i*, *simp-all*)

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*
$\{f = fac\ n\}$
apply *vcg-simp*
apply(*subgoal-tac* $i = Suc\ n$)
apply *simp*
apply *arith*
done

lemma *factorial2-time*: *VARS i f t*
$\{t=0\}$
$i := (1::nat)$; $t := t+1$; $f := 1$; $t := t+1$;
WHILE $i \leq n$ *INV* $\{f = fac(i - 1)\ \&\ 1 \leq i\ \&\ i \leq n+1\ \&\ t = 2*(i-1)+2\}$
DO $f := f*i$; $t := t+1$; $i := i+1$; $t := t+1$ *OD*
$\{f = fac\ n\ \&\ t = 2*n+2\}$
apply *vcg-simp*
apply *auto*
apply(*subgoal-tac* $i = Suc\ n$)
apply *simp*
apply *arith*
done

5.1.6 Square root

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

lemma *sqrt-time*: *VARS r t*
$\{t=0\}$
$r := (0::nat)$; $t := t+1$;

WHILE $(r+1)*(r+1) <= X$
INV $\{r*r \leq X \ \& \ t = r+1\}$
DO $r := r+1$; $t := t+1$ *OD*
$\{r*r <= X \ \& \ X < (r+1)*(r+1) \ \& \ (t-1)*(t-1) \leq X\}$
apply *vcg-simp*
done

— without multiplication
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*
$\{r*r <= X \ \& \ X < (r+1)*(r+1)\}$
apply *vcg-simp*
done

5.2 Lists

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

lemma *imperative-reverse-time*: *VARS y x t*
$\{x=X \ \& \ t=0\}$
$y:=[]$; $t := t+1$;
WHILE $x \sim= []$
INV $\{rev(x)@y = rev(X) \ \& \ t = 2*(length\ y) + 1\}$
DO $y := (hd\ x \ \# \ y)$; $t := t+1$; $x := tl\ x$; $t := t+1$ *OD*
$\{y=rev(X) \ \& \ t = 2*length\ X + 1\}$
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 $x\sim=[]$
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 imperative-append-time-no-rev: VARS x y t
 {x=X & y=Y}
 x := rev(x); t := 0;
 WHILE x~=[]
 INV {rev(x)@y = X@Y & length x ≤ length X & t = 2 * (length X - length x)}
 DO y := (hd x # y); t := t+1;
    x := tl x; t := t+1
 OD
 {y = X@Y & t = 2 * length X}
apply vcg-simp
apply(simp add: neq-Nil-conv)
apply auto
done
```

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 ≠ 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 zero-search-time: VARS A i t
 {t=0}
 i := 0; t := t+1;
 WHILE i < length A ∧ A!i ≠ key
 INV {(∀j. j<i ⟶ A!j ≠ key) ∧ i ≤ length A ∧ t = i+1}
 DO i := i+1; t := t+1 OD
 {(i < length A ⟶ A!i = key) ∧
  (i = length A ⟶ (∀j. j < length A --> A!j ≠ key)) ∧ t ≤ length A + 1}
apply vcg-simp
apply(blast elim!: less-SucE)
done
```

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 *Partition*:
fixes *pivot*
defines $leq \equiv \lambda A\ i.\ \forall k.\ k<i \longrightarrow A!k \leq pivot$
and $geq \equiv \lambda A\ i.\ \forall k.\ i<k \wedge k<length\ A \longrightarrow pivot \leq A!k$
shows
VARS A u l
$\{0 < length(A::('a::order)list)\}$
$l := 0;\ u := length\ A - Suc\ 0;$
WHILE $l \leq u$
INV $\{leq\ A\ l \wedge geq\ A\ u \wedge u<length\ A \wedge l \leq length\ A\}$
DO WHILE $l < length\ A \wedge A!l \leq pivot$
INV $\{leq\ A\ l\ \&\ geq\ A\ u \wedge u<length\ A \wedge l \leq length\ A\}$
DO $l := l+1$ *OD*;
WHILE $0 < u\ \&\ pivot \leq A!u$
INV $\{leq\ A\ l\ \&\ geq\ A\ u\ \wedge u<length\ A \wedge l \leq length\ A\}$
DO $u := u - 1$ *OD*;
IF $l \leq u$ *THEN* $A := A[l := A!u,\ u := A!l]$ *ELSE SKIP FI*
OD
$\{leq\ A\ u\ \&\ (\forall k.\ u<k \wedge k<l \longrightarrow A!k = pivot) \wedge geq\ A\ l\}$
proof −
have *eq*: $m - Suc\ 0 < n \Longrightarrow m < Suc\ n$ **for** *m n*
by *arith*
show *?thesis*
apply (*simp add*: *assms*)
apply *vcg-simp*
apply (*force simp*: *neq-Nil-conv*)
apply (*blast elim*!: *less-SucE intro*: *Suc-leI*)
apply (*blast elim*!: *less-SucE intro*: *less-imp-diff-less dest*: *eq*)
apply (*force simp*: *nth-list-update*)
done
qed

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 ∈ b ⟹ Sem c1 (Some s) s' ⟹ Sem (Cond b c1 c2) (Some s) s'*
| *s ∉ b ⟹ Sem c2 (Some s) s' ⟹ Sem (Cond b c1 c2) (Some s) s'*
| *Sem (While b c) None None*
| *s ∉ b ⟹ Sem (While b c) (Some s) (Some s)*
| *s ∈ b ⟹ Sem c (Some s) s'' ⟹ Sem (While b c) s'' s' ⟹*
   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 −
  have *Sem c s s1 ⟹ (∀ s2. Sem c s s2 ⟶ s1 = s2)*
    by (*induct rule*: *Sem.induct*) (*subst Sem.simps*, *blast*)+
  thus *?thesis*
    using *assms* **by** *simp*
qed

definition *Valid :: 'a bexp ⇒ 'a com ⇒ 'a anno ⇒ 'a bexp ⇒ bool*
  where *Valid p c a q ≡ ∀ s s'. Sem c s s' ⟶ s ∈ Some ‘ p ⟶ s' ∈ Some ‘ q*

definition *ValidTC* :: *'a bexp ⇒ 'a com ⇒ 'a anno ⇒ 'a bexp ⇒ bool*
where *ValidTC p c a q ≡ ∀ s . s ∈ p ⟶ (∃ t . Sem c (Some s) (Some t) ∧ t ∈ q)*

lemma *tc-implies-pc*:
ValidTC p c a q ⟹ Valid p c a q
by (*smt* (*verit*) *Sem-deterministic ValidTC-def Valid-def image-iff*)

lemma *tc-extract-function*:
ValidTC p c a q ⟹ ∃ f . ∀ s . s ∈ p ⟶ f s ∈ q
by (*meson ValidTC-def*)

The proof rules for partial correctness

lemma *SkipRule*: *p ⊆ q ⟹ Valid p* (*Basic id*) *a q*
by (*auto simp*:*Valid-def*)

lemma *BasicRule*: *p ⊆ {s. f s ∈ q} ⟹ Valid p* (*Basic f*) *a q*
by (*auto simp*:*Valid-def*)

lemma *SeqRule*: *Valid P c1 a1 Q ⟹ Valid Q c2 a2 R ⟹ Valid P* (*Seq c1 c2*) (*Aseq a1 a2*) *R*
by (*auto simp*:*Valid-def*)

lemma *CondRule*:
p ⊆ {s. (s ∈ b ⟶ s ∈ w) ∧ (s ∉ b ⟶ s ∈ w')}
⟹ Valid w c1 a1 q ⟹ Valid w' c2 a2 q ⟹ Valid p (*Cond b c1 c2*) (*Acond a1 a2*) *q*
by (*fastforce simp*:*Valid-def image-def*)

lemma *While-aux*:
assumes *Sem* (*While b c*) *s s'*
shows *∀ s s'. Sem c s s' ⟶ s ∈ Some ' (I ∩ b) ⟶ s' ∈ Some ' I ⟹*
s ∈ Some ' I ⟹ s' ∈ Some ' (I ∩ −b)
using *assms*
by (*induct While b c s s'*) *auto*

lemma *WhileRule*:
p ⊆ i ⟹ Valid (*i ∩ b*) *c* (*A 0*) *i ⟹ i ∩* (*−b*) *⊆ q ⟹ Valid p* (*While b c*) (*Awhile i v A*) *q*
apply (*clarsimp simp*:*Valid-def*)
apply(*drule While-aux*)
apply *assumption*
apply *blast*
apply *blast*
done

lemma *AbortRule*: *p ⊆ {s. False} ⟹ Valid p Abort a q*
by(*auto simp*:*Valid-def*)

