

Hoare Logic

Norbert Galm
Walter Guttman
Farhad Mehta
Tobias Nipkow
Leonor Prensa Nieto

December 12, 2021

Abstract

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

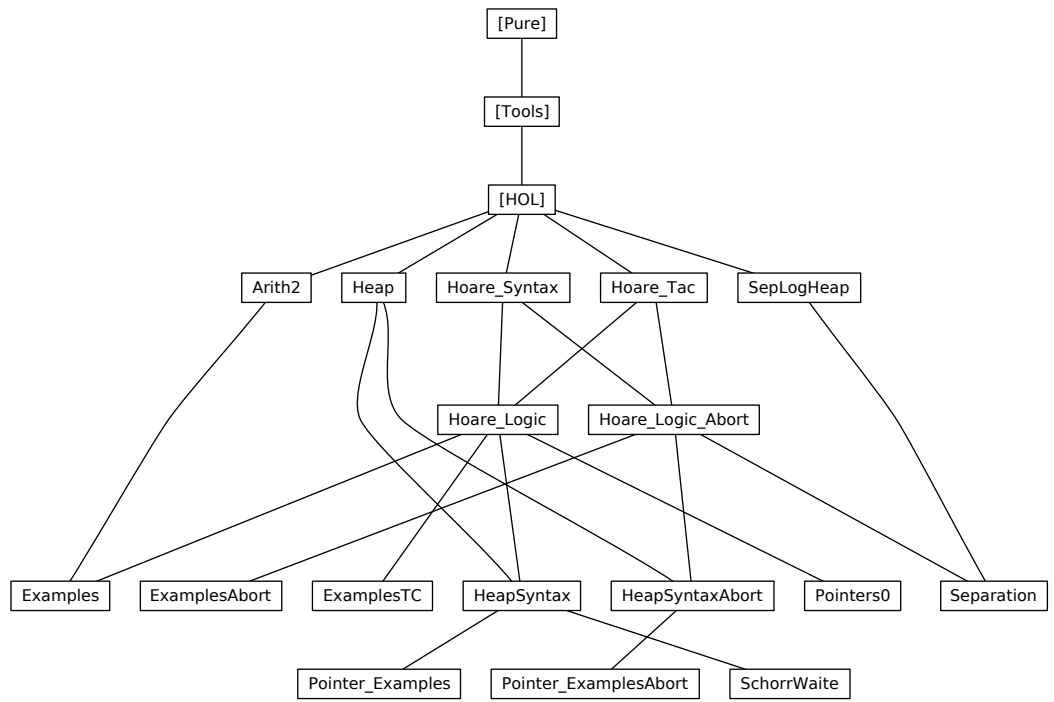
Special infrastructure for modelling and reasoning about pointer programs is provided, together with many examples, including Schorr-Waite. See [1, 2] for an excellent exposition.

Contents

1	Concrete syntax for Hoare logic, with translations for variables	5
2	Hoare logic VCG tactic	6
3	Hoare logic	6
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	12
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	34
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	39
12.1 Verifications	40
12.1.1 List reversal	40
12.1.2 Searching in a list	41
12.1.3 Splicing two lists	43
12.1.4 Merging two lists	43
12.1.5 Cyclic list reversal	47
12.1.6 Storage allocation	48
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	50
15.1 Machinery for the Schorr-Waite proof	50
15.2 The Schorr-Waite algorithm	54
16 Heap abstractions for Separation Logic	63
16.1 Paths in the heap	63
16.2 Lists on the heap	63
17 Separation logic	64



1 Concrete syntax for Hoare logic, with translations for variables

```
theory Hoare-Syntax
imports Main
begin
```

```
syntax
```

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

The $VAR \{-\}$ syntax supports two variants:

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

```
syntax
```

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

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

```
syntax
```

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

```
syntax (output)
```

```
-hoare :: ['assn, 'com, 'assn] ⇒ bool ({-} // - // {-} [0, 55, 0] 50)
-hoare-tc :: ['assn, 'com, 'assn] ⇒ bool ([-] // - // [-] [0, 55, 0] 50)
```

Completeness requires(?) the ability to refer to an outer variant in an inner invariant. Thus we need to abstract over a variable equated with the variant, the x in $VAR \{x = t\}$. But the x should only occur in invariants. To enforce this, syntax translations in *hoare-syntax.ML* separate the program from its annotations and only the latter are abstracted over 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  $\Rightarrow$  nat nat  $\Rightarrow$  'a anno

```

ML-file $\langle hoare-syntax.ML \rangle$

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

```

ML-file $\langle hoare-tac.ML \rangle$

end

end

3 Hoare logic

```

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

```

3.1 Sugared semantic embedding of Hoare logic

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

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

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

datatype `'a com =`

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

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

type-synonym `'a sem = 'a  $\Rightarrow$  'a  $\Rightarrow$  bool`

inductive `Sem :: 'a com  $\Rightarrow$  'a sem`

where

`Sem (Basic f) s (f s)`
`| Sem c1 s s''  $\Longrightarrow$  Sem c2 s'' s'  $\Longrightarrow$  Sem (Seq c1 c2) s s'`
`| s  $\in$  b  $\Longrightarrow$  Sem c1 s s'  $\Longrightarrow$  Sem (Cond b c1 c2) s s'`
`| s  $\notin$  b  $\Longrightarrow$  Sem c2 s s'  $\Longrightarrow$  Sem (Cond b c1 c2) s s'`
`| s  $\notin$  b  $\Longrightarrow$  Sem (While b c) s s`
`| s  $\in$  b  $\Longrightarrow$  Sem c s s''  $\Longrightarrow$  Sem (While b c) s'' s'  $\Longrightarrow$`
`Sem (While b c) s s'`

definition `Valid :: 'a bexp  $\Rightarrow$  'a com  $\Rightarrow$  'a anno  $\Rightarrow$  'a bexp  $\Rightarrow$  bool`

where `Valid p c a q  $\equiv \forall s s'. Sem c s s' \longrightarrow s \in p \longrightarrow s' \in q$`

definition `ValidTC :: 'a bexp  $\Rightarrow$  'a com  $\Rightarrow$  'a anno  $\Rightarrow$  'a bexp  $\Rightarrow$  bool`

where `ValidTC p c a q  $\equiv \forall s. s \in p \longrightarrow (\exists t. Sem c s t \wedge t \in q)$`

inductive-cases `[elim!]:`

`Sem (Basic f) s s' Sem (Seq c1 c2) s s'`
`Sem (Cond b c1 c2) s s'`

lemma *Sem-deterministic:*

assumes `Sem c s s1`

and `Sem c s s2`

shows `s1 = s2`

proof `—`

have `Sem c s s1  $\Longrightarrow$  ( $\forall s2. Sem c s s2 \longrightarrow s1 = s2$ )`

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 $\implies$ Valid p c a q

by (*metis Sem-deterministic Valid-def ValidTC-def*)

lemma *tc-extract-function*:

ValidTC p c a q $\implies \exists f . \forall s . s \in p \longrightarrow f s \in q$

by (*metis ValidTC-def*)

lemma *SkipRule*: *p $\subseteq$ q $\implies$ Valid p (Basic id) a q*

by (*auto simp: Valid-def*)

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

by (*auto simp: Valid-def*)

lemma *SeqRule*: *Valid P c1 a1 Q $\implies$ Valid Q c2 a2 R $\implies$ Valid P (Seq c1 c2) (Aseq a1 a2) R*

by (*auto simp: Valid-def*)

lemma *CondRule*:

p $\subseteq$ {s. (s $\in$ b $\longrightarrow$ s $\in$ w) $\wedge$ (s $\notin$ b $\longrightarrow$ s $\in$ w')}

$\implies$ Valid w c1 a1 q $\implies$ Valid w' c2 a2 q $\implies$ 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 *$\forall s s' . \text{Sem } c s s' \longrightarrow s \in I \wedge s \in b \longrightarrow s' \in I \implies$*

