

# Hoare Logic

Norbert Galm  
Walter Guttmann  
Farhad Mehta  
Tobias Nipkow  
Leonor Prensa Nieto

January 18, 2026

## Abstract

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

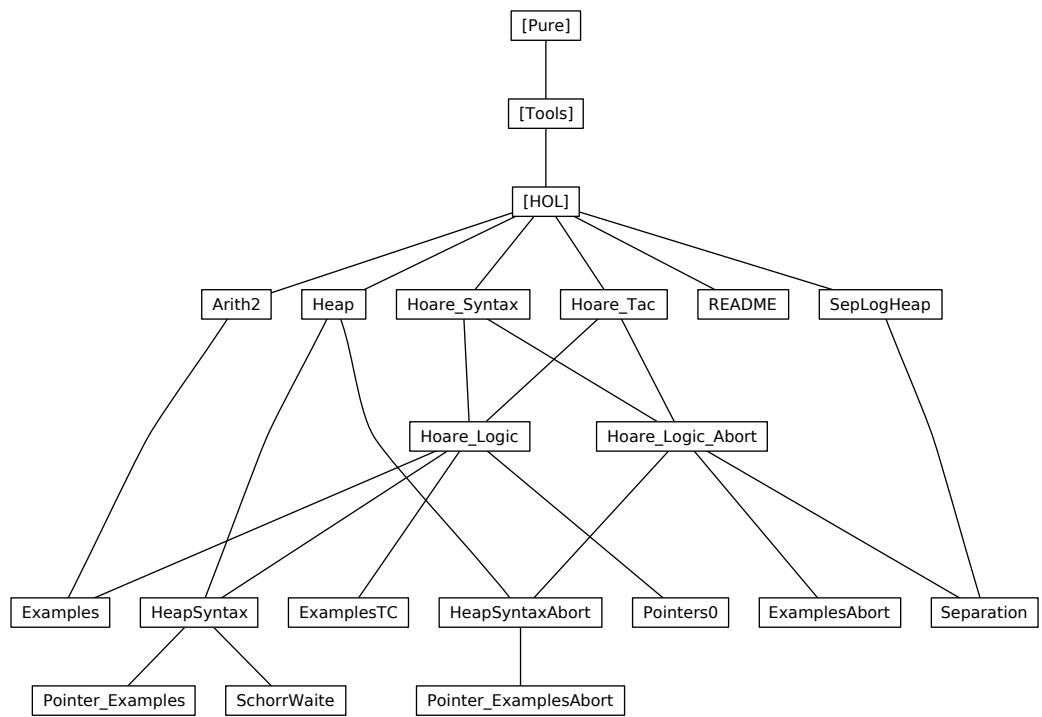
Special infrastructure for modelling and reasoning about pointer programs is provided, together with many examples, including Schorr-Waite. See [1, 2] for an excellent exposition.

## Contents

<b>1</b>	<b>Concrete syntax for Hoare logic, with translations for variables</b>	<b>5</b>
<b>2</b>	<b>Hoare logic VCG tactic</b>	<b>6</b>
<b>3</b>	<b>Hoare logic</b>	<b>7</b>
3.1	Sugared semantic embedding of Hoare logic . . . . .	7
3.1.1	Concrete syntax . . . . .	9
3.1.2	Proof methods: VCG . . . . .	9
<b>4</b>	<b>More arithmetic</b>	<b>9</b>
4.1	cd . . . . .	10
4.2	gcd . . . . .	10
4.3	pow . . . . .	10
<b>5</b>	<b>Various examples</b>	<b>11</b>
5.1	Arithmetic . . . . .	11
5.1.1	Multiplication by successive addition . . . . .	11
5.1.2	Euclid's algorithm for GCD . . . . .	12