The proof rules for total correctness

lemma *SkipRuleTC*:
assumes $p \subseteq q$
shows *ValidTC p* (*Basic id*) *a q*
by (*metis Sem.intros*(*2*) *ValidTC-def assms id-def subsetD*)

lemma *BasicRuleTC*:
assumes $p \subseteq \{s.\ f\ s \in q\}$
shows *ValidTC p* (*Basic f*) *a q*
by (*metis Ball-Collect Sem.intros*(*2*) *ValidTC-def assms*)

lemma *SeqRuleTC*:
assumes *ValidTC p c1 a1 q*
and *ValidTC q c2 a2 r*
shows *ValidTC p* (*Seq c1 c2*) (*Aseq a1 a2*) *r*
by (*meson assms Sem.intros*(*4*) *ValidTC-def*)

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*) (*Acons a1 a2*) *q*
proof (*unfold ValidTC-def*, *rule allI*)
fix *s*
show $s \in p \longrightarrow (\exists t\ .\ Sem\ (Cond\ b\ c1\ c2)\ (Some\ s)\ (Some\ t) \wedge t \in q)$
apply (*cases* $s \in b$)
apply (*metis* (*mono-tags*, *lifting*) *Ball-Collect Sem.intros*(*6*) *ValidTC-def assms*(*1*,*2*))
by (*metis* (*mono-tags*, *lifting*) *Ball-Collect Sem.intros*(*7*) *ValidTC-def assms*(*1*,*3*))
qed

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 −
have $s \in i \wedge v\ s = n \longrightarrow (\exists t\ .\ Sem\ (While\ b\ c)\ (Some\ s)\ (Some\ t) \wedge t \in q)$ **for** *s n*
proof (*induction n arbitrary*: *s rule*: *less-induct*)
fix $n :: nat$
fix $s :: 'a$
assume *1*: $\bigwedge(m{::}nat)\ s{::}'a\ .\ m < n \Longrightarrow s \in i \wedge v\ s = m \longrightarrow (\exists t\ .\ Sem\ (While\ b\ c)\ (Some\ s)\ (Some\ t) \wedge t \in q)$
show $s \in i \wedge v\ s = n \longrightarrow (\exists t\ .\ Sem\ (While\ b\ c)\ (Some\ s)\ (Some\ t) \wedge t \in q)$
proof (*rule impI*, *cases* $s \in b$)
assume *2*: $s \in b$ **and** $s \in i \wedge v\ s = n$
hence $s \in i \cap b \cap \{s\ .\ v\ s = n\}$
using *assms*(*1*) **by** *auto*
hence $\exists t\ .\ Sem\ c\ (Some\ s)\ (Some\ t) \wedge t \in i \cap \{s\ .\ v\ s < n\}$
by (*metis assms*(*2*) *ValidTC-def*)

from *this* **obtain** t **where** *3*: *Sem c* (*Some s*) (*Some t*) $\wedge$ $t \in i \cap \{s$. $v\ s < n\}$
by *auto*
hence $\exists u$. *Sem* (*While b c*) (*Some t*) (*Some u*) $\wedge$ $u \in q$
using *1* **by** *auto*
thus $\exists t$. *Sem* (*While b c*) (*Some s*) (*Some t*) $\wedge$ $t \in q$
using *2 3 Sem.intros*(*10*) **by** *force*
next
assume $s \notin b$ **and** $s \in i \wedge v\ s = n$
thus $\exists t$. *Sem* (*While b c*) (*Some s*) (*Some t*) $\wedge$ $t \in q$
using *Sem.intros*(*9*) *assms*(*3*) **by** *fastforce*
qed
qed
thus *?thesis*
using *assms*(*1*) *ValidTC-def* **by** *force*
qed

6.1 Concrete syntax

setup ‹
Hoare-Syntax.setup
{*Basic* = ***const-syntax*** ‹*Basic*›,
Skip = ***const-syntax*** ‹*annskip*›,
Seq = ***const-syntax*** ‹*Seq*›,
Cond = ***const-syntax*** ‹*Cond*›,
While = ***const-syntax*** ‹*While*›,
Valid = ***const-syntax*** ‹*Valid*›,
ValidTC = ***const-syntax*** ‹*ValidTC*›}
›

— Special syntax for guarded statements and guarded array updates:
syntax
-guarded-com :: *bool* $\Rightarrow$ *'a com* $\Rightarrow$ *'a com*
(‹(‹*indent=2 notation=‹mixfix Hoare guarded statement›*›- $\rightarrow$/ -)› *71*)
-array-update :: *'a list* $\Rightarrow$ *nat* $\Rightarrow$ *'a* $\Rightarrow$ *'a com*
(‹(‹*indent=2 notation=‹mixfix Hoare array update›*›-[-] :=/ -)› [*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-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*]

method-setup *vcg* = ‹
Scan.succeed (*fn ctxt =>* *SIMPLE-METHOD'* (*Hoare-Tac.hoare-tac ctxt* (*K all-tac*)))›
verification condition generator

method-setup *vcg-simp* = ‹
Scan.succeed (*fn ctxt =>*
SIMPLE-METHOD' (*Hoare-Tac.hoare-tac ctxt* (*asm-full-simp-tac ctxt*)))›
verification condition generator plus simplification

method-setup *vcg-tc* = ‹
Scan.succeed (*fn ctxt =>* *SIMPLE-METHOD'* (*Hoare-Tac.hoare-tc-tac ctxt* (*K all-tac*)))›
verification condition generator

method-setup *vcg-tc-simp* = ‹
Scan.succeed (*fn ctxt =>*
SIMPLE-METHOD' (*Hoare-Tac.hoare-tc-tac ctxt* (*asm-full-simp-tac ctxt*)))›
verification condition generator plus simplification

end

7 Some small examples for programs that may abort

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

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

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

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*}
by *vcg-simp*

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* ∧ *b*=*B*}
m := *0*; *s* := *0*;
WHILE m≠*a*
INV {*s*=*m*∗*b* ∧ *a*=*A* ∧ *b*=*B* ∧ *m*≤*a*}
DO s := *s*+*b*; *m* := *m*+(*1*::*nat*) *OD*
{*s* = *A*∗*B*}
by *vcg-simp*

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* ∧ *b*=*B*]
m := *0*; *s* := *0*;
WHILE m≠*a*
INV {*s*=*m*∗*b* ∧ *a*=*A* ∧ *b*=*B* ∧ *m*≤*a*}
VAR {*a*−*m*}
DO s := *s*+*b*; *m* := *m*+(*1*::*nat*) *OD*
[*s* = *A*∗*B*]
apply *vcg-tc-simp*
by *auto*

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^i ∧ i ≤ n }
    DO p := p * x;
       i := i + 1
    OD
{ p = x^n }
apply vcg-simp
by auto
```

Here is its total-correctness proof.

```
lemma power-tc: VARS (p::int) i
 [ True ]
 p := 1;
 i := 0;
 WHILE i < n
   INV { p = x^i ∧ i ≤ n }
   VAR { n − i }
    DO p := p * x;
       i := i + 1
    OD
 [ p = x^n ]
 apply vcg-tc
 by auto
```

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 ≤ (x::int)]
 r := 0;
 WHILE (r+1)*(r+1) <= x
 INV {r*r ≤ x}
 VAR { nat (x−r)}
 DO r := r+1 OD
 [r*r ≤ x ∧ x < (r+1)*(r+1)]
 apply vcg-tc-simp
 by (smt (verit) div-pos-pos-trivial mult-less-0-iff nonzero-mult-div-cancel-left)
```