$s \in I \implies s' \in I \wedge s' \notin b$

using *assms*

by (*induct While b c s s' auto*)

lemma *WhileRule*:

p $\subseteq$ i $\implies$ Valid (i $\cap$ b) c (A 0) i $\implies$ i $\cap$ ($\neg$ b) $\subseteq$ q $\implies$ 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 $\text{ValidTC } p \text{ (Basic } f) a q$
by (*metis* *assms* *Ball-Collect Sem.intros(1)* *ValidTC-def*)

lemma *SeqRuleTC*:
assumes $\text{ValidTC } p \text{ } c1 \text{ } a1 \text{ } q$
and $\text{ValidTC } q \text{ } c2 \text{ } a2 \text{ } r$
shows $\text{ValidTC } p \text{ (Seq } c1 \text{ } c2) \text{ (Aseq } a1 \text{ } a2) \text{ } r$
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 $\text{ValidTC } w \text{ } c1 \text{ } a1 \text{ } q$
and $\text{ValidTC } w' \text{ } c2 \text{ } a2 \text{ } q$
shows $\text{ValidTC } p \text{ (Cond } b \text{ } c1 \text{ } c2) \text{ (Acond } a1 \text{ } a2) \text{ } q$
proof (*unfold* *ValidTC-def*, *rule* *allI*)
fix s
show $s \in p \longrightarrow (\exists t. \text{Sem } (\text{Cond } b \text{ } c1 \text{ } c2) \text{ } s \text{ } t \wedge t \in q)$
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 $\bigwedge n::\text{nat}. \text{ValidTC } (i \cap b \cap \{s. v \text{ } s = n\}) \text{ } c \text{ } (A \text{ } n) \text{ } (i \cap \{s. v \text{ } s < n\})$
and $i \cap \text{uminus } b \subseteq q$
shows $\text{ValidTC } p \text{ (While } b \text{ } c) \text{ (Awhile } i \text{ } v \text{ } (\lambda n. A \text{ } n)) \text{ } q$
proof –
{
fix $s \text{ } n$
have $s \in i \wedge v \text{ } s = n \longrightarrow (\exists t. \text{Sem } (\text{While } b \text{ } c) \text{ } s \text{ } t \wedge t \in q)$
proof (*induction* n *arbitrary*: s *rule*: *less-induct*)
fix $n :: \text{nat}$
fix $s :: 'a$
assume $1: \bigwedge (m::\text{nat}) \text{ } s::'a. m < n \implies s \in i \wedge v \text{ } s = m \longrightarrow (\exists t. \text{Sem } (\text{While } b \text{ } c) \text{ } s \text{ } t \wedge t \in q)$
show $s \in i \wedge v \text{ } s = n \longrightarrow (\exists t. \text{Sem } (\text{While } b \text{ } c) \text{ } s \text{ } t \wedge t \in q)$
proof (*rule* *impI*, *cases* $s \in b$)
assume $2: s \in b$ **and** $s \in i \wedge v \text{ } s = n$
hence $s \in i \cap b \cap \{s. v \text{ } s = n\}$
using *assms(1)* **by** *auto*
hence $\exists t. \text{Sem } c \text{ } s \text{ } t \wedge t \in i \cap \{s. v \text{ } s < n\}$
by (*metis* *assms(2)* *ValidTC-def*)
from *this* **obtain** t **where** $3: \text{Sem } c \text{ } s \text{ } t \wedge t \in i \cap \{s. v \text{ } s < n\}$
by *auto*
hence $\exists u. \text{Sem } (\text{While } b \text{ } c) \text{ } t \text{ } u \wedge u \in q$
using 1 **by** *auto*
thus $\exists t. \text{Sem } (\text{While } b \text{ } c) \text{ } s \text{ } t \wedge t \in q$
using $2 \text{ } 3$ *Sem.intros(6)* **by** *force*

```

    next
    assume  $s \notin b$  and  $s \in i \wedge v\ s = n$ 
    thus  $\exists t. \text{Sem } (\text{While } b\ c)\ s\ t \wedge t \in q$ 
    using  $\text{Sem.intros}(5)\ \text{assms}(3)$  by fastforce
  qed
qed
}
thus ?thesis
using  $\text{assms}(1)\ \text{ValidTC-def}$  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 *vsg-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* :: $[nat, nat, nat] \Rightarrow bool$

where *cd* *x m n* $\longleftrightarrow x \text{ dvd } m \wedge x \text{ dvd } n$

definition *gcd* :: $[nat, nat] \Rightarrow nat$

where *gcd* *m n* = (*SOME* *x*. *cd* *x m n* & $(\forall y. (cd \ y \ m \ n) \longrightarrow y \leq x)$)

primrec *fac* :: $nat \Rightarrow nat$

where

fac 0 = *Suc* 0

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

4.1 cd

lemma *cd-nnn*: $0 < n \implies cd \ n \ n \ n$

apply (*simp* *add*: *cd-def*)

done

lemma *cd-le*: $[[\text{cd } x \ m \ n; 0 < m; 0 < n]] \implies x \leq m \ \& \ x \leq n$

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*: $n \leq m \implies cd \ x \ m \ n = cd \ x \ (m - n) \ n$

apply (*unfold* *cd-def*)

apply (*fastforce* *dest*: *dvd-diffD*)

done

```

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

```

4.2 gcd

```

lemma gcd-nnn:  $0 < n \implies n = gcd\ n\ n$ 
  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:  $n \leq m \implies gcd\ m\ n = gcd\ (m-n)\ n$ 
  apply (unfold gcd-def)
  apply (subgoal-tac  $n \leq m \implies \forall x. cd\ x\ m\ n = cd\ x\ (m-n)\ n$ )
  apply simp
  apply (rule allI)
  apply (erule cd-diff-l)
done

```

```

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

```

4.3 pow

```

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

$\{a=A \wedge b=B\}$

$m := 0; s := 0;$

WHILE $m \neq a$

INV $\{s=m*b \wedge a=A \wedge b=B\}$

DO $s := s+b; m := m+(1::\text{nat})$ *OD*

$\{s = A*B\}$

by *vcg-simp*

lemma *multiply-by-add-time*: $\text{VARs } m \ s \ a \ b \ t$

$\{a=A \wedge b=B \wedge t=0\}$

$m := 0; t := t+1; s := 0; t := t+1;$

WHILE $m \neq a$

INV $\{s=m*b \wedge a=A \wedge b=B \wedge t = 2*m + 2\}$

DO $s := s+b; t := t+1; m := m+(1::\text{nat}); t := t+1$ *OD*

$\{s = A*B \wedge t = 2*A + 2\}$

by *vcg-simp*

lemma *multiply-by-add2*: $\text{VARs } M \ N \ P :: \text{int}$

$\{m=M \wedge n=N\}$

IF $M < 0$ *THEN* $M := -M; N := -N$ *ELSE SKIP FI*;

$P := 0;$

WHILE $0 < M$

INV $\{0 \leq M \wedge (\exists p. p = (\text{if } m < 0 \text{ then } -m \text{ else } m) \ \& \ p*N = m*n \ \& \ P = (p-M)*N)\}$

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*: $\text{VARs } M \ N \ P \ t :: \text{int}$

$\{m=M \wedge n=N \wedge t=0\}$

IF $M < 0$ *THEN* $M := -M; t := t+1; N := -N; t := t+1$ *ELSE SKIP FI*;

$P := 0; t := t+1;$

WHILE $0 < M$

INV $\{0 \leq M \ \& \ (\exists p. p = (\text{if } m < 0 \text{ then } -m \text{ else } m) \ \& \ p*N = m*n \ \& \ P = (p-M)*N \ \& \ t \geq 0 \ \& \ t \leq 2*(p-M)+3)\}$