5.1.3	Dijkstra's extension of Euclid's algorithm for simultaneous GCD and SCM . . . . .	12
5.1.4	Power by iterated squaring and multiplication . . . . .	12
5.1.5	Factorial . . . . .	13
5.1.6	Square root . . . . .	13
5.2	Lists . . . . .	14
5.3	Arrays . . . . .	15
5.3.1	Search for a key . . . . .	15
<b>6</b>	<b>Hoare Logic with an Abort statement for modelling run time errors</b>	<b>16</b>
6.1	Concrete syntax . . . . .	19
6.2	Proof methods: VCG . . . . .	19
<b>7</b>	<b>Some small examples for programs that may abort</b>	<b>19</b>
<b>8</b>	<b>Examples using Hoare Logic for Total Correctness</b>	<b>20</b>
<b>9</b>	<b>Alternative pointers</b>	<b>22</b>
9.1	References . . . . .	22
9.2	Field access and update . . . . .	22
9.3	The heap . . . . .	23
9.3.1	Paths in the heap . . . . .	23
9.3.2	Lists on the heap . . . . .	23
9.3.3	Functional abstraction . . . . .	24
9.4	Verifications . . . . .	25
9.4.1	List reversal . . . . .	25
9.4.2	Searching in a list . . . . .	25
9.4.3	Merging two lists . . . . .	26
9.4.4	Storage allocation . . . . .	27
<b>10</b>	<b>Pointers, heaps and heap abstractions</b>	<b>27</b>
10.1	References . . . . .	27
10.2	The heap . . . . .	28
10.2.1	Paths in the heap . . . . .	28
10.2.2	Non-repeating paths . . . . .	28
10.2.3	Lists on the heap . . . . .	28
10.2.4	Functional abstraction . . . . .	29
<b>11</b>	<b>Heap syntax</b>	<b>30</b>
11.1	Field access and update . . . . .	30

<b>12 Examples of verifications of pointer programs</b>	<b>31</b>
12.1 Verifications . . . . .	31
12.1.1 List reversal . . . . .	31
12.1.2 Searching in a list . . . . .	32
12.1.3 Splicing two lists . . . . .	33
12.1.4 Merging two lists . . . . .	33
12.1.5 Cyclic list reversal . . . . .	36
12.1.6 Storage allocation . . . . .	37
<b>13 Heap syntax (abort)</b>	<b>37</b>
13.1 Field access and update . . . . .	37
<b>14 Examples of verifications of pointer programs</b>	<b>38</b>
14.1 Verifications . . . . .	38
14.1.1 List reversal . . . . .	38
<b>15 Proof of the Schorr-Waite graph marking algorithm</b>	<b>39</b>
15.1 Machinery for the Schorr-Waite proof . . . . .	39
15.2 The Schorr-Waite algorithm . . . . .	42
<b>16 Heap abstractions for Separation Logic</b>	<b>42</b>
16.1 Paths in the heap . . . . .	43
16.2 Lists on the heap . . . . .	43
<b>17 Separation logic</b>	<b>44</b>



# 1 Concrete syntax for Hoare logic, with translations for variables

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

**syntax**

```
-assign :: idt ⇒ 'b ⇒ 'com
  (⟨⟨indent notation=⟨infix Hoare assignment⟩⟩- :=/ -⟩ [70, 65] 61)
-Seq :: 'com ⇒ 'com ⇒ 'com
  (⟨⟨notation=⟨infix Hoare sequential composition⟩⟩-;/ -⟩ [61, 60] 60)
-Cond :: 'bexp ⇒ 'com ⇒ 'com ⇒ 'com
  (⟨⟨notation=⟨mixfix Hoare if expression⟩⟩IF -/ THEN - / ELSE -/ FI⟩ [0, 0, 0] 61)
-While :: 'bexp ⇒ 'assn ⇒ 'var ⇒ 'com ⇒ 'com
  (⟨⟨notation=⟨mixfix Hoare while expression⟩⟩WHILE -/ INV {(-)} / VAR {(-)} //DO - /OD⟩ [0, 0, 0, 0] 61)
```

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

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

**syntax**

```
-While0 :: 'bexp ⇒ 'assn ⇒ 'com ⇒ 'com
  (⟨⟨indent=1 notation=⟨mixfix Hoare while expression⟩⟩WHILE -/ INV (⟨open-block notation=⟨mixfix Hoare invariant⟩⟩{-}) //DO - /OD⟩ [0, 0, 0] 61)
```