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 ≤ (x::int) ⟹ ∃ r' . r'*r' ≤ x ∧ x < (r'+1)*(r'+1)
 using tc-extract-function sqrt-tc by blast

definition sqrt (x::int) ≡ (SOME r' . r'*r' ≤ x ∧ x < (r'+1)*(r'+1))

lemma sqrt-function:
 assumes 0 ≤ (x::int)
   and r' = sqrt x
 shows r'*r' ≤ x ∧ x < (r'+1)*(r'+1)
```

proof −
let *?P* = $\lambda r'$. $r'*r' \leq x \wedge x < (r'+1)*(r'+1)$
have *?P* (*SOME z* . *?P z*)
by (*metis* (*mono-tags*, *lifting*) *assms*(*1*) *sqrt-exists some-eq-imp*)
thus *?thesis*
using *assms*(*2*) *sqrt-def* **by** *auto*
qed

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
[$i \leq 0$]
apply *vcg-tc*
by *auto*

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*
(‹(‹*indent=2 notation=‹mixfix Hoare field assignment›*›-^.- :=/ -)› [*70*,*1000*,*65*] *61*)
-faccess :: *'a*::*ref* => (*'a*::*ref* ⇒ *'v*) => *'v*
(‹(‹*open-block notation=‹mixfix Hoare field access›*›-^.-)› [*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*}
by *vcg-simp*

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* ≠ *Null* ∧ *x* = *a* ∧ *Path h* (*h a*) *as y*)

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

lemma [*simp*]: *a* ≠ *Null* ⟹ *Path h a as z* =
(*as* = [] ∧ *z* = *a* ∨ (∃ *bs*. *as* = *a*#*bs* ∧ *Path h* (*h a*) *bs z*))
apply(*case-tac as*)
apply *fastforce*
apply *fastforce*
done

lemma [*simp*]: ⋀*x*. *Path f x* (*as*@*bs*) *z* = (∃ *y*. *Path f x as y* ∧ *Path f y bs z*)
by(*induct as*, *simp*+)

lemma [*simp*]: ⋀*x*. *u* ∉ *set as* ⟹ *Path* (*f*(*u* := *v*)) *x as y* = *Path f x as y*
by(*induct as*, *simp*, *simp add*:*eq-sym-conv*)

9.3.2 Lists on the heap

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

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

lemma [*simp*]: *List h x* (*a*#*as*) = (*x* ≠ *Null* ∧ *x* = *a* ∧ *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*: $\bigwedge x\ bs.\ List\ h\ x\ as \Longrightarrow List\ h\ x\ bs \Longrightarrow as = bs$
by(*induct as*, *simp*, *clarsimp*)

lemma *List-unique1*: $List\ h\ p\ as \Longrightarrow \exists!as.\ List\ h\ p\ as$
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*]: $List\ h\ (h\ a)\ as \Longrightarrow a \notin set\ as$
apply (*clarsimp simp add*:*in-set-conv-decomp*)
apply(*frule List-app*[*THEN iffD1*])
apply(*fastforce dest*: *List-unique*)
done

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

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*: $List\ h\ p\ as = (islist\ h\ p \wedge as = list\ h\ p)$
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*]: $a \neq Null \Longrightarrow islist\ h\ a = islist\ h\ (h\ a)$
by(*simp add*:*islist-def*)

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

lemma *list-Ref-conv*[*simp*]:
$[\![\ a \neq Null;\ islist\ h\ (h\ a)\ ]\!] \Longrightarrow list\ h\ a = a\ \#\ list\ h\ (h\ a)$
apply(*insert List-Ref*[*of* - *h*])
apply(*fastforce simp*:*List-conv-islist-list*)
done

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

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

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

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 $\{\exists\, ps\ qs.\ List\ tl\ p\ ps \wedge List\ tl\ q\ qs \wedge set\ ps \cap set\ qs = \{\} \wedge$
$rev\ ps\ @\ qs = rev\ Ps\ @\ Qs\}$
DO $r := p;\ p := p^{\frown}.tl;\ r^{\frown}.tl := q;\ q := r$ *OD*
$\{List\ tl\ q\ (rev\ Ps\ @\ Qs)\}$
apply *vcg-simp*
apply *fastforce*
apply(*fastforce intro*:*notin-List-update*[*THEN iffD2*])
— explicit:
~~*apply clarify*~~
~~*apply(rename-tac ps qs)*~~
~~*apply clarsimp*~~
~~*apply(rename-tac ps')*~~
~~*apply(rule-tac x=# ps' in exI)*~~
~~*apply simp*~~
~~*apply(rule-tac x=# p#qs in exI)*~~
~~*apply simp*~~
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 $r := p;\ p := p^{\frown}.tl;\ r^{\frown}.tl := q;\ q := r$ *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 $List\ (tl(p := q))\ (p^{\frown}.tl)\ ps' \wedge$
$List\ (tl(p := q))\ p\ \ (p\#qs) \wedge$
$set\ ps' \cap set\ (p\#qs) = \{\} \wedge$
$rev\ ps' @ (p\#qs) = rev\ Ps\ @\ Qs$
using *I* **by** *fastforce*
thus $\exists\, ps'\ qs'.\ List\ (tl(p := q))\ (p^{\frown}.tl)\ ps' \wedge$
$List\ (tl(p := q))\ p\ \ qs' \wedge$
$set\ ps' \cap set\ qs' = \{\} \wedge$
$rev\ ps' @ qs' = rev\ Ps\ @\ Qs$ **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

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*}
apply *vcg-simp*
apply *clarsimp*
apply *clarsimp*
done

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*}
apply *vcg-simp*
apply(*case-tac p* $=$ *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 $\neq$ *Null* $\wedge$ *p* $\neq$ *X*
INV {$\exists$ *ps. Path tl p ps X*}
DO p $:=$ *p*$\hat{}$*.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) ∈ {(x,y). y = tl x & x ≠ Null}*}
  WHILE p ≠ Null ∧ p ≠ X
  INV {(p,X) ∈ {(x,y). y = tl x & x ≠ 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
```

9.4.3 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 {∃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

9.4.4 Storage allocation

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

lemma *new-notin*:
⟦ ~*finite*(*UNIV*::(*'a*::*ref*)*set*); *finite*(*A*::*'a set*); *B* ⊆ *A* ⟧ ⟹
  new (*A*) ∉ *B* & *new A* ≠ *Null*
apply(*unfold new-def*)
apply(*rule someI2-ex*)
 apply (*fast dest*:*ex-new-if-finite*[*of insert Null A*])
apply (*fast*)
done

lemma ~*finite*(*UNIV*::(*'a*::*ref*)*set*) ⟹
  VARS xs elem next alloc p q
  {*Xs* = *xs* ∧ *p* = (*Null*::*'a*)}
  WHILE xs ≠ []
  INV {*islist next p* ∧ *set*(*list next p*) ⊆ *set alloc* ∧
      map elem (*rev*(*list next p*)) @ *xs* = *Xs*}
  DO q := *new*(*set alloc*); *alloc* := *q*#*alloc*;
    q^.*next* := *p*; *q*^.*elem* := *hd xs*; *xs* := *tl xs*; *p* := *q*
  OD
  {*islist next p* ∧ *map elem* (*rev*(*list next p*)) = *Xs*}
apply *vcg-simp*
 apply (*clarsimp simp*: *subset-insert-iff neq-Nil-conv fun-upd-apply new-notin*)
done

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*]: $(x \neq Null) = (\exists\, y.\ x = Ref\ y)$
by (*induct x*) *auto*

lemma *not-Ref-eq* [*iff*]: $(\forall\, y.\ x \neq Ref\ y) = (x = Null)$
by (*induct x*) *auto*

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

10.2 The heap

10.2.1 Paths in the heap

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

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

lemma [*simp*]: *Path h* (*Ref a*) *as z* =
(*as* = [] ∧ *z* = *Ref a* ∨ (∃ *bs*. *as* = *a*#*bs* ∧ *Path h* (*h a*) *bs z*))
apply(*case-tac as*)
apply *fastforce*
apply *fastforce*
done

lemma [*simp*]: ⋀*x*. *Path f x* (*as*@*bs*) *z* = (∃ *y*. *Path f x as y* ∧ *Path f y bs z*)
by(*induct as*, *simp*+)

lemma *Path-upd*[*simp*]:
⋀*x*. *u* ∉ *set as* ⟹ *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*) ⟹ *Path* (*f*(*a* := *q*)) *p* (*as* @ [*a*]) *q*
by *simp*

10.2.2 Non-repeating paths

definition *distPath* :: (*'a* ⇒ *'a ref*) ⇒ *'a ref* ⇒ *'a list* ⇒ *'a ref* ⇒ *bool*
where *distPath h x as y* ⟷ *Path h x as y* ∧ *distinct as*

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

lemma *neq-dP*: $p \neq q \Longrightarrow Path\ h\ p\ Ps\ q \Longrightarrow distinct\ Ps \Longrightarrow$
$\exists\, a\ Qs.\ p = Ref\ a \wedge Ps = a\#Qs \wedge a \notin set\ Qs$
by (*case-tac Ps*, *auto*)

lemma *neq-dP-disp*: $[\![\ p \neq q;\ distPath\ h\ p\ Ps\ q\ ]\!] \Longrightarrow$
$\exists\, a\ Qs.\ p = Ref\ a \wedge Ps = a\#Qs \wedge a \notin set\ Qs$
apply (*simp only*:*distPath-def*)
by (*case-tac Ps*, *auto*)