DO $P := P+N; t := t+1; M := M - 1; t := t+1$ *OD*

$\{P = m*n \ \& \ t \leq 2*\text{abs } m + 3\}$

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 ≠ 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 ≠ b
INV {0 < a & 0 < b & gcd A B = gcd a b & a ≤ A & b ≤ B & t ≤ max A B - max
a b + 2}
DO IF a < b THEN b := b - a; t := t + 1 ELSE a := a - b; t := t + 1 FI OD
{a = gcd A B & t ≤ max A B + 2}
```

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. Dijkstra. 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 multiplying with $gcd\ x\ y$.

lemmas *distrib* =

diff-mult-distrib diff-mult-distrib2 add-mult-distrib add-mult-distrib2

lemma *gcd-scm*: *VARs a b x y*

```
{0 < A & 0 < B & a = A & b = B & x = B & y = A}
WHILE a ~ = 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 ≤ i ⇒ fac (i - Suc 0) * i = fac i

```

by(*induct i, simp-all*)

lemma factorial2: *VARs i f*
 $\{True\}$
 $i := (1::nat); f := 1;$
 $WHILE\ i \leq n\ INV\ \{f = fac(i - 1) \ \&\ 1 \leq i \ \&\ i \leq n+1\}$
 $DO\ f := f*i; i := i+1\ OD$
 $\{f = fac\ n\}$
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*
 $\{True\}$
 $r := (0::nat);$
 $WHILE\ (r+1)*(r+1) \leq X$
 $INV\ \{r*r \leq X\}$
 $DO\ r := r+1\ OD$
 $\{r*r \leq X \ \&\ X < (r+1)*(r+1)\}$
apply *vcg-simp*
done

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

— without multiplication

lemma *sqr-without-multiplication*: *VARs* *u w r*
 $\{x=X\}$
 $u := 1; w := 1; r := (0::nat);$
 $WHILE\ w \leq X$
 $INV\ \{u = r+r+1 \ \&\ w = (r+1)*(r+1) \ \&\ r*r \leq X\}$
 $DO\ r := r + 1; w := w + u + 2; u := u + 2\ OD$
 $\{r*r \leq X \ \&\ X < (r+1)*(r+1)\}$
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*
 $\{x=X \ \&\ y=Y\}$
 $x := rev(x);$
 $WHILE\ x \sim = []$
 $INV\ \{rev(x)@y = X@Y\}$
 $DO\ y := (hd\ x \# y);$
 $\quad x := tl\ x$
 OD
 $\{y = X@Y\}$
apply *vcg-simp*
apply(*simp* *add*: *neq-Nil-conv*)
apply *auto*

done

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

$\{x=X \ \& \ y=Y\}$
 $x := \text{rev}(x); t := 0;$
 WHILE $x^\sim = []$
 INV $\{\text{rev}(x)@y = X@Y \ \& \ \text{length } x \leq \text{length } X \ \& \ t = 2 * (\text{length } X - \text{length } x)\}$
 DO $y := (\text{hd } x \# y); t := t+1;$
 $x := \text{tl } x; t := t+1$
 OD
 $\{y = X@Y \ \& \ t = 2 * \text{length } X\}$
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 $A\ i$

$\{\text{True}\}$
 $i := 0;$
 WHILE $i < \text{length } A \ \& \ A!i \neq \text{key}$
 INV $\{\forall j. j < i \longrightarrow A!j \neq \text{key}\}$
 DO $i := i+1$ OD
 $\{(i < \text{length } A \longrightarrow A!i = \text{key}) \ \& \ (i = \text{length } A \longrightarrow (\forall j. j < \text{length } A \longrightarrow A!j \neq \text{key}))\}$
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 $i < \text{length } A \ \wedge \ A!i \neq \text{key}$
 INV $\{(\forall j. j < i \longrightarrow A!j \neq \text{key}) \ \wedge \ i \leq \text{length } A \ \wedge \ t = i+1\}$
 DO $i := i+1; t := t+1$ OD
 $\{(i < \text{length } A \longrightarrow A!i = \text{key}) \ \wedge \ (i = \text{length } A \longrightarrow (\forall j. j < \text{length } A \longrightarrow A!j \neq \text{key})) \ \wedge \ t \leq \text{length } A + 1\}$
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 *lem*: $m - \text{Suc } 0 < n \implies m < \text{Suc } n$
by *arith*

lemma *Partition*:

```

[[ leq ==  $\lambda A\ i.\ \forall k.\ k < i \longrightarrow A!k \leq \text{pivot}$ ;
   geq ==  $\lambda A\ i.\ \forall k.\ i < k \wedge k < \text{length } A \longrightarrow \text{pivot} \leq A!k$  ]] ==>
  VARS  $A\ u\ l$ 
  {  $0 < \text{length}(A::('a::\text{order})\text{list})$  }
   $l := 0$ ;  $u := \text{length } A - \text{Suc } 0$ ;
  WHILE  $l \leq u$ 
  INV {  $\text{leq } A\ l \wedge \text{geq } A\ u \wedge u < \text{length } A \wedge l \leq \text{length } A$  }
  DO WHILE  $l < \text{length } A \wedge A!l \leq \text{pivot}$ 
  INV {  $\text{leq } A\ l \wedge \text{geq } A\ u \wedge u < \text{length } A \wedge l \leq \text{length } A$  }
  DO  $l := l + 1$  OD;
  WHILE  $0 < u \wedge \text{pivot} \leq A!u$ 
  INV {  $\text{leq } A\ l \wedge \text{geq } A\ u \wedge u < \text{length } A \wedge l \leq \text{length } A$  }
  DO  $u := u - 1$  OD;
  IF  $l \leq u$  THEN  $A := A[l := A!u, u := A!l]$  ELSE SKIP FI
  OD
  {  $\text{leq } A\ u \wedge (\forall k.\ u < k \wedge k < l \longrightarrow A!k = \text{pivot}) \wedge \text{geq } A\ l$  }
— expand and delete abbreviations first
apply (simp)
apply (erule thin-rl)+
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: lem)
apply (force simp: nth-list-update)
done

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 $'a\ \text{bexp} = 'a\ \text{set}$
type-synonym $'a\ \text{assn} = 'a\ \text{set}$
type-synonym $'a\ \text{var} = 'a \Rightarrow \text{nat}$

datatype $'a\ \text{com} =$
Basic $'a \Rightarrow 'a$

```

| 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*:

$ValidTC\ p\ c\ a\ q \implies \exists f . \forall s . s \in p \longrightarrow f\ s \in q$

by (*meson ValidTC-def*)

The proof rules for partial correctness

lemma *SkipRule*: $p \subseteq q \implies Valid\ p\ (Basic\ id)\ a\ q$

by (*auto simp: Valid-def*)

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

by (*auto simp: Valid-def*)

lemma *SeqRule*: $Valid\ P\ c1\ a1\ Q \implies Valid\ Q\ c2\ a2\ R \implies Valid\ P\ (Seq\ c1\ c2)\ (Aseq\ a1\ a2)\ R$

by (*auto simp: Valid-def*)

lemma *CondRule*:

$p \subseteq \{s. (s \in b \longrightarrow s \in w) \wedge (s \notin b \longrightarrow s \in w')\}$

$\implies Valid\ w\ c1\ a1\ q \implies Valid\ w'\ c2\ a2\ q \implies Valid\ p\ (Cond\ b\ c1\ c2)\ (Acond\ a1\ a2)\ q$

by (*fastforce simp: Valid-def image-def*)

lemma *While-aux*:

assumes *Sem* (*While* *b* *c*) *s* *s'*

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

using *assms*

by (*induct While* *b* *c* *s* *s'*) *auto*

lemma *WhileRule*:

$p \subseteq i \implies Valid\ (i \cap b)\ c\ (A\ 0)\ i \implies i \cap (-b) \subseteq q \implies Valid\ p\ (While\ b\ c)\ (Awhile\ i\ v\ A)\ q$

apply (*clarsimp simp: Valid-def*)

apply (*drule While-aux*)

apply *assumption*

apply *blast*

apply *blast*

done

lemma *AbortRule*: $p \subseteq \{s. False\} \implies Valid\ p\ Abort\ a\ q$

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 -
  {
    fix s n
    have s  $\in$  i  $\wedge$  v s = n  $\longrightarrow$  ( $\exists t$  . Sem (While b c) (Some s) (Some t)  $\wedge$  t  $\in$  q)
    proof (induction n arbitrary: s rule: less-induct)
      fix n :: nat
      fix s :: 'a
      assume 1:  $\bigwedge (m::nat)$  s::'a . m < n  $\implies$  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
      qed
    qed
  }