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

**syntax**

```
-hoare-vars :: [idts, 'assn, 'com, 'assn] ⇒ bool
  (⟨⟨open-block notation=⟨mixfix Hoare triple⟩⟩VARS -// (⟨open-block notation=⟨mixfix Hoare precondition⟩⟩{-}) // - // (⟨open-block notation=⟨mixfix Hoare postcondition⟩⟩{-})⟩ [0, 0, 55, 0] 50)
-hoare-vars-tc :: [idts, 'assn, 'com, 'assn] ⇒ bool
  (⟨⟨open-block notation=⟨mixfix Hoare triple⟩⟩VARS -// (⟨open-block notation=⟨mixfix Hoare precondition⟩⟩[-]) // - // (⟨open-block notation=⟨mixfix Hoare postcondition⟩⟩[-])⟩ [0, 0, 55, 0] 50)
```

**syntax (output)**

```
-hoare :: ['assn, 'com, 'assn] ⇒ bool
  (⟨⟨notation=⟨mixfix Hoare triple⟩⟩(⟨open-block notation=⟨mixfix Hoare precondition⟩⟩{-})// -// (⟨open-block notation=⟨mixfix Hoare postcondition⟩⟩{-})⟩ [0, 55, 0] 50)
```

```

-hoare-tc :: ['assn, 'com, 'assn] ⇒ bool
  (⟨(⟨notation=⟨mixfix Hoare triple⟩⟨⟨open-block notation=⟨mixfix Hoare precondition⟩⟨[-]⟩//[-]//⟨⟨open-block notation=⟨mixfix Hoare postcondition⟩⟨[-]⟩⟩⟩ [0, 55, 0] 50)

```

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

```

datatype 'a anno =
  Abasic |
  Aseq 'a anno 'a anno |
  Acond 'a anno 'a anno |
  Awhile 'a set 'a ⇒ nat nat ⇒ 'a anno

```

⟨*ML*⟩

**end**

## 2 Hoare logic VCG tactic

```

theory Hoare-Tac
  imports Main
begin

```

```

context
begin

```

```

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

```

```

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

```

```

lemma Compl-Collect: ¬(Collect b) = {x. ¬(b x)}
  ⟨proof⟩

```

⟨*ML*⟩

end

end

### 3 Hoare logic

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

#### 3.1 Sugared semantic embedding of Hoare logic

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

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

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

```
abbreviation annskip ( $\langle$ SKIP $\rangle$ ) 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  $\equiv \forall s s'. \text{Sem } c \ s \ s' \longrightarrow s \in p \longrightarrow s' \in q$ 
```

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

```
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**  $s1 = s2$

$\langle proof \rangle$

**lemma** *tc-implies-pc*:

$ValidTC\ p\ c\ a\ q \implies Valid\ p\ c\ a\ q$

$\langle proof \rangle$

**lemma** *tc-extract-function*:

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$(Aseq\ a1\ a2)\ R$

$\langle proof \rangle$

**lemma** *CondRule*:

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

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

$a2)\ q$

$\langle proof \rangle$

**lemma** *While-aux*:

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

**shows**  $\forall s\ s'. Sem\ c\ s\ s' \longrightarrow s \in I \wedge s \in b \longrightarrow s' \in I \implies$

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

$\langle proof \rangle$

**lemma** *WhileRule*:

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

$(Awhile\ i\ v\ A)\ q$

$\langle proof \rangle$

**lemma** *SkipRuleTC*:

**assumes**  $p \subseteq q$

**shows**  $ValidTC\ p\ (Basic\ id)\ a\ q$

$\langle proof \rangle$

**lemma** *BasicRuleTC*:

**assumes**  $p \subseteq \{s. f\ s \in q\}$