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

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

lemma *List-unique1*: $List\ h\ p\ as \Longrightarrow \exists!as.\ List\ h\ p\ as$
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*]: $List\ h\ (h\ a)\ as \Longrightarrow a \notin set\ as$
apply (*clarsimp simp add*:*in-set-conv-decomp*)
apply(*frule List-app*[*THEN iffD1*])
apply(*fastforce dest*: *List-unique*)
done

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

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

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*: $List\ h\ p\ as = (islist\ h\ p \wedge as = list\ h\ p)$
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*]:
$islist\ h\ (h\ a) \Longrightarrow list\ h\ (Ref\ a) = a\ \#\ list\ h\ (h\ a)$
apply(*insert List-Ref*[*of h*])
apply(*fastforce simp*:*List-conv-islist-list*)
done

lemma [*simp*]: *islist h (h a)* $\Longrightarrow$ *a* $\notin$ *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 $\Longrightarrow$ *y* $\notin$ *set*(*list h p*) $\Longrightarrow$ *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 $\Longrightarrow$ *y* $\notin$ *set*(*list h p*) $\Longrightarrow$ *islist* (*h*(*y* := *q*)) *p*
apply(*frule notin-List-update*[*of - - h q p*])
apply(*simp add*:*List-conv-islist-list*)
done

end

11 Heap syntax

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

11.1 Field access and update

syntax
-refupdate :: (*'a* $\Rightarrow$ *'b*) $\Rightarrow$ *'a ref* $\Rightarrow$ *'b* $\Rightarrow$ (*'a* $\Rightarrow$ *'b*)
(‹(‹*open-block notation=‹mixfix Hoare ref update››-/'((- → -)')*)› [*1000*,*0*] *900*)
-fassign :: *'a ref* => *id* => *'v* => *'s com*
(‹(‹*indent=2 notation=‹mixfix Hoare ref assignment››-^.- :=/ -*)› [*70*,*1000*,*65*] *61*)
-faccess :: *'a ref* => (*'a ref* $\Rightarrow$ *'v*) => *'v*
(‹(‹*open-block notation=‹infix Hoare ref access››-^.-*)› [*65*,*1000*] *65*)
translations
f(*r* → *v*) == *f*(*CONST addr r* := *v*)
p^.*f* := *e* => *f* := *f*(*p* → *e*)
p^.*f* => *f*(*CONST addr p*)

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

An example due to Suzuki:

lemma *VARS v n*
{*w* = *Ref w0* & *x* = *Ref x0* & *y* = *Ref y0* & *z* = *Ref z0* &
distinct[*w0*,*x0*,*y0*,*z0*]}
w^.*v* := (*1*::*int*); *w*^.*n* := *x*;
x^.*v* := *2*; *x*^.*n* := *y*;

$y\hat{}.v := 3$; $y\hat{}.n := z$;
$z\hat{}.v := 4$; $x\hat{}.n := z$
$\{w\hat{}.n\hat{}.n\hat{}.v = 4\}$
by *vcg-simp*

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*
$\{List\ tl\ q\ (rev\ Ps\ @\ Qs)\}$
apply *vcg-simp*
apply *fastforce*
apply(*fastforce intro*:*notin-List-update*[*THEN iffD2*])
— explicit:
~~*apply clarify*~~
~~*apply(rename-tac ps' q' qs')*~~
~~*apply clarsimp*~~
~~*apply(rename-tac ps')*~~
~~*apply(fastforce intro:notin-List-update[THEN iffD2])*~~
done

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*
$\{List\ next\ q\ (rev\ Ps\ @\ Qs)\}$
apply *vcg-simp*

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 (∃ *ps qs. List tl p ps* ∧ *List tl q qs* ∧ *set ps* ∩ *set qs* = {} ∧
rev ps @ *qs* = *rev Ps* @ *Qs*) ∧ *p* ≠ *Null*
(**is** (∃ *ps qs. ?I ps qs*) ∧ -)
then obtain *ps qs a* **where** *I*: *?I ps qs* ∧ *p* = *Ref a*
by *fast*
then obtain *ps*′ **where** *ps* = *a* # *ps*′ **by** *fastforce*
hence *List* (*tl*(*p* → *q*)) (*p*^.*tl*) *ps*′ ∧
List (*tl*(*p* → *q*)) *p* (*a*#*qs*) ∧
set ps′ ∩ *set* (*a*#*qs*) = {} ∧
rev ps′ @ (*a*#*qs*) = *rev Ps* @ *Qs*
using *I* **by** *fastforce*
thus ∃ *ps*′ *qs*′. *List* (*tl*(*p* → *q*)) (*p*^.*tl*) *ps*′ ∧
List (*tl*(*p* → *q*)) *p* *qs*′ ∧
set ps′ ∩ *set qs*′ = {} ∧
rev ps′ @ *qs*′ = *rev Ps* @ *Qs* **by** *fast*
next
fix *tl p q r*
assume (∃ *ps qs. List tl p ps* ∧ *List tl q qs* ∧ *set ps* ∩ *set qs* = {} ∧
rev ps @ *qs* = *rev Ps* @ *Qs*) ∧ ¬ *p* ≠ *Null*
thus *List tl q* (*rev Ps* @ *Qs*) **by** *fastforce*
qed

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

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

```
  {islist tl q ∧ list tl q = rev Ps @ Qs}
apply vcg-simp
 apply clarsimp
apply clarsimp
done
```

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 ∧ X ∈ set Ps}
  WHILE p ≠ Null ∧ p ≠ Ref X
  INV {∃ps. List tl p ps ∧ X ∈ set ps}
  DO p := p^.tl OD
  {p = Ref X}
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*
{$p \neq Null \wedge (addr\ p,X) \in \{(x,y).\ tl\ x = Ref\ y\}^*$}
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*
{$p = Ref\ X$}
apply *vcg-simp*
apply *clarsimp*
apply(*erule converse-rtranclE*)
apply *simp*
apply *clarsimp*
apply *clarsimp*
done

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*
{$List\ tl\ p\ (splice\ Ps\ Qs)$}
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

12.1.4 Merging two lists

This is still a bit rough, especially the proof.

definition *cor* :: *bool* ⇒ *bool* ⇒ *bool*
  where *cor P Q* ⟷ (*if P then True else Q*)

definition *cand* :: *bool* ⇒ *bool* ⇒ *bool*
  where *cand P Q* ⟷ (*if P then Q else False*)

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*) = []

Simplifies the proof a little:

lemma [*simp*]: ({} = *insert a A* ∩ *B*) = (*a* ∉ *B* & {} = *A* ∩ *B*)
by *blast*
lemma [*simp*]: ({} = *A* ∩ *insert b B*) = (*b* ∉ *A* & {} = *A* ∩ *B*)
by *blast*
lemma [*simp*]: ({} = *A* ∩ (*B* ∪ *C*)) = ({} = *A* ∩ *B* & {} = *A* ∩ *C*)
by *blast*

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 cor (*q* = *Null*) (*cand* (*p* ≠ *Null*) (*p*^.*hd* ≤ *q*^.*hd*))
 THEN r := *p*; *p* := *p*^.*tl ELSE r* := *q*; *q* := *q*^.*tl FI*;
 s := *r*;
 WHILE p ≠ *Null* ∨ *q* ≠ *Null*
 INV {∃ *rs ps qs a*. *Path tl r rs s* ∧ *List tl p ps* ∧ *List tl q qs* ∧
      distinct(*a* # *ps* @ *qs* @ *rs*) ∧ *s* = *Ref a* ∧
      merge(*Ps*,*Qs*,λ*x y*. *hd x* ≤ *hd y*) =
      rs @ *a* # *merge*(*ps*,*qs*,λ*x y*. *hd x* ≤ *hd y*) ∧
      (*tl a* = *p* ∨ *tl a* = *q*)}
 DO IF cor (*q* = *Null*) (*cand* (*p* ≠ *Null*) (*p*^.*hd* ≤ *q*^.*hd*))
    THEN s^.*tl* := *p*; *p* := *p*^.*tl ELSE s*^.*tl* := *q*; *q* := *q*^.*tl FI*;
    s := *s*^.*tl*
 OD
 {*List tl r* (*merge*(*Ps*,*Qs*,λ*x y*. *hd x* ≤ *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
$ispath :: ('a \Rightarrow 'a\ ref) \Rightarrow 'a\ ref \Rightarrow 'a\ ref \Rightarrow bool$ **and**
$path :: ('a \Rightarrow 'a\ ref) \Rightarrow 'a\ ref \Rightarrow 'a\ ref \Rightarrow 'a\ list$