```

```

      using 2 3 Sem.intros(10) by force
    next
      assume  $s \notin b$  and  $s \in i \wedge v\ s = n$ 
      thus  $\exists t . \text{Sem } (\text{While } b\ c) (\text{Some } s) (\text{Some } t) \wedge t \in q$ 
        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 ((2-  $\rightarrow$ / -) 71)
  -array-update :: 'a list  $\Rightarrow$  nat  $\Rightarrow$  'a  $\Rightarrow$  'a com ((2-[-] :=/ -) [70, 65] 61)

```

```

translations
   $P \rightarrow c \rightleftharpoons \text{IF } P \text{ THEN } c \text{ ELSE CONST Abort FI}$ 
   $a[i] := v \rightarrow (i < \text{CONST length } a) \rightarrow (a := \text{CONST list-update } a\ i\ v)$ 
  — reverse translation not possible because of duplicate  $a$ 

```

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

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 ≠ 0} z ≠ 0 → x := y div z {x = 1}
by vcg-simp

```

```

lemma
  VARs a i j
  {k ≤ length a & i < k & j < k} j < length a → 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: VAR  $(p::int)$   $i$ 
[ True ]
 $p := 1$ ;
 $i := 0$ ;
WHILE  $i < n$ 
  INV  $\{ p = x^{\wedge}i \wedge i \leq n \}$ 
  VAR  $\{ n - i \}$ 
  DO  $p := p * x$ ;
     $i := i + 1$ 
  OD
[  $p = x^{\wedge}n$  ]
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: VAR  $r$ 
[  $0 \leq (x::int)$  ]
 $r := 0$ ;
WHILE  $(r+1)*(r+1) <= x$ 
  INV  $\{ r*r \leq x \}$ 
  VAR  $\{ \text{nat } (x-r) \}$ 
  DO  $r := r+1$  OD
[  $r*r \leq x \wedge x < (r+1)*(r+1)$  ]
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:
 $0 \leq (x::int) \implies \exists r' . r'*r' \leq x \wedge x < (r'+1)*(r'+1)$ 
using tc-extract-function sqrt-tc by blast

```

```

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

```

```

lemma sqrt-function:
  assumes  $0 \leq (x::int)$ 
  and  $r' = \text{sqrt } x$ 
  shows  $r'*r' \leq x \wedge x < (r'+1)*(r'+1)$ 
proof -
  let  $?P = \lambda r' . r'*r' \leq x \wedge x < (r'+1)*(r'+1)$ 
  have  $?P (\text{SOME } z . ?P z)$ 
    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 ≤ 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
    ((2- $\hat{\cdot}$ - := / -) [70,1000,65] 61)
  -faccess :: 'a::ref => ('a::ref  $\Rightarrow$  'v) => 'v
    (- $\hat{\cdot}$ - [65,1000] 65)

```

```

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

```

An example due to Suzuki:

```

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

```

9.3 The heap

9.3.1 Paths in the heap

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

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

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

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

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

9.3.2 Lists on the heap

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

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

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

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

lemma *List-Ref*[simp]:
 a $\neq$ Null $\implies$ *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]:
 $\bigwedge x. a \notin \text{set } as \implies \text{List } (h(a := y)) x as = \text{List } h x as$
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:  $\bigwedge x \text{ bs. List } h \ x \ as \implies List \ h \ x \ bs \implies as = bs$ 
by(induct as, simp, clarsimp)

```

```

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

```

```

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

```

```

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

```

```

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

```

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

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

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

```

9.4 Verifications

9.4.1 List reversal

A short but unreadable proof:

```

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

```

```

apply/rev-mt-tac/ps/qs
apply/clarify
apply/rev-mt-tac/ps'
apply/rule-tac/x/#/ps'/m/etI
apply/simp
apply/rule-tac/x/#/p#qs/m/etI
apply/simp
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 where I: ?I ps qs ∧ p ≠ Null by fast
  then obtain ps' where ps = p # ps' by fastforce
  hence List (tl(p := q)) (p^.tl) ps' ∧
      List (tl(p := q)) p (p#qs) ∧
      set ps' ∩ set (p#qs) = {} ∧
      rev ps' @ (p#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

```

Finally, 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  $p \neq \text{Null}$ 
  INV  $\{islist\ tl\ p \wedge islist\ tl\ q \wedge$ 
     $set(list\ tl\ p) \cap set(list\ tl\ q) = \{\} \wedge$ 
     $rev(list\ tl\ p) @ (list\ tl\ q) = rev\ Ps @ Qs\}$ 
  DO  $r := p; p := p.^{.}tl; r.^{.}tl := q; q := r$  OD
   $\{islist\ tl\ q \wedge list\ tl\ q = rev\ Ps @ Qs\}$ 
apply vcg-simp
apply clarsimp
apply clarsimp
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  $p \neq \text{Null} \wedge p \neq X$ 
  INV  $\{p \neq \text{Null} \wedge (\exists ps. List\ tl\ p\ ps \wedge X \in set\ ps)\}$ 
  DO  $p := p.^{.}tl$  OD
   $\{p = X\}$ 
apply vcg-simp
apply (case-tac  $p = \text{Null}$ )
apply clarsimp
apply fastforce
apply clarsimp
apply fastforce
apply clarsimp
done

```

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

```

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

```

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

```

lemma VARs tl p

```

```

  {(p,X) ∈ {(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*: $(P|Q \longrightarrow R) = ((P \longrightarrow R) \wedge (\sim P \longrightarrow Q \longrightarrow R))$
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 $\Rightarrow$ 'a::ref

```

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

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

lemma  $\sim \text{finite}(\text{UNIV}::('a::\text{ref})\text{set}) \implies$ 
  VARS xs elem next alloc p q
  {Xs = xs  $\wedge$  p = (Null::'a)}
  WHILE xs  $\neq$  []
  INV {islist next p  $\wedge$  set(list next p)  $\subseteq$  set alloc  $\wedge$ 
    map elem (rev(list next p)) @ xs = Xs}
  DO q := new(set alloc); alloc := q#alloc;
    q $\hat{\cdot}$ .next := p; q $\hat{\cdot}$ .elem := hd xs; xs := tl xs; p := q
  OD
  {islist next p  $\wedge$  map elem (rev(list next p)) = Xs}
apply vcg-simp
apply (clarsimp simp: subset-insert-iff neq-Nil-conv fun-upd-apply new-notin)
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) = ( $\exists y. x = \text{Ref } y$ )
by (induct x) auto

```