```

    shows ValidTC p (Basic f) a q
  <proof>

lemma SeqRuleTC:
  assumes ValidTC p c1 a1 q
    and ValidTC q c2 a2 r
  shows ValidTC p (Seq c1 c2) (Aseq a1 a2) r
  <proof>

lemma CondRuleTC:
  assumes p  $\subseteq$  {s. (s  $\in$  b  $\longrightarrow$  s  $\in$  w)  $\wedge$  (s  $\notin$  b  $\longrightarrow$  s  $\in$  w')}
    and ValidTC w c1 a1 q
    and ValidTC w' c2 a2 q
  shows ValidTC p (Cond b c1 c2) (Acond a1 a2) q
  <proof>

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

```

### 3.1.1 Concrete syntax

<ML>

### 3.1.2 Proof methods: VCG

```

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

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

```

<ML>

end

## 4 More arithmetic

```

theory Arith2
  imports Main

```

**begin**

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

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

**primrec**  $fac :: nat \Rightarrow nat$   
**where**  
   $fac\ 0 = Suc\ 0$   
   $| fac\ (Suc\ n) = Suc\ n * fac\ n$

#### 4.1 cd

**lemma**  $cd\text{-}nnn$ :  $0 < n \implies cd\ n\ n\ n$   
*<proof>*

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

**lemma**  $cd\text{-}swap$ :  $cd\ x\ m\ n = cd\ x\ n\ m$   
*<proof>*

**lemma**  $cd\text{-}diff\text{-}l$ :  $n \leq m \implies cd\ x\ m\ n = cd\ x\ (m - n)\ n$   
*<proof>*

**lemma**  $cd\text{-}diff\text{-}r$ :  $m \leq n \implies cd\ x\ m\ n = cd\ x\ m\ (n - m)$   
*<proof>*

#### 4.2 gcd

**lemma**  $gcd\text{-}nnn$ :  $0 < n \implies n = gcd\ n\ n$   
*<proof>*

**lemma**  $gcd\text{-}swap$ :  $gcd\ m\ n = gcd\ n\ m$   
*<proof>*

**lemma**  $gcd\text{-}diff\text{-}l$ :  $n \leq m \implies gcd\ m\ n = gcd\ (m - n)\ n$   
*<proof>*

**lemma**  $gcd\text{-}diff\text{-}r$ :  $m \leq n \implies gcd\ m\ n = gcd\ m\ (n - m)$   
*<proof>*

#### 4.3 pow

**lemma**  $sq\text{-}pow\text{-}div2$  *[simp]*:  
   $m\ mod\ 2 = 0 \implies ((n::nat)*n)^{(m\ div\ 2)} = n^m$   
*<proof>*

end

## 5 Various examples

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

### 5.1 Arithmetic

#### 5.1.1 Multiplication by successive addition

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

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

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

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

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

### 5.1.2 Euclid's algorithm for GCD

**lemma** *Euclid-GCD*: *VARs a b*  
 $\{0 < A \ \& \ 0 < B\}$   
 $a := A; \ b := B;$   
**WHILE**  $a \neq b$   
**INV**  $\{0 < a \ \& \ 0 < b \ \& \ gcd \ A \ B = gcd \ a \ b\}$   
**DO IF**  $a < b$  **THEN**  $b := b - a$  **ELSE**  $a := a - b$  **FI OD**  
 $\{a = gcd \ A \ B\}$   
 $\langle proof \rangle$

**lemma** *Euclid-GCD-time*: *VARs a b t*  
 $\{0 < A \ \& \ 0 < B \ \& \ t = 0\}$   
 $a := A; \ t := t + 1; \ b := B; \ t := t + 1;$   
**WHILE**  $a \neq b$   
**INV**  $\{0 < a \ \& \ 0 < b \ \& \ gcd \ A \ B = gcd \ a \ b \ \& \ a \leq A \ \& \ b \leq B \ \& \ t \leq max \ A \ B - max \ a \ b + 2\}$   
**DO IF**  $a < b$  **THEN**  $b := b - a; \ t := t + 1$  **ELSE**  $a := a - b; \ t := t + 1$  **FI OD**  
 $\{a = gcd \ A \ B \ \& \ t \leq max \ A \ B + 2\}$   
 $\langle proof \rangle$

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

From E.W. Disjkstra. Selected Writings on Computing, p 98 (EWD474), where it is given without the invariant. Instead of defining *scm* explicitly we have used the theorem  $scm \ x \ y = x * y / gcd \ x \ y$  and avoided division by 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**  $a \neq b$   