First some basic lemmas:

lemma [*simp*]: $ispath\ f\ p\ p$
by (*rule unproven*)
lemma [*simp*]: $path\ f\ p\ p = []$
by (*rule unproven*)
lemma [*simp*]: $ispath\ f\ p\ q \Longrightarrow a \notin set(path\ f\ p\ q) \Longrightarrow ispath\ (f(a := r))\ p\ q$
by (*rule unproven*)
lemma [*simp*]: $ispath\ f\ p\ q \Longrightarrow a \notin set(path\ f\ p\ q) \Longrightarrow$
$path\ (f(a := r))\ p\ q = path\ f\ p\ q$
by (*rule unproven*)

Some more specific lemmas needed by the example:

lemma [*simp*]: $ispath\ (f(a := q))\ p\ (Ref\ a) \Longrightarrow ispath\ (f(a := q))\ p\ q$
by (*rule unproven*)
lemma [*simp*]: $ispath\ (f(a := q))\ p\ (Ref\ a) \Longrightarrow$
$path\ (f(a := q))\ p\ q = path\ (f(a := q))\ p\ (Ref\ a)\ @\ [a]$
by (*rule unproven*)
lemma [*simp*]: $ispath\ f\ p\ (Ref\ a) \Longrightarrow f\ a = Ref\ b \Longrightarrow$
$b \notin set\ (path\ f\ p\ (Ref\ a))$
by (*rule unproven*)
lemma [*simp*]: $ispath\ f\ p\ (Ref\ a) \Longrightarrow f\ a = Null \Longrightarrow islist\ f\ p$
by (*rule unproven*)

lemma [*simp*]: *ispath f p* (*Ref a*) $\Longrightarrow$ *f a* = *Null* $\Longrightarrow$ *list f p* = *path f p* (*Ref a*) @ [*a*]
by (*rule unproven*)

lemma [*simp*]: *islist f p* $\Longrightarrow$ *distinct* (*list f p*)
by (*rule unproven*)