```

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

```

```

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

```

10.2 The heap

10.2.1 Paths in the heap

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

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

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

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

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

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

10.2.2 Non-repeating paths

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

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

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

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

10.2.3 Lists on the heap

Relational abstraction **definition** $List :: ('a \Rightarrow 'a \text{ ref}) \Rightarrow 'a \text{ ref} \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$

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

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

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

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

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

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

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

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

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

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

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

lemma $Path-is-List:$
 $\llbracket Path\ h\ b\ Ps\ (Ref\ a);\ a \notin set\ Ps \rrbracket \implies List\ (h(a := Null))\ b\ (Ps\ @\ [a])$
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) $\implies$ *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) $\implies$ 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 $\implies$ y $\notin$ set(*list* h p) $\implies$ *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 $\implies$ y $\notin$ set(*list* h p) $\implies$ *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)
    (-/'((-  $\rightarrow$  -)') [1000,0] 900)
  -fassign :: 'a ref  $\Rightarrow$  id  $\Rightarrow$  'v  $\Rightarrow$  's com
    ((2- $\wedge$ - := / -) [70,1000,65] 61)
  -faccess :: 'a ref  $\Rightarrow$  ('a ref  $\Rightarrow$  'v)  $\Rightarrow$  'v
    (- $\wedge$ - [65,1000] 65)
translations
  f(r  $\rightarrow$  v) == f(CONST addr r := v)
  p $\wedge$ .f := e  $\Rightarrow$  f := f(p  $\rightarrow$  e)
  p $\wedge$ .f  $\Rightarrow$  f(CONST addr p)

```

```

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

```

An example due to Suzuki:

```

lemma VARs v n
  {w = Ref w0 & x = Ref x0 & y = Ref y0 & z = Ref z0 &
   distinct[w0,x0,y0,z0]}
  w $\wedge$ .v := (1::int); w $\wedge$ .n := x;
  x $\wedge$ .v := 2; x $\wedge$ .n := y;
  y $\wedge$ .v := 3; y $\wedge$ .n := z;
  z $\wedge$ .v := 4; x $\wedge$ .n := z
  {w $\wedge$ .n $\wedge$ .n $\wedge$ .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 ∧ List tl q Qs ∧ set Ps ∩ set Qs = {}}
  WHILE p ≠ Null
  INV {∃ ps qs. List tl p ps ∧ List tl q qs ∧ set ps ∩ set qs = {} ∧
      rev ps @ qs = rev Ps @ Qs}
  DO r := p; p := p^.tl; r^.tl := q; q := r OD
  {List tl q (rev Ps @ Qs)}
apply vcg-simp
apply fastforce
apply (fastforce intro:notin-List-update[THEN iffD2])
— explicit:
apply (clarsimp)
apply (verifem-acc-ps-qs)
apply (clarsimp)
apply (verifem-acc-ps)
apply (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 ∧ List next q Qs ∧ set Ps ∩ set Qs = {} ∧
   ps = Ps ∧ qs = Qs}
  WHILE p ≠ Null
  INV {List next p ps ∧ List next q qs ∧ set ps ∩ set qs = {} ∧
      rev ps @ qs = rev Ps @ Qs}
  DO r := p; p := p^.next; r^.next := q; q := r;
     qs := (hd ps) # qs; ps := tl ps OD
  {List next q (rev Ps @ Qs)}
apply vcg-simp
apply fastforce
done

```

A longer readable version:

```

lemma VARS tl p q r
  {List tl p Ps ∧ List tl q Qs ∧ set Ps ∩ set Qs = {}}
  WHILE p ≠ Null
  INV {∃ ps qs. List tl p ps ∧ List tl q qs ∧ set ps ∩ set qs = {} ∧
      rev ps @ qs = rev Ps @ Qs}
  DO r := p; p := p^.tl; r^.tl := q; q := r OD
  {List tl q (rev Ps @ Qs)}
proof vcg
  fix tl p q r
  assume List tl p Ps ∧ List tl q Qs ∧ set Ps ∩ set Qs = {}
  thus ∃ ps qs. List tl p ps ∧ List tl q qs ∧ set ps ∩ set qs = {} ∧
      rev ps @ qs = rev Ps @ Qs by fastforce

```

```

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

```

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

```

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

```

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  $\wedge X \in set\ Ps$ }
WHILE  $p \neq Null \wedge p \neq Ref\ X$ 
INV { $\exists ps. List\ tl\ p\ ps \wedge X \in set\ ps$ }

```

```

    DO  $p := p \hat{.} tl$  OD
    { $p = Ref\ X$ }
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  $p \neq Null \wedge p \neq X$ 
  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. The first version uses a relation on *'a ref*:

```

lemma VARs tl p
  {( $p, X$ )  $\in \{(Ref\ x, tl\ x) \mid x. True\}^*$ }
  WHILE  $p \neq Null \wedge p \neq X$ 
  INV {( $p, X$ )  $\in \{(Ref\ x, tl\ x) \mid x. True\}^*$ }
  DO  $p := p \hat{.} tl$  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 ∧ List tl q Qs ∧ set Ps ∩ set Qs = {} ∧ size Qs ≤ size Ps}
  pp := p;
  WHILE q ≠ Null
  INV {∃ as bs qs.
    distinct as ∧ Path tl p as pp ∧ List tl pp bs ∧ List tl q qs ∧
    set bs ∩ set qs = {} ∧ set as ∩ (set bs ∪ set qs) = {} ∧
    size qs ≤ size bs ∧ splice Ps Qs = as @ splice bs qs}
  DO qq := q^.tl; q^.tl := pp^.tl; pp^.tl := q; pp := q^.tl; q := qq OD
  {List tl p (splice Ps Qs)}
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]: $(\{\} = insert\ a\ A \cap B) = (a \notin B \ \& \ \{\} = A \cap B)$

by blast

lemma [simp]: $(\{\} = A \cap insert\ b\ B) = (b \notin A \ \& \ \{\} = A \cap B)$

by blast

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

by blast

lemma VARS $hd\ tl\ p\ q\ r\ s$

$\{List\ tl\ p\ Ps \wedge List\ tl\ q\ Qs \wedge set\ Ps \cap set\ Qs = \{\} \wedge$
 $(p \neq Null \vee q \neq Null)\}$

IF cor $(q = Null) (cand\ (p \neq Null) (p^{\wedge}.hd \leq q^{\wedge}.hd))$

THEN $r := p; p := p^{\wedge}.tl\ ELSE\ r := q; q := q^{\wedge}.tl\ FI;$

$s := r;$

WHILE $p \neq Null \vee q \neq Null$

INV $\{\exists rs\ ps\ qs\ a.\ Path\ tl\ r\ rs\ s \wedge List\ tl\ p\ ps \wedge List\ tl\ q\ qs \wedge$
 $distinct(a \# ps @ qs @ rs) \wedge s = Ref\ a \wedge$

$merge(Ps, Qs, \lambda x\ y.\ hd\ x \leq hd\ y) =$

$rs @ a \# merge(ps, qs, \lambda x\ y.\ hd\ x \leq hd\ y) \wedge$

$(tl\ a = p \vee tl\ a = q)\}$

DO IF cor $(q = Null) (cand\ (p \neq Null) (p^{\wedge}.hd \leq q^{\wedge}.hd))$

THEN $s^{\wedge}.tl := p; p := p^{\wedge}.tl\ ELSE\ s^{\wedge}.tl := q; q := q^{\wedge}.tl\ FI;$

$s := s^{\wedge}.tl$

OD

$\{List\ tl\ r\ (merge(Ps, Qs, \lambda x\ y.\ hd\ x \leq hd\ y))\}$

apply vcg-simp

apply (simp-all add: cand-def cor-def)

apply (fastforce)

apply clarsimp

apply(rule conjI)

apply clarsimp

apply(rule conjI)

apply (fastforce intro!:Path-snoc intro:Path-upd[THEN iffD2] notin-List-update[THEN iffD2] simp:eq-sym-conv)

apply clarsimp

apply(rule conjI)

apply (clarsimp)

apply(rule-tac $x = rs @ [a]$ in exI)

apply(clarsimp simp:eq-sym-conv)

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

apply(clarsimp simp:eq-sym-conv)

apply(rule-tac $x = ya \# bsa$ in exI)

apply(simp)

apply(clarsimp simp:eq-sym-conv)

apply(rule-tac $x = rs @ [a]$ in exI)