**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**  
 $\{a = gcd \ A \ B \ \& \ 2*A*B = a*(x+y)\}$   
 $\langle proof \rangle$

### 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**  $b \neq 0$   
**INV**  $\{A^b = c * a^b\}$

$DO \text{ WHILE } b \bmod 2 = 0$   
 $INV \{A \wedge B = c * a \wedge b\}$   
 $DO \ a := a * a; \ b := b \text{ div } 2 \ OD;$   
 $c := c * a; \ b := b - 1$   
 $OD$   
 $\{c = A \wedge B\}$   
 $\langle proof \rangle$

### 5.1.5 Factorial

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

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

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

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

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

### 5.1.6 Square root

**lemma sqrt:**  $VARs \ r \ x$

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

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

**lemma** *sqrt-without-multiplication*:  $VARs \ u \ w \ r$   
 $\{x=X\}$   
 $u := 1; \ w := 1; \ r := (0::nat);$   
 $WHILE w \leq X$   
 $INV \{u = r+r+1 \ \& \ w = (r+1)*(r+1) \ \& \ r*r \leq X\}$   
 $DO r := r + 1; \ w := w + u + 2; \ u := u + 2 OD$   
 $\{r*r \leq X \ \& \ X < (r+1)*(r+1)\}$   
 $\langle proof \rangle$

## 5.2 Lists

**lemma** *imperative-reverse*:  $VARs \ y \ x$   
 $\{x=X\}$   
 $y:=[];$   
 $WHILE x \sim= []$   
 $INV \{rev(x)@y = rev(X)\}$   
 $DO y := (hd \ x \ \# \ y); \ x := tl \ x OD$   
 $\{y=rev(X)\}$   
 $\langle proof \rangle$

**lemma** *imperative-reverse-time*:  $VARs \ y \ x \ t$   
 $\{x=X \ \& \ t=0\}$   
 $y:=[]; \ t := t+1;$   
 $WHILE x \sim= []$   
 $INV \{rev(x)@y = rev(X) \ \& \ t = 2*(length \ y) + 1\}$   
 $DO y := (hd \ x \ \# \ y); \ t := t+1; \ x := tl \ x; \ t := t+1 OD$   
 $\{y=rev(X) \ \& \ t = 2*length \ X + 1\}$   
 $\langle proof \rangle$

**lemma** *imperative-append*:  $VARs \ x \ y$   
 $\{x=X \ \& \ y=Y\}$   
 $x := rev(x);$   
 $WHILE x \sim= []$

$INV \{rev(x)@y = X@Y\}$   
 $DO y := (hd x \# y);$   
 $x := tl x$   
 $OD$   
 $\{y = X@Y\}$   
 $\langle proof \rangle$

**lemma** *imperative-append-time-no-rev*:  $VARs x y t$   
 $\{x=X \ \& \ y=Y\}$   
 $x := rev(x); t := 0;$   
 $WHILE x \neq []$   
 $INV \{rev(x)@y = X@Y \ \& \ length \ x \leq length \ X \ \& \ t = 2 * (length \ X - length \ x)\}$   
 $DO y := (hd x \# y); t := t+1;$   
 $x := tl x; t := t+1$   
 $OD$   
 $\{y = X@Y \ \& \ t = 2 * length \ X\}$   
 $\langle proof \rangle$

### 5.3 Arrays

#### 5.3.1 Search for a key

**lemma** *zero-search*:  $VARs A i$   
 $\{True\}$   
 $i := 0;$   
 $WHILE i < length \ A \ \& \ A[i] \neq key$   
 $INV \{\forall j. j < i \longrightarrow A[j] \neq key\}$   
 $DO i := i+1 \ OD$   
 $\{(i < length \ A \longrightarrow A[i] = key) \ \& \$   
 $(i = length \ A \longrightarrow (\forall j. j < length \ A \longrightarrow A[j] \neq key))\}$   
 $\langle proof \rangle$

**lemma** *zero-search-time*:  $VARs A i t$   
 $\{t=0\}$   
 $i := 0; t := t+1;$   
 $WHILE i < length \ A \ \wedge \ A[i] \neq key$   
 $INV \{(\forall j. j < i \longrightarrow A[j] \neq key) \ \wedge \ i \leq length \ A \ \wedge \ t = i+1\}$   
 $DO i := i+1; t := t+1 \ OD$   
 $\{(i < length \ A \longrightarrow A[i] = key) \ \wedge \$   
 $(i = length \ A \longrightarrow (\forall j. j < length \ A \longrightarrow A[j] \neq key)) \ \wedge \ t \leq length \ A + 1\}$   
 $\langle proof \rangle$

The *partition* procedure for quicksort.

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

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