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*^.*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 {$\exists$ *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*,λ*x y. hd x* $\leq$ *hd y*) =
rs @ *a* # *merge*(*ps*,*qs*,λ*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
{*islist tl r* & *list tl r* = (*merge*(*Ps*,*Qs*,λ*x y. hd x* $\leq$ *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 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*^.*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*^.*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 ∧ distPath next p (r#Ps) p}
q:=Null;
WHILE p ≠ Null
INV
{ ((q = Null) ⟶ (∃ps. distPath next p (ps) (Ref r) ∧ ps = r#Ps)) ∧
  ((q ≠ Null) ⟶ (∃ps qs. distPath next q (qs) (Ref r) ∧ List next p ps  ∧
                 set ps ∩ set qs = {} ∧ rev qs @ ps = Ps@[r])) ∧
  ¬ (p = Null ∧ q = Null) }
DO tmp := p; p := p^.next; tmp^.next := q; q:=tmp OD
{q = Ref r ∧ distPath next q (r # rev Ps) q}
apply (simp only:distPath-def)
apply vcg-simp
  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
```

12.1.6 Storage allocation

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

lemma *new-notin*:
⟦ ~*finite*(*UNIV*::*'a set*); *finite*(*A*::*'a set*); *B* ⊆ *A* ⟧ ⟹ *new* (*A*) ∉ *B*
apply(*unfold new-def*)
apply(*rule someI2-ex*)
apply (*fast intro*:*ex-new-if-finite*)
apply (*fast*)
done

lemma ~*finite*(*UNIV*::*'a set*) ⟹
VARS xs elem next alloc p q
{*Xs* = *xs* ∧ *p* = (*Null*::*'a ref*)}
WHILE xs ≠ []
INV {*islist next p* ∧ *set*(*list next p*) ⊆ *set alloc* ∧
map elem (*rev*(*list next p*)) @ *xs* = *Xs*}
DO q := *Ref*(*new*(*set alloc*)); *alloc* := (*addr q*)#*alloc*;
q^.*next* := *p*; *q*^.*elem* := *hd xs*; *xs* := *tl xs*; *p* := *q*
OD
{*islist next p* ∧ *map elem* (*rev*(*list next p*)) = *Xs*}
apply *vcg-simp*
apply (*clarsimp simp*: *subset-insert-iff neq-Nil-conv fun-upd-apply new-notin*)
done

end

13 Heap syntax (abort)

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

13.1 Field access and update

Heap update *p*^.*h* := *e* is now guarded against *p* being Null. However, *p* may still be illegal, e.g. uninitialized or dangling. To guard against that, one needs a more detailed model of the heap where allocated and free addresses are distinguished, e.g. by making the heap a map, or by carrying the 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*)

(‹(‹open-block notation=‹mixfix Hoare ref update››-/'((- → -)')›) [1000,0] 900)
-fassign :: 'a ref => id => 'v => 's com
(‹(‹indent=2 notation=‹mixfix Hoare ref assignment››-^.- :=/ -)› [70,1000,65] 61)
-faccess :: 'a ref => ('a ref ⇒ 'v) => 'v
(‹(‹open-block notation=‹infix Hoare ref access››-^.-)› [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}
by *vcg-simp*

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

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* ⇒ '*a ref*) ⇒ ('*a* × '*a*) *set*
where *rel m* = {(*x*,*y*). *m x* = *Ref y*}

definition
relS :: ('*a* ⇒ '*a ref*) *set* ⇒ ('*a* × '*a*) *set*
where *relS M* = (⋃ *m* ∈ *M*. *rel m*)

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

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

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

Rewrite rules for relations induced by a mapping

lemma *self-reachable*: *b* ∈ *B* ⟹ *b* ∈ *R** `` *B*
apply *blast*
done

lemma *oneStep-reachable*: *b* ∈ *R*``*B* ⟹ *b* ∈ *R*\* `` *B*
apply *blast*
done

lemma *still-reachable*: ⟦*B*⊆*Ra** ``*A*; ∀ (*x*,*y*) ∈ *Rb*−*Ra*. *y*∈ (*Ra*\* ``*A*)⟧ ⟹ *Rb** `` *B* ⊆ *Ra*\* `` *A*
apply (*clarsimp simp only*:*Image-iff*)
apply (*erule rtrancl-induct*)
apply *blast*
apply (*subgoal-tac* (*y*, *z*) ∈ *Ra*∪(*Rb*−*Ra*))
apply (*erule UnE*)
apply (*auto intro*:*rtrancl-into-rtrancl*)

apply *blast*
done

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

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

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

lemma *reachable-union*: (*reachable mS aS* $\cup$ *reachable mS bS*) = *reachable mS* (*aS* $\cup$ *bS*)
apply (*simp add*: *reachable-def rel-defs addrs-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*: $(a,b) \notin$ *rel* ($r(q:=t)$) $\Longrightarrow$ $(a,b) \in$ *rel r* $\Longrightarrow$ $a=q$
apply (*rule classical*)
apply (*simp add*:*rel-defs fun-upd-apply*)
done

lemma *rel-upd2*: $(a,b) \notin$ *rel r* $\Longrightarrow$ $(a,b) \in$ *rel* ($r(q:=t)$) $\Longrightarrow$ $a=q$
apply (*rule classical*)
apply (*simp add*:*rel-defs fun-upd-apply*)
done

definition
— Restriction of a relation
restr ::$('a \times 'a)$ *set* $\Rightarrow$ $('a \Rightarrow$ *bool*$)$ $\Rightarrow$ $('a \times 'a)$ *set*
(‹(‹*notation*=‹*mixfix relation restriction*››-/ | -)› [*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*]:
$(\forall x.\ \neg\ m\ x) \Longrightarrow (R\ |m) = R$
by (*auto simp add*:*restr-def*)

lemma *restr-rtrancl*[*simp*]: $[\![m\ l]\!] \Longrightarrow (R \mid m)^* \ `` \{l\} = \{l\}$
by (*auto simp add*:*restr-def elim*:*converse-rtranclE*)

lemma [*simp*]: $[\![m\ l]\!] \Longrightarrow (l,x) \in (R \mid m)^* = (l{=}x)$
by (*auto simp add*:*restr-def elim*:*converse-rtranclE*)

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

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*: $(a,\ b) \notin (r|(m(q := t))) \Longrightarrow (a,b) \in (r|m) \Longrightarrow a = q$
apply (*rule classical*)
apply (*simp add*:*restr-def fun-upd-apply*)
done

definition
— A short form for the stack mapping function for List
$S :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a\ ref) \Rightarrow ('a \Rightarrow 'a\ ref) \Rightarrow ('a \Rightarrow 'a\ ref)$
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*]:
$\forall p.\ a \notin set\ stack \longrightarrow List\ (S\ c\ l\ r)\ p\ stack = List\ (S\ (c(a{:=}x))\ (l(a{:=}y))\ (r(a{:=}z)))\ p\ stack$
apply(*induct-tac stack*)
apply(*simp add*:*fun-upd-apply S-def*)+
done

lemma [*rule-format*,*simp*]:
$\forall p.\ a \notin set\ stack \longrightarrow List\ (S\ c\ l\ (r(a{:=}z)))\ p\ stack = List\ (S\ c\ l\ r)\ p\ stack$
apply(*induct-tac stack*)
apply(*simp add*:*fun-upd-apply S-def*)+
done

lemma [*rule-format*,*simp*]:
$\forall p.\ a \notin set\ stack \longrightarrow List\ (S\ c\ (l(a{:=}z))\ r)\ p\ stack = List\ (S\ c\ l\ r)\ p\ stack$
apply(*induct-tac stack*)
apply(*simp add*:*fun-upd-apply S-def*)+
done

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

$\forall p.\ a \notin set\ stack \longrightarrow List\ (S\ (c(a{:=}z))\ l\ r)\ p\ stack = List\ (S\ c\ l\ r)\ p\ stack$
apply(*induct-tac stack*)
apply(*simp add:fun-upd-apply S-def*)+
done

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*) ∧
iL p = (*if c p then l p else t*) ∧
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*
apply (*induct xs*)
apply (*auto simp:eq-sym-conv*)
done

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*
apply (*induct xs*)
apply (*auto simp:eq-sym-conv*)
done

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*
apply (*induct xs*)
apply (*auto simp:eq-sym-conv*)
done

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*
apply (*induct xs*)
apply (*auto simp:eq-sym-conv*)
done

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*
apply (*induct xs*)
apply (*auto simp:eq-sym-conv*)
done

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*
apply (*induct xs*)
apply (*auto simp*:*eq-sym-conv*)
done

15.2 The Schorr-Waite algorithm

theorem *SchorrWaiteAlgorithm*:
VARS c m l r t p q root
{*R* = *reachable* (*relS* {*l*, *r*}) {*root*} ∧ (∀ *x*. ¬ *m x*) ∧ *iR* = *r* ∧ *iL* = *l*}
t := *root*; *p* := *Null*;
WHILE p ≠ *Null* ∨ *t* ≠ *Null* ∧ ¬ *t*^.*m*
INV {∃ *stack*.
List (*S c l r*) *p stack* ∧ — *i1*
(∀ *x* ∈ *set stack*. *m x*) ∧ — *i2*
R = *reachable* (*relS*{*l*, *r*}) {*t*,*p*} ∧ — *i3*
(∀ *x*. *x* ∈ *R* ∧ ¬*m x* ⟶ — *i4*
x ∈ *reachable* (*relS*{*l*,*r*}|*m*) ({*t*}∪*set*(*map r stack*))) ∧
(∀ *x*. *m x* ⟶ *x* ∈ *R*) ∧ — *i5*
(∀ *x*. *x* ∉ *set stack* ⟶ *r x* = *iR x* ∧ *l x* = *iL x*) ∧ — *i6*
(*stkOk c l r iL iR t stack*) — *i7*}
DO IF t = *Null* ∨ *t*^.*m*
THEN IF p^.*c*
THEN q := *t*; *t* := *p*; *p* := *p*^.*r*; *t*^.*r* := *q* — *pop*
ELSE q := *t*; *t* := *p*^.*r*; *p*^.*r* := *p*^.*l*; — *swing*
p^.*l* := *q*; *p*^.*c* := *True* *FI*
ELSE q := *p*; *p* := *t*; *t* := *t*^.*l*; *p*^.*l* := *q*; — *push*
p^.*m* := *True*; *p*^.*c* := *False* *FI* *OD*
{(∀ *x*. (*x* ∈ *R*) = *m x*) ∧ (*r* = *iR* ∧ *l* = *iL*) }
(**is** *Valid*
{(*c*, *m*, *l*, *r*, *t*, *p*, *q*, *root*). *?Pre c m l r root*}
(*Seq* - (*Seq* - (*While* {(*c*, *m*, *l*, *r*, *t*, *p*, *q*, *root*). *?whileB m t p*} -)))
(*Aseq* - (*Aseq* - (*Awhile* {(*c*, *m*, *l*, *r*, *t*, *p*, *q*, *root*). *?inv c m l r t p*} - -))) -)
proof (*vcg*)
{
fix *c m l r t p q root*
assume *?Pre c m l r root*
thus *?inv c m l r root Null* **by** (*auto simp add*: *reachable-def addrs-def*)
next
fix *c m l r t p q*
let ∃ *stack*. *?Inv stack* = *?inv c m l r t p*
assume *a*: *?inv c m l r t p* ∧ ¬(*p* ≠ *Null* ∨ *t* ≠ *Null* ∧ ¬ *t*^.*m*)
then obtain *stack* **where** *inv*: *?Inv stack* **by** *blast*
from *a* **have** *pNull*: *p* = *Null* **and** *tDisj*: *t*=*Null* ∨ (*t*≠*Null* ∧ *t*^.*m*) **by** *auto*
let *?I1* ∧ - ∧ - ∧ *?I4* ∧ *?I5* ∧ *?I6* ∧ - = *?Inv stack*
from *inv* **have** *i1*: *?I1* **and** *i4*: *?I4* **and** *i5*: *?I5* **and** *i6*: *?I6* **by** *simp*+
from *pNull i1* **have** *stackEmpty*: *stack* = [] **by** *simp*
from *tDisj i4* **have** *RisMarked*[*rule-format*]: ∀ *x*. *x* ∈ *R* ⟶ *m x* **by**(*auto simp*: *reachable-def addrs-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 root*
let $\exists$ *stack. ?Inv stack* = *?inv c m l r t p*
let $\exists$ *stack. ?popInv stack* = *?inv c m l* $(r(p \rightarrow t))$ *p* $(p\hat{}.r)$
let $\exists$ *stack. ?swInv stack* =
?inv $(c(p \rightarrow True))$ *m* $(l(p \rightarrow t))$ $(r(p \rightarrow p\hat{}.l))$ $(p\hat{}.r)$ *p*
let $\exists$ *stack. ?puInv stack* =
?inv $(c(t \rightarrow False))$ $(m(t \rightarrow True))$ $(l(t \rightarrow p))$ *r* $(t\hat{}.l)$ *t*
let *?ifB1* = $(t = Null \vee t\hat{}.m)$
let *?ifB2* = $p\hat{}.c$

assume $(\exists\, stack.?Inv\ stack) \wedge$ *?whileB m t p*
then obtain *stack* **where** *inv*: *?Inv stack* **and** *whileB*: *?whileB m t p* **by** *blast*
let *?I1* $\wedge$ *?I2* $\wedge$ *?I3* $\wedge$ *?I4* $\wedge$ *?I5* $\wedge$ *?I6* $\wedge$ *?I7* = *?Inv 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*: *distinct* (*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*: $t = Null \vee t\hat{}.m$ **and** *ifB2*: $p\hat{}.c$
from *ifB1 whileB* **have** *pNotNull*: $p \neq Null$ **by** *auto*
then obtain *addr-p* **where** *addr-p-eq*: $p = Ref\ addr\text{-}p$ **by** *auto*
with *i1* **obtain** *stack-tl* **where** *stack-eq*: $stack = (addr\ p)\ \#\ stack\text{-}tl$
by *auto*
with *i2* **have** *m-addr-p*: $p\hat{}.m$ **by** *auto*
have *stackDist*: *distinct* (*stack*) **using** *i1* **by** (*rule List-distinct*)
from *stack-eq stackDist* **have** *p-notin-stack-tl*: $addr\ p \notin set\ stack\text{-}tl$ **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* $(S\ c\ l\ (r(p \rightarrow t)))\ (p\hat{}.r)$ *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*: $\forall\ x \in set\ stack\text{-}tl.\ m\ x$ **by** (*simp add*:*stack-eq*)
moreover

— Everything is still reachable:
let $(R = reachable\ ?Ra\ ?A) = ?I3$

let *?Rb* = (*relS* {*l*, *r*(*p* → *t*)})
let *?B* = {*p*, *p*⌢*r*}
— Our goal is *R* = *reachable ?Rb ?B*.
have *?Ra** `` *addrs ?A* = *?Rb*\* `` *addrs ?B* (**is** *?L* = *?R*)
proof
show *?L* ⊆ *?R*
proof (*rule still-reachable*)
show *addrs ?A* ⊆ *?Rb** `` *addrs ?B* **by**(*fastforce simp*:*addrs-def relS-def rel-def addr-p-eq*
intro:*oneStep-reachable Image-iff*[*THEN iffD2*])
show ∀(*x*,*y*) ∈ *?Ra*−*?Rb*. *y* ∈ (*?Rb** `` *addrs ?B*) **by** (*clarsimp simp*:*relS-def*)
(*fastforce simp add*:*rel-def Image-iff addrs-def dest*:*rel-upd1*)
qed
show *?R* ⊆ *?L*
proof (*rule still-reachable*)
show *addrs ?B* ⊆ *?Ra** `` *addrs ?A*
by(*fastforce simp*:*addrs-def rel-defs addr-p-eq*
intro:*oneStep-reachable Image-iff*[*THEN iffD2*])
next
show ∀(*x*, *y*)∈*?Rb*−*?Ra*. *y*∈(*?Ra** ``*addrs ?A*)
by (*clarsimp simp*:*relS-def*)
(*fastforce simp add*:*rel-def Image-iff addrs-def dest*:*rel-upd2*)
qed
qed
with *i3* **have** *poI3*: *R* = *reachable ?Rb ?B* **by** (*simp add*:*reachable-def*)
moreover

— If it is reachable and not marked, it is still reachable using...
let ∀*x*. *x* ∈ *R* ∧ ¬ *m x* ⟶ *x* ∈ *reachable ?Ra ?A* = *?I4*
let *?Rb* = *relS* {*l*, *r*(*p* → *t*)} | *m*
let *?B* = {*p*} ∪ *set* (*map* (*r*(*p* → *t*)) *stack-tl*)
— Our goal is ∀*x*. *x* ∈ *R* ∧ ¬ *m x* ⟶ *x* ∈ *reachable ?Rb ?B*.
let *?T* = {*t*, *p*⌢*r*}

have *?Ra** `` *addrs ?A* ⊆ *?Rb*\* `` (*addrs ?B* ∪ *addrs ?T*)
proof (*rule still-reachable*)
have *rewrite*: ∀*s*∈*set stack-tl*. (*r*(*p* → *t*)) *s* = *r s*
by (*auto simp add*:*p-notin-stack-tl intro*:*fun-upd-other*)
show *addrs ?A* ⊆ *?Rb** `` (*addrs ?B* ∪ *addrs ?T*)
by (*fastforce cong*:*map-cong simp*:*stack-eq addrs-def rewrite intro*:*self-reachable*)
show ∀(*x*, *y*)∈*?Ra*−*?Rb*. *y*∈(*?Rb** ``(*addrs ?B* ∪ *addrs ?T*))
by (*clarsimp simp*:*restr-def relS-def*)
(*fastforce simp add*:*rel-def Image-iff addrs-def dest*:*rel-upd1*)
qed
— We now bring a term from the right to the left of the subset relation.
hence *subset*: *?Ra** `` *addrs ?A* − *?Rb*\* `` *addrs ?T* ⊆ *?Rb** `` *addrs ?B*
by *blast*