```

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

```

```

apply(clarsimp simp add:List-app)
done

```

And now with ghost variables:

```

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

```

```

apply (fastforce)

```

```

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

```

```

apply(clarsimp simp add:List-app)
done

```

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

More of the previous proof without ghost variables can be automated, but the runtime goes up drastically. In general it is usually more efficient to give the witness directly than to have it found by proof.

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

axiomatization

ispath :: ('a ⇒ 'a ref) ⇒ 'a ref ⇒ 'a ref ⇒ bool **and**
path :: ('a ⇒ 'a ref) ⇒ 'a ref ⇒ 'a ref ⇒ 'a list

First some basic lemmas:

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

Some more specific lemmas needed by the example:

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

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

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

```

distinct(a # ps @ qs @ rs) ∧ s = Ref a ∧
merge(Ps, Qs, λx y. hd x ≤ hd y) =
rs @ a # merge(ps, qs, λx y. hd x ≤ hd y) ∧
(tl a = p ∨ tl a = q)
DO IF cor (q = Null) (cand (p ≠ Null) (p^.hd ≤ q^.hd))
THEN s^.tl := p; p := p^.tl ELSE s^.tl := q; q := q^.tl FI;
s := s^.tl
OD
{islist tl r & list tl r = (merge(Ps, Qs, λx y. hd x ≤ hd y))}
apply vcg-simp

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

```

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

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

```

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

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

If $q = \text{root}$, 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 $Ps = \text{rev } ps @ qs$. 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 $r \# \text{rev } Ps$.

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

```

lemma circular-list-rev-II:
  VARS next p q tmp
  { p = Ref r  $\wedge$  distPath next p (r#Ps) p }
  q:=Null;
  WHILE p  $\neq$  Null
  INV
  { ((q = Null)  $\longrightarrow$  ( $\exists ps$ . distPath next p (ps) (Ref r)  $\wedge$  ps = r#Ps))  $\wedge$ 
    ((q  $\neq$  Null)  $\longrightarrow$  ( $\exists ps qs$ . distPath next q (qs) (Ref r)  $\wedge$  List next p ps  $\wedge$ 
      set ps  $\cap$  set qs = {}  $\wedge$  rev qs @ ps = Ps@[r]))  $\wedge$ 
     $\neg$  (p = Null  $\wedge$  q = Null) }
  DO tmp := p; p := p^.next; tmp^.next := q; q:=tmp OD
  { q = Ref r  $\wedge$  distPath next q (r # rev Ps) q }
apply (simp only:distPath-def)
apply vcg-simp
  apply clarsimp
  apply (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 $\Rightarrow$ 'a
 where *new* A = (SOME a. a $\notin$ A)

```

lemma new-notin:
  [  $\sim$ finite(UNIV::'a set); finite(A::'a set); B  $\subseteq$  A ]  $\Longrightarrow$  new (A)  $\notin$  B
apply(unfold new-def)
apply(rule someI2-ex)
  apply (fast intro:ex-new-if-finite)
apply (fast)
done

```

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

```

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

end

```

13 Heap syntax (abort)

```

theory HeapSyntaxAbort
imports Hoare-Logic-Abort Heap
begin

```

13.1 Field access and update

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

syntax

```

-refupdate :: ('a  $\Rightarrow$  'b)  $\Rightarrow$  'a ref  $\Rightarrow$  'b  $\Rightarrow$  ('a  $\Rightarrow$  'b)
  (-/'((-  $\rightarrow$  -)') [1000,0] 900)
-fassign :: 'a ref  $\Rightarrow$  id  $\Rightarrow$  'v  $\Rightarrow$  's com
  ((2- $\wedge$ - :=/ -) [70,1000,65] 61)
-faccess :: 'a ref  $\Rightarrow$  ('a ref  $\Rightarrow$  'v)  $\Rightarrow$  'v
  (- $\wedge$ - [65,1000] 65)

```

translations

```

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

```

```

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

```

An example due to Suzuki:

```

lemma VARs v n

```

```

{w = Ref w0 & x = Ref x0 & y = Ref y0 & z = Ref z0 &

```

```

    distinct[w0,x0,y0,z0]}
  w^.v := (1::int); w^.n := x;
  x^.v := 2; x^.n := y;
  y^.v := 3; y^.n := z;
  z^.v := 4; x^.n := z
  {w^.n^.n^.v = 4}
by vcg-simp

end

```

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
 $rel :: ('a \Rightarrow 'a\ ref) \Rightarrow ('a \times 'a)\ set$
where $rel\ m = \{(x,y). m\ x = Ref\ y\}$

definition

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

definition

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

definition

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

lemmas $rel\text{-}defs = relS\text{-}def \ rel\text{-}def$

Rewrite rules for relations induced by a mapping

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

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

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

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

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

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

lemma *reachable-union*: $(\text{reachable } mS \ aS \cup \text{reachable } mS \ bS) = \text{reachable } mS \ (aS \cup bS)$
apply (*simp add: reachable-def rel-defs addrs-def*)
apply *blast*
done

lemma *reachable-union-sym*: $\text{reachable } r \ (\text{insert } a \ aS) = (r^* \text{ `` } \text{addrs } \{a\}) \cup \text{reachable } r \ aS$
apply (*simp add: reachable-def rel-defs addrs-def*)
apply *blast*
done

lemma *rel-upd1*: $(a,b) \notin \text{rel } (r(q:=t)) \implies (a,b) \in \text{rel } r \implies a=q$
apply (*rule classical*)
apply (*simp add:rel-defs fun-upd-apply*)
done

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

definition

— Restriction of a relation

restr :: $('a \times 'a) \text{ set} \Rightarrow ('a \Rightarrow \text{bool}) \Rightarrow ('a \times 'a) \text{ set}$ $((-/ \mid -) [50, 51] 50)$
where *restr* *r m* = $\{(x,y). (x,y) \in r \wedge \neg m \ x\}$

Rewrite rules for the restriction of a relation

lemma *restr-identity*[*simp*]:
 $(\forall x. \neg m \ x) \implies (R \mid m) = R$
by (*auto simp add:restr-def*)

lemma *restr-rtranc1*[*simp*]: $\llbracket m \ l \rrbracket \implies (R \mid m)^* \text{ `` } \{l\} = \{l\}$
by (*auto simp add:restr-def elim:converse-rtranc1E*)

lemma [*simp*]: $\llbracket m \ l \rrbracket \implies (l,x) \in (R \mid m)^* = (l=x)$
by (*auto simp add:restr-def elim:converse-rtranc1E*)

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

apply (*auto simp:restr-def rel-def fun-upd-apply*)
apply (*rename-tac a b*)
apply (*case-tac a=q*)
apply *auto*
done

lemma *restr-un*: $((r \cup s) \mid m) = (r \mid m) \cup (s \mid m)$
by (*auto simp add:restr-def*)

lemma *rel-upd3*: $(a, b) \notin (r|(m(q := t))) \implies (a, b) \in (r|m) \implies a = q$
apply (*rule classical*)
apply (*simp add:restr-def fun-upd-apply*)
done

definition

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

Rewrite rules for Lists using S as their mapping

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

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

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

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

primrec

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

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

where

$\text{stkOk-nil: } \text{stkOk } c \ l \ r \ iL \ iR \ t \ [] = \text{True}$
 $| \text{stkOk-cons:}$
 $\text{stkOk } c \ l \ r \ iL \ iR \ t \ (p \# \text{stk}) = (\text{stkOk } c \ l \ r \ iL \ iR \ (\text{Ref } p) \ (\text{stk}) \wedge$
 $\quad iL \ p = (\text{if } c \ p \text{ then } l \ p \text{ else } t) \wedge$
 $\quad iR \ p = (\text{if } c \ p \text{ then } t \text{ else } r \ p))$

Rewrite rules for stkOk

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

$stkOk (c(x := f)) \ l \ r \ iL \ iR \ t \ xs = stkOk \ c \ l \ r \ iL \ iR \ t \ xs$
apply (*induct xs*)
apply (*auto simp:eq-sym-conv*)
done

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

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

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

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

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

15.2 The Schorr-Waite algorithm

theorem *SchorrWaiteAlgorithm*:

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