```

lemma Partition:
  fixes pivot
  defines leq  $\equiv \lambda A\ i.\ \forall k.\ k < i \longrightarrow A!k \leq pivot$ 
    and geq  $\equiv \lambda A\ i.\ \forall k.\ i < k \wedge k < length\ A \longrightarrow pivot \leq A!k$ 
  shows
    VARs A u l
    {  $0 < length(A::('a::order)list)$  }
    l := 0; u := length A - Suc 0;
    WHILE l ≤ u
      INV { leq A l ∧ geq A u ∧ u < length A ∧ l ≤ length A }
      DO WHILE l < length A ∧ A!l ≤ pivot
        INV { leq A l & geq A u ∧ u < length A ∧ l ≤ length A }
        DO l := l+1 OD;
        WHILE 0 < u & pivot ≤ A!u
          INV { leq A l & geq A u ∧ u < length A ∧ l ≤ length A }
          DO u := u - 1 OD;
          IF l ≤ u THEN A := A[l := A!u, u := A!l] ELSE SKIP FI
        OD
    { leq A u & (∀ k. u < k ∧ k < l  $\longrightarrow$  A!k = pivot) ∧ geq A l }
  ⟨proof⟩

end

```

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

```

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

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

  datatype 'a 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  $\Rightarrow$  'a option  $\Rightarrow$  bool