```
      have poI4: ∀ x. x ∈ R ∧ ¬ m x ⟶ x ∈ reachable ?Rb ?B
      proof (rule allI, rule impI)
        fix x
        assume a: x ∈ R ∧ ¬ m x
        — First, a disjunction on r (addr p) used later in the proof
        have pDisj:p^.r = Null ∨ (p^.r ≠ Null ∧ p^.r^.m) using poI1 poI2
          by auto
        — x belongs to the left hand side of subset:
       have incl: x ∈ ?Ra* ‘‘addrs ?A using  a i4 by (simp only:reachable-def,
clarsimp)
          have excl: x ∉ ?Rb* ‘‘ addrs ?T using pDisj ifB1 a by (auto simp
add:addrs-def)
        — And therefore also belongs to the right hand side of subset,
        — which corresponds to our goal.
          from incl excl subset  show x ∈ reachable ?Rb ?B by (auto simp
add:reachable-def)
      qed
      moreover

      — If it is marked, then it is reachable
      from i5 have poI5: ∀ x. m x ⟶ x ∈ R .
      moreover

      — If it is not on the stack, then its l and r fields are unchanged
      from i7 i6 ifB2
      have poI6: ∀ x. x ∉ set stack-tl ⟶ (r(p → t)) x = iR x ∧ 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 ∃ stack. ?popInv stack ..
  }
  moreover

  — Proofs of the Swing and Push arm follow.
  — Since they are in principle simmilar to the Pop arm proof,
  — we show fewer comments and use frequent pattern matching.
  {
    — Swing arm
    assume ifB1: ?ifB1 and nifB2: ¬?ifB2
    from ifB1 whileB have pNotNull: p ≠ Null by clarsimp
    then obtain addr-p where addr-p-eq: p = Ref addr-p by clarsimp
     with i1 obtain stack-tl where stack-eq: stack = (addr p) # stack-tl by
```

clarsimp
with *i2* **have** *m-addr-p*: $p\hat{}.m$ **by** *clarsimp*
from *stack-eq stackDist* **have** *p-notin-stack-tl*: $(addr\ p) \notin set\ stack\text{-}tl$
by *simp*
let $?swI1 \wedge ?swI2 \wedge ?swI3 \wedge ?swI4 \wedge ?swI5 \wedge ?swI6 \wedge ?swI7 = ?swInv\ stack$
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 $R = reachable\ ?Ra\ ?A = ?I3$
let $R = reachable\ ?Rb\ ?B = ?swI3$
have $?Ra^*\ ``\ addrs\ ?A = ?Rb^*\ ``\ addrs\ ?B$
proof (*rule still-reachable-eq*)
show *addrs ?A* $\subseteq$ $?Rb^*\ ``\ addrs\ ?B$
by(*fastforce simp*:*addrs-def rel-defs addr-p-eq intro*:*oneStep-reachable Image-iff*[*THEN iffD2*])
next
show *addrs ?B* $\subseteq$ $?Ra^*\ ``\ addrs\ ?A$
by(*fastforce simp*:*addrs-def rel-defs addr-p-eq intro*:*oneStep-reachable Image-iff*[*THEN iffD2*])
next
show $\forall (x,\ y) \in ?Ra - ?Rb.\ y \in (?Rb^*\ ``addrs\ ?B)$
by (*clarsimp simp*:*relS-def*) (*fastforce simp add*:*rel-def Image-iff addrs-def fun-upd-apply dest*:*rel-upd1*)
next
show $\forall (x,\ y) \in ?Rb - ?Ra.\ y \in (?Ra^*\ ``addrs\ ?A)$
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 $\forall x.\ x \in R \wedge \neg\ m\ x \longrightarrow x \in reachable\ ?Ra\ ?A = ?I4$
let $\forall x.\ x \in R \wedge \neg\ m\ x \longrightarrow x \in reachable\ ?Rb\ ?B = ?swI4$
let $?T = \{t\}$
have $?Ra^*\ ``addrs\ ?A \subseteq ?Rb^*\ ``(addrs\ ?B \cup addrs\ ?T)$

proof (*rule still-reachable*)
have *rewrite*: $(\forall\, s{\in}set\ stack\text{-}tl.\ (r(addr\ p := l(addr\ p)))\ s = r\ s)$
by (*auto simp add*:*p-notin-stack-tl intro*:*fun-upd-other*)
show $addrs\ ?A \subseteq ?Rb^*\ ``\ (addrs\ ?B \cup addrs\ ?T)$
by (*fastforce cong*:*map-cong simp*:*stack-eq addrs-def rewrite intro*:*self-reachable*)
next
show $\forall\,(x,\ y){\in}?Ra{-}?Rb.\ y{\in}(?Rb^*``(addrs\ ?B \cup addrs\ ?T))$
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*: $?Ra^*``addrs\ ?A - ?Rb^*``addrs\ ?T \subseteq ?Rb^*``addrs\ ?B$
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*: $x \in ?Ra^*``addrs\ ?A$
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: ¬?ifB1
  from nifB1 whileB have tNotNull: t ≠ 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: ¬ (t^.m) by clarsimp
  with i2 have t-notin-stack: (addr t) ∉ set stack by blast
  let ?puI1∧?puI2∧?puI3∧?puI4∧?puI5∧?puI6∧?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* `` addrs ?A = ?Rb* `` addrs ?B
    proof (rule still-reachable-eq)
      show addrs ?A ⊆ ?Rb* `` addrs ?B
        by(fastforce simp:addrs-def rel-defs addr-t-eq intro:oneStep-reachable
Image-iff[THEN iffD2])
    next
      show addrs ?B ⊆ ?Ra* `` addrs ?A
        by(fastforce simp:addrs-def rel-defs addr-t-eq intro:oneStep-reachable
Image-iff[THEN iffD2])
    next
      show ∀(x, y)∈?Ra−?Rb. y∈(?Rb*``addrs ?B)
     by (clarsimp simp:relS-def) (fastforce simp add:rel-def Image-iff addrs-def
dest:rel-upd1)
    next
      show ∀(x, y)∈?Rb−?Ra. y∈(?Ra*``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*``addrs ?A* $\subseteq$ *?Rb\*``(addrs ?B* $\cup$ *addrs ?T)*
proof (*rule still-reachable*)
show *addrs ?A* $\subseteq$ *?Rb* `` (addrs ?B* $\cup$ *addrs ?T)*
by (*fastforce simp*:*new-stack-eq addrs-def intro*:*self-reachable*)
next
show $\forall (x,\ y) \in$ *?Ra* $-$ *?Rb. y* $\in$ *(?Rb*``(addrs ?B* $\cup$ *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*``addrs ?A − ?Rb\*``addrs ?T* $\subseteq$ *?Rb*``addrs ?B*
by *blast*
have *?puI4*
proof (*rule allI*, *rule impI*)
fix *x*
assume *a*: $x \in R \wedge \neg$ *?new-m x*
have *xDisj*: $x = ($*addr t*$) \vee x \neq ($*addr t*$)$ **by** *simp*
with *i4 a* **have** *inc*: $x \in$ *?Ra*``addrs ?A*
by (*fastforce simp*:*addr-t-eq addrs-def reachable-def intro*:*self-reachable*)
have *exc*: $x \notin$ *?Rb*`` addrs ?T*
using *xDisj a n-m-addr-t*
by (*clarsimp simp add*:*addrs-def addr-t-eq*)
from *inc exc subset* **show** $x \in$ *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 ∃ stack. ?puInv stack by blast

    }
    ultimately show ?thesis by blast
  qed
 }
qed

end
```