```

      (∀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 ∨ t̂.m
  THEN IF p̂.c
    THEN q := t; t := p; p := p̂.r; t̂.r := q                — pop
    ELSE q := t; t := p̂.r; p̂.r := p̂.l;                    — swing
      p̂.l := q; p̂.c := True                                FI
    ELSE q := p; p := t; t := t̂.l; p̂.l := q;                — push
      p̂.m := True; p̂.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 addr-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 ∧ ¬ t̂.m)
  then obtain stack where inv: ?Inv stack by blast
  from a have pNull: p = Null and tDisj: t=Null ∨ (t≠Null ∧ t̂.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 addr-def stackEmpty)
  from i5 i6 show (∀x.(x ∈ R) = m x) ∧ r = iR ∧ l = iL by (auto simp:
stackEmpty fun-eq-iff intro: RisMarked)

next
  fix c m l r t p q root
  let ∃ stack. ?Inv stack = ?inv c m l r t p
  let ∃ stack. ?popInv stack = ?inv c m l (r(p → t)) p (p̂.r)
  let ∃ stack. ?swInv stack =
    ?inv (c(p → True)) m (l(p → t)) (r(p → p̂.l)) (p̂.r) p
  let ∃ stack. ?puInv stack =
    ?inv (c(t → False)) (m(t → True)) (l(t → p)) r (t̂.l) t
  let ?ifB1 = (t = Null ∨ t̂.m)
  let ?ifB2 = p̂.c

  assume (∃ stack. ?Inv stack) ∧ ?whileB m t p
  then obtain stack where inv: ?Inv stack and whileB: ?whileB m t p by
blast

```

```

let ?I1 ∧ ?I2 ∧ ?I3 ∧ ?I4 ∧ ?I5 ∧ ?I6 ∧ ?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 → (?ifB2 → (∃ stack. ?popInv stack)) ∧
      (¬?ifB2 → (∃ stack. ?swInv stack)) ) ∧
      (¬?ifB1 → (∃ stack. ?puInv stack))
proof -
{
  assume ifB1: t = Null ∨ t∧.m and ifB2: p∧.c
  from ifB1 whileB have pNotNull: p ≠ Null by auto
  then obtain addr-p where addr-p-eq: p = Ref addr-p by auto
  with i1 obtain stack-tl where stack-eq: stack = (addr p) # stack-tl
    by auto
  with i2 have m-addr-p: p∧.m by auto
  have stackDist: distinct (stack) using i1 by (rule List-distinct)
  from stack-eq stackDist have p-notin-stack-tl: addr p ∉ set stack-tl by
simp
let ?poI1 ∧ ?poI2 ∧ ?poI3 ∧ ?poI4 ∧ ?poI5 ∧ ?poI6 ∧ ?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 → t))) (p∧.r) stack-tl
    by(simp add: addr-p-eq stack-eq, simp add: S-def)

  moreover

  — Everything on the stack is marked:
  from i2 have poI2: ∀ x ∈ set stack-tl. m x by (simp add: stack-eq)
  moreover

  — Everything is still reachable:
  let (R = reachable ?Ra ?A) = ?I3
  let ?Rb = (relS {l, r(p → 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  $\forall (x, y) \in ?Rb - ?Ra. y \in (?Ra^* \text{ “ } \textit{addrs } ?A)$ 
      by (clarsimp simp: relS-def)
      (fastforce simp add:rel-def Image-iff addrs-def 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  $\forall x. x \in R \wedge \neg m\ x \longrightarrow x \in \textit{reachable } ?Ra\ ?A = ?I4$ 
let ?Rb = relS {l, r(p → t)} | m
let ?B = {p} ∪ set (map (r(p → t)) stack-tl)
— Our goal is  $\forall x. x \in R \wedge \neg m\ x \longrightarrow x \in \textit{reachable } ?Rb\ ?B$ .
let ?T = {t, p^.r}

have ?Ra* “ addrs ?A ⊆ ?Rb* “ (addrs ?B ∪ addrs ?T)
proof (rule still-reachable)
  have rewrite:  $\forall s \in \textit{set } \textit{stack-tl}. (r(p \rightarrow t))\ s = r\ s$ 
    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  $\forall (x, y) \in ?Ra - ?Rb. y \in (?Rb^* \text{ “ } (\textit{addrs } ?B \cup \textit{addrs } ?T))$ 
    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:  $\forall x. x \in R \wedge \neg m\ x \longrightarrow x \in \textit{reachable } ?Rb\ ?B$ 
proof (rule allI, rule impI)
  fix x
  assume a:  $x \in R \wedge \neg m\ x$ 
  — First, a disjunction on r (addr p) used later in the proof
  have pDisj:  $p^{\wedge}.r = \textit{Null} \vee (p^{\wedge}.r \neq \textit{Null} \wedge p^{\wedge}.r^{\wedge}.m)$  using poI1 poI2
    by auto
  — x belongs to the left hand side of subset:
  have incl:  $x \in ?Ra^* \text{ “ } \textit{addrs } ?A$  using a i4 by (simp only: reachable-def,
clarsimp)
    have excl:  $x \notin ?Rb^* \text{ “ } \textit{addrs } ?T$  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** $x \in \text{reachable } ?Rb \ ?B$ **by** (*auto simp*
add:reachable-def)
qed
moreover

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

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

moreover

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

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

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

— Everything on the stack is marked:

```

from  $i2$ 
have  $swI2$ :  $?swI2$  .
moreover

```

— Everything is still reachable:

```

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

```

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

```

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