  inductive Sem :: 'a com  $\Rightarrow$  'a sem
  where
    Sem (Basic f) None None

```



$| \text{Sem } (\text{Basic } f) \text{ (Some } s) \text{ (Some } (f \ s))$   
 $| \text{Sem } \text{Abort } s \text{ None}$   
 $| \text{Sem } c1 \ s \ s'' \implies \text{Sem } c2 \ s'' \ s' \implies \text{Sem } (\text{Seq } c1 \ c2) \ s \ s'$   
 $| \text{Sem } (\text{Cond } b \ c1 \ c2) \text{ None None}$   
 $| s \in b \implies \text{Sem } c1 \text{ (Some } s) \ s' \implies \text{Sem } (\text{Cond } b \ c1 \ c2) \text{ (Some } s) \ s'$   
 $| s \notin b \implies \text{Sem } c2 \text{ (Some } s) \ s' \implies \text{Sem } (\text{Cond } b \ c1 \ c2) \text{ (Some } s) \ s'$   
 $| \text{Sem } (\text{While } b \ c) \text{ None None}$   
 $| s \notin b \implies \text{Sem } (\text{While } b \ c) \text{ (Some } s) \text{ (Some } s)$   
 $| s \in b \implies \text{Sem } c \text{ (Some } s) \ s'' \implies \text{Sem } (\text{While } b \ c) \ s'' \ s' \implies$   
 $\quad \text{Sem } (\text{While } b \ c) \text{ (Some } s) \ s'$

**inductive-cases** [elim!]:

$\text{Sem } (\text{Basic } f) \ s \ s' \text{ Sem } (\text{Seq } c1 \ c2) \ s \ s'$   
 $\text{Sem } (\text{Cond } b \ c1 \ c2) \ s \ s'$

**lemma** *Sem-deterministic*:

**assumes**  $\text{Sem } c \ s \ s1$   
**and**  $\text{Sem } c \ s \ s2$   
**shows**  $s1 = s2$

$\langle \text{proof} \rangle$

**definition** *Valid* ::  $'a \text{ bexp} \Rightarrow 'a \text{ com} \Rightarrow 'a \text{ anno} \Rightarrow 'a \text{ bexp} \Rightarrow \text{bool}$

**where**  $\text{Valid } p \ c \ a \ q \equiv \forall s \ s'. \text{Sem } c \ s \ s' \longrightarrow s \in \text{Some } 'p \longrightarrow s' \in \text{Some } 'q$

**definition** *ValidTC* ::  $'a \text{ bexp} \Rightarrow 'a \text{ com} \Rightarrow 'a \text{ anno} \Rightarrow 'a \text{ bexp} \Rightarrow \text{bool}$

**where**  $\text{ValidTC } p \ c \ a \ q \equiv \forall s. s \in p \longrightarrow (\exists t. \text{Sem } c \text{ (Some } s) \text{ (Some } t) \wedge t \in q)$

**lemma** *tc-implies-pc*:

$\text{ValidTC } p \ c \ a \ q \implies \text{Valid } p \ c \ a \ q$   
 $\langle \text{proof} \rangle$

**lemma** *tc-extract-function*:

$\text{ValidTC } p \ c \ a \ q \implies \exists f. \forall s. s \in p \longrightarrow f \ s \in q$   
 $\langle \text{proof} \rangle$

The proof rules for partial correctness

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$(\text{Aseq } a1 \ a2) \ R$

$\langle \text{proof} \rangle$

**lemma** *CondRule*:

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

$\implies \text{Valid } w \text{ } c1 \text{ } a1 \text{ } q \implies \text{Valid } w' \text{ } c2 \text{ } a2 \text{ } q \implies \text{Valid } p \text{ } (Cond \text{ } b \text{ } c1 \text{ } c2) \text{ } (Acond \text{ } a1 \text{ } a2) \text{ } q$   
 $\langle proof \rangle$

**lemma** *While-aux*:

**assumes**  $Sem \text{ } (While \text{ } b \text{ } c) \text{ } s \text{ } s'$   
**shows**  $\forall s \text{ } s'. Sem \text{ } c \text{ } s \text{ } s' \longrightarrow s \in Some \text{ } ' (I \cap b) \longrightarrow s' \in Some \text{ } ' I \implies$   
 $s \in Some \text{ } ' I \implies s' \in Some \text{ } ' (I \cap -b)$   
 $\langle proof \rangle$

**lemma** *WhileRule*:

$p \subseteq i \implies \text{Valid } (i \cap b) \text{ } c \text{ } (A \text{ } 0) \text{ } i \implies i \cap (-b) \subseteq q \implies \text{Valid } p \text{ } (While \text{ } b \text{ } c)$   
 $(Awhile \text{ } i \text{ } v \text{ } A) \text{ } q$   
 $\langle proof \rangle$

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

$\langle proof \rangle$

The proof rules for total correctness

**lemma** *SkipRuleTC*:

**assumes**  $p \subseteq q$   
**shows**  $\text{ValidTC } p \text{ } (Basic \text{ } id) \text{ } a \text{ } q$   
 $\langle proof \rangle$

**lemma** *BasicRuleTC*:

**assumes**  $p \subseteq \{s. f \text{ } s \in q\}$   
**shows**  $\text{ValidTC } p \text{ } (Basic \text{ } f) \text{ } a \text{ } q$   
 $\langle proof \rangle$

**lemma** *SeqRuleTC*:

**assumes**  $\text{ValidTC } p \text{ } c1 \text{ } a1 \text{ } q$   
**and**  $\text{ValidTC } q \text{ } c2 \text{ } a2 \text{ } r$   
**shows**  $\text{ValidTC } p \text{ } (Seq \text{ } c1 \text{ } c2) \text{ } (Aseq \text{ } a1 \text{ } a2) \text{ } r$   
 $\langle proof \rangle$

**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**  $\text{ValidTC } p \text{ } (Cond \text{ } b \text{ } c1 \text{ } c2) \text{ } (Acons \text{ } a1 \text{ } a2) \text{ } q$   
 $\langle proof \rangle$

**lemma** *WhileRuleTC*:

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

## 6.1 Concrete syntax

$\langle ML \rangle$

**syntax**

-*guarded-com* :: *bool*  $\Rightarrow$  '*a com*  $\Rightarrow$  '*a com*  
 ( $\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix Hoare guarded statement} \rangle \rangle - \rightarrow / - \rangle$  71)  
 -*array-update* :: '*a list*  $\Rightarrow$  *nat*  $\Rightarrow$  '*a*  $\Rightarrow$  '*a com*  
 ( $\langle \langle \text{indent}=2 \text{ notation}=\langle \text{mixfix Hoare array update} \rangle \rangle - [-] := / - \rangle$  [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*]

$\langle ML \rangle$

**end**

## 7 Some small examples for programs that may abort

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

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

**lemma**  
*VARs* *a i j*  
 $\{k \leq \text{length } a \ \& \ i < k \ \& \ j < k\} \ j < \text{length } a \rightarrow a[i] := a!j \ \{\text{True}\}$

*<proof>*