16 Heap abstractions for Separation Logic

(at the moment only Path and List)

```
theory SepLogHeap
 imports Main
begin

type-synonym heap = (nat ⇒ nat option)
```

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

16.1 Paths in the heap

```
primrec Path :: heap ⇒ nat ⇒ nat list ⇒ nat ⇒ bool
where
  Path h x [] y = (x = y)
| Path h x (a#as) y = (x≠0 ∧ a=x ∧ (∃ b. h x = Some b ∧ Path h b as y))

lemma [iff]: Path h 0 xs y = (xs = [] ∧ y = 0)
by (cases xs) simp-all

lemma [simp]: x≠0 ⟹ Path h x as z =
 (as = [] ∧ z = x ∨ (∃ y bs. as = x#bs ∧ h x = Some y & Path h y bs z))
by (cases as) auto

lemma [simp]: ⋀x. Path f x (as@bs) z = (∃ y. Path f x as y ∧ Path f y bs z)
by (induct as) auto

lemma Path-upd[simp]:
 ⋀x. u ∉ set as ⟹ Path (f(u := v)) x as y = Path f x as y
by (induct as) simp-all
```

16.2 Lists on the heap

```
definition List :: heap ⇒ nat ⇒ nat list ⇒ bool
```

where *List h x as = Path h x as 0*

lemma [*simp*]: *List h x [] = (x = 0)*
by (*simp add*: *List-def*)

lemma [*simp*]:
List h x (a#as) = (x≠0 ∧ a=x ∧ (∃ y. h x = Some y ∧ 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≠0 ⟹*
List h a as = (∃ b bs. as = a#bs ∧ h a = Some b ∧ List h b bs)
by (*cases as*) *simp-all*

theorem *notin-List-update*[*simp*]:
⋀x. a ∉ set as ⟹ List (h(a := y)) x as = List h x as
by (*induct as*) *simp-all*

lemma *List-unique*: *⋀x bs. List h x as ⟹ List h x bs ⟹ as = bs*
by (*induct as*) (*auto simp add*:*List-non-null*)

lemma *List-unique1*: *List h p as ⟹ ∃!as. List h p as*
by (*blast intro*: *List-unique*)

lemma *List-app*: *⋀x. List h x (as@bs) = (∃ y. Path h x as y ∧ List h y bs)*
by (*induct as*) *auto*

lemma *List-hd-not-in-tl*[*simp*]: *List h b as ⟹ h a = Some b ⟹ a ∉ 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*]: *⋀x. List h x as ⟹ distinct as*
by (*induct as*) (*auto dest*:*List-hd-not-in-tl*)

lemma *list-in-heap*: *⋀p. List h p ps ⟹ set ps ⊆ dom h*
by (*induct ps*) *auto*

lemma *list-ortho-sum1*[*simp*]:
⋀p. ⟦ List h1 p ps; dom h1 ∩ dom h2 = {}⟧ ⟹ List (h1++h2) p ps
by (*induct ps*) (*auto simp add*:*map-add-def split*:*option.split*)

lemma *list-ortho-sum2*[*simp*]:
⋀p. ⟦ List h2 p ps; dom h1 ∩ dom h2 = {}⟧ ⟹ List (h1++h2) p ps
by (*induct ps*) (*auto simp add*:*map-add-def split*:*option.split*)

end

17 Separation logic

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

The semantic definition of a few connectives:

definition *ortho* :: *heap* ⇒ *heap* ⇒ *bool* (**infix** ‹⊥› *55*)
where *h1* ⊥ *h2* ⟷ *dom h1* ∩ *dom h2* = {}

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

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

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

definition *wand*:: (*heap* ⇒ *bool*) ⇒ (*heap* ⇒ *bool*) ⇒ (*heap* ⇒ *bool*)
where *wand P Q* = (λ*h*. ∀*h'*. *h'* ⊥ *h* ∧ *P h'* ⟶ *Q*(*h*++*h'*))

This is what assertions look like without any syntactic sugar:

lemma *VARS x y z w h*
{*star* (%*h*. *singl h x y*) (%*h*. *singl h z w*) *h*}
SKIP
{*x* ≠ *z*}
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 Hs - otherwise they may bind the implicit H.

syntax
-emp :: *bool* (‹*emp*›)
-singl :: *nat* ⇒ *nat* ⇒ *bool* (‹(‹*open-block notation*=‹*mixfix singl*››[- ↦ -])›)
-star :: *bool* ⇒ *bool* ⇒ *bool* (**infixl** ‹**› *60*)
-wand :: *bool* ⇒ *bool* ⇒ *bool* (**infixl** ‹−*› *60*)

syntax-consts
-emp ⇌ *is-empty* **and**
-singl ⇌ *singl* **and**

-star $\rightleftharpoons$ *star* **and**
-wand $\rightleftharpoons$ *wand*

ML ‹
— *free-tr* takes care of free vars in the scope of separation logic connectives: they are implicitly applied to the heap
fun free-tr(*t as Free* -) = *t* $ *Syntax.free H*
| 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*);
›

parse-translation ‹
[(***syntax-const***‹*-emp*›, *K emp-tr*),
(***syntax-const***‹*-singl*›, *K singl-tr*),
(***syntax-const***‹*-star*›, *K star-tr*),
(***syntax-const***‹*-wand*›, *K 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*(-,-,(*t as Const*(*List*.-) $ *B* $ *p* $ *ps*)) = *t as* $ *p* $ *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*›, *K is-empty-tr'*),
  (***const-syntax*** ‹*singl*›, *K singl-tr'*),
  (***const-syntax*** ‹*star*›, *K star-tr'*),
  (***const-syntax*** ‹*wand*›, *K wand-tr'*)]
›

Now the intermediate proof states are also readable:

lemma *VARS H x y z w*
 {[*x*↦*y*] ** [*z*↦*w*]}
 y := *w*
 {*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

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*≠*0* ∧ [*p* ↦ *x*] ** *List H q qs*}
H := *H*(*p* ↦ *q*)
{*List H p* (*p*#*qs*)}
apply *vcg*
apply(*simp add*: *star-def ortho-def singl-def*)
apply *clarify*
apply(*subgoal-tac p* ∉ *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 ≠ *0*
INV {∃*ps qs*. (*List H p ps* ** *List H q qs*) ∧ *rev ps* @ *qs* = *rev Ps* @ *Qs*}
DO r := *p*; *p* := *the*(*H p*); *H* := *H*(*r* ↦ *q*); *q* := *r OD*
{*List H q* (*rev Ps* @ *Qs*)}
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*↦*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

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.