```



```

from  $i1$   $t\text{-notin-stack}$ 
have  $puI1$ :  $?puI1$ 
  by ( $simp$   $add:addr\text{-}t\text{-}eq$   $new\text{-}stack\text{-}eq$ ,  $simp$   $add:S\text{-}def$ )
moreover

— Everything on the stack is marked:
from  $i2$ 
have  $puI2$ :  $?puI2$ 
  by ( $simp$   $add:new\text{-}stack\text{-}eq$   $fun\text{-}upd\text{-}apply$ )
moreover

— Everything is still reachable:
let  $R = reachable$   $?Ra$   $?A = ?I3$ 
let  $R = reachable$   $?Rb$   $?B = ?puI3$ 
have  $?Ra^* \text{ `` } addr s ?A = ?Rb^* \text{ `` } addr s ?B$ 
proof ( $rule$   $still\text{-}reachable\text{-}eq$ )
  show  $addr s ?A \subseteq ?Rb^* \text{ `` } addr s ?B$ 
    by ( $fastforce$   $simp:addr\text{-}def$   $rel\text{-}defs$   $addr\text{-}t\text{-}eq$   $intro:oneStep\text{-}reachable$ 
 $Image\text{-}iff[THEN\ iffD2]$ )
  next
    show  $addr s ?B \subseteq ?Ra^* \text{ `` } addr s ?A$ 
      by ( $fastforce$   $simp:addr\text{-}def$   $rel\text{-}defs$   $addr\text{-}t\text{-}eq$   $intro:oneStep\text{-}reachable$ 
 $Image\text{-}iff[THEN\ iffD2]$ )
  next
    show  $\forall (x, y) \in ?Ra - ?Rb. y \in (?Rb^* \text{ `` } addr s ?B)$ 
      by ( $clarsimp$   $simp:relS\text{-}def$ ) ( $fastforce$   $simp$   $add:rel\text{-}def$   $Image\text{-}iff$ 
 $addr\text{-}def$   $dest:rel\text{-}upd1$ )
  next
    show  $\forall (x, y) \in ?Rb - ?Ra. y \in (?Ra^* \text{ `` } addr s ?A)$ 
      by ( $clarsimp$   $simp:relS\text{-}def$ ) ( $fastforce$   $simp$   $add:rel\text{-}def$   $Image\text{-}iff$ 
 $addr\text{-}def$   $fun\text{-}upd\text{-}apply$   $dest:rel\text{-}upd2$ )
  qed
with  $i3$ 
have  $puI3$ :  $?puI3$  by ( $simp$   $add:reachable\text{-}def$ )
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  $\forall x. x \in R \wedge \neg ?new\text{-}m\ x \longrightarrow x \in reachable$   $?Rb$   $?B = ?puI4$ 
let  $?T = \{t\}$ 
have  $?Ra^* \text{ `` } addr s ?A \subseteq ?Rb^* \text{ `` } (addr s ?B \cup addr s ?T)$ 
proof ( $rule$   $still\text{-}reachable$ )
  show  $addr s ?A \subseteq ?Rb^* \text{ `` } (addr s ?B \cup addr s ?T)$ 
    by ( $fastforce$   $simp:new\text{-}stack\text{-}eq$   $addr\text{-}def$   $intro:self\text{-}reachable$ )
  next
    show  $\forall (x, y) \in ?Ra - ?Rb. y \in (?Rb^* \text{ `` } (addr s ?B \cup addr s ?T))$ 
      by ( $clarsimp$   $simp:relS\text{-}def$   $new\text{-}stack\text{-}eq$   $restr\text{-}un$   $restr\text{-}upd$ )
      ( $fastforce$   $simp$   $add:rel\text{-}def$   $Image\text{-}iff$   $restr\text{-}def$   $addr\text{-}def$   $fun\text{-}upd\text{-}apply$ 
 $addr\text{-}t\text{-}eq$   $dest:rel\text{-}upd3$ )

```

```

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:addr-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:addr-def i3 reachable-def addr-t-eq fun-upd-apply
intro:self-reachable)
moreover

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

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

ultimately show ?thesis by auto
qed
then have  $\exists\ stack. ?puInv\ stack$  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  $\Rightarrow$  nat option)
```

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

16.1 Paths in the heap

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

```
where
```

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

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

```
by (cases xs) simp-all
```

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

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

```
by (cases as) auto
```

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

```
by (induct as) auto
```

```
lemma Path-upd[simp]:
```

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

```
by (induct as) simp-all
```

16.2 Lists on the heap

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

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

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

```
by (simp add: List-def)
```

```
lemma [simp]:
```

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

```
by (simp add: List-def)
```

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

```
by (cases as) simp-all
```

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

List h a as = ($\exists b$ bs. as = a#bs $\wedge$ h a = Some b $\wedge$ List h b bs)
by (cases as) simp-all

theorem notin-List-update[simp]:
 $\bigwedge x. a \notin \text{set } as \implies \text{List } (h(a := y)) \ x \ as = \text{List } h \ x \ as$
by (induct as) simp-all

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

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

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

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

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

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

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

lemma list-ortho-sum2[simp]:
 $\bigwedge p. [\![\text{List } h2 \ p \ ps; \text{dom } h1 \cap \text{dom } h2 = \{\}]\!] \implies \text{List } (h1 ++ h2) \ p \ ps$
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 $\Rightarrow$ heap $\Rightarrow$ bool (**infix** $\perp$ 55)
where $h1 \perp h2 \iff \text{dom } h1 \cap \text{dom } h2 = \{\}$

definition *is-empty* :: *heap* $\Rightarrow$ *bool*

where *is-empty* *h* $\longleftrightarrow$ *h* = *Map.empty*

definition *singl* :: *heap* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool*

where *singl* *h* *x* *y* $\longleftrightarrow$ *dom* *h* = {*x*} & *h* *x* = *Some* *y*

definition *star* :: (*heap* $\Rightarrow$ *bool*) $\Rightarrow$ (*heap* $\Rightarrow$ *bool*) $\Rightarrow$ (*heap* $\Rightarrow$ *bool*)

where *star* *P* *Q* = ($\lambda h. \exists h1\ h2. h = h1 ++ h2 \wedge h1 \perp h2 \wedge P\ h1 \wedge Q\ h2$)

definition *wand* :: (*heap* $\Rightarrow$ *bool*) $\Rightarrow$ (*heap* $\Rightarrow$ *bool*) $\Rightarrow$ (*heap* $\Rightarrow$ *bool*)

where *wand* *P* *Q* = ($\lambda h. \forall h'. h' \perp h \wedge P\ h' \longrightarrow Q(h ++ h')$)

This is what assertions look like without any syntactic sugar:

lemma *VARs* *x* *y* *z* *w* *h*

{*star* (%*h*. *singl* *h* *x* *y*) (%*h*. *singl* *h* *z* *w*) *h*}

SKIP

{*x* $\neq$ *z*}

apply *vcg*

apply(*auto simp:star-def ortho-def singl-def*)

done

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

syntax

-*emp* :: *bool* (*emp*)

-*singl* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool* (*[-* $\mapsto$ *-]*)

-*star* :: *bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool* (**infixl** ** 60)

-*wand* :: *bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool* (**infixl** -* 60)

ML $\langle$

— *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((List.ofFree(List.of) \$ p \$ ps) /# List \$ Syntax.free H \$ p \$ ps
| *free-tr* *t* = *t*

fun *emp-tr* [] = *Syntax.const* **const-syntax** $\langle$ *is-empty* $\rangle$ \$ *Syntax.free* *H*

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

fun *singl-tr* [*p*, *q*] = *Syntax.const* **const-syntax** $\langle$ *singl* $\rangle$ \$ *Syntax.free* *H* \$ *p* \$ *q*

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

fun *star-tr* [*P*, *Q*] = *Syntax.const* **const-syntax** $\langle$ *star* $\rangle$ \$

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** $\langle$ *wand* $\rangle$ \$

```

    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,-) $ Bound 0) $ Bound 0) $ Bound 0) = t
  | strip (Abs(-,-(t as Const(-var,-) $ Var -) $ Bound 0)) = t
  | strip (Abs(-,P)) = P
  | strip (Const(const-syntax <-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
  {[ $x \mapsto y$ ] ** [ $z \mapsto w$ ]}
   $y := w$ 
  { $x \neq z$ }
apply vcg
apply(auto simp:star-def ortho-def singl-def)
done

```

```

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

```

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

```

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

```

```

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

```

```

lemma VARs H
  { $p \neq 0 \wedge [p \mapsto x] ** \text{List } H \text{ } q \text{ } qs$ }
   $H := H(p \mapsto q)$ 
  { $\text{List } H \text{ } p \text{ } (p \# qs)$ }
apply vcg
apply(simp add: star-def ortho-def singl-def)
apply clarify
apply(subgoal-tac p  $\notin$  set qs)

```

```

prefer 2
apply(blast dest:list-in-heap)
apply simp
done

lemma VARS  $H\ p\ q\ r$ 
  {List  $H\ p\ Ps\ **\ List\ H\ q\ Qs$ }
  WHILE  $p \neq 0$ 
  INV { $\exists ps\ qs. (List\ H\ p\ ps\ **\ List\ H\ q\ qs) \wedge rev\ ps\ @\ qs = rev\ Ps\ @\ Qs$ }
  DO  $r := p; p := the(H\ p); H := H(r \mapsto q); q := r$  OD
  {List  $H\ q\ (rev\ Ps\ @\ Qs)$ }
apply vcg
apply(simp-all add: star-def ortho-def singl-def)

apply fastforce

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

apply(fastforce)
done

end

```

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.