```
lemma VARS (a::int list) i
{ True }
i := 0;
WHILE i < length a
INV { i ≤ length a }
DO a[i] := 7; i := i+1 OD
{ True }
<proof>
```

**end**

## 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 }
<proof>
```

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 ]
<proof>
```

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 = xi ∧ i ≤ n }
  DO p := p * x;
    i := i + 1
  OD
{ p = xn }
⟨proof⟩

```

Here is its total-correctness proof.

```

lemma power-tc: VARS (p::int) i
[ True ]
p := 1;
i := 0;
WHILE i < n
  INV { p = xi ∧ i ≤ n }
  VAR { n - i }
  DO p := p * x;
    i := i + 1
  OD
[ p = xn ]
⟨proof⟩

```

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 sqr-tc: VARS r
[ 0 ≤ (x::int) ]
r := 0;
WHILE (r+1)*(r+1) ≤ x
  INV { r*r ≤ x }
  VAR { nat (x-r) }
  DO r := r+1 OD
[ r*r ≤ x ∧ x < (r+1)*(r+1) ]
⟨proof⟩

```

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 sqr-t-exists:
0 ≤ (x::int) ⇒ ∃ r'. r'*r' ≤ x ∧ x < (r'+1)*(r'+1)
⟨proof⟩

```

```

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

```

```

lemma sqr-function:
assumes 0 ≤ (x::int)
and r' = sqr x
shows r'*r' ≤ x ∧ x < (r'+1)*(r'+1)
⟨proof⟩

```

Nested loops!

```

lemma VARS (i::nat) j
  [ True ]
  WHILE 0 < i
    INV { True }
    VAR { z = i }
    DO i := i - 1; j := i;
      WHILE 0 < j
        INV { z = i+1 }
        VAR { j }
        DO j := j - 1 OD
    OD
  [ i ≤ 0 ]
  ⟨proof⟩

```

end

## 9 Alternative pointers

```

theory Pointers0
imports Hoare-Logic
begin

```

### 9.1 References

```

class ref =
  fixes Null :: 'a

```

### 9.2 Field access and update

```

syntax
  -fassign :: 'a::ref => id => 'v => 's com
    (⟨⟨indent=2 notation=⟨mixfix Hoare field assignment⟩⟩-^.- := / -)⟩ [70,1000,65]
61)
  -faccess :: 'a::ref => ('a::ref ⇒ 'v) => 'v
    (⟨⟨open-block notation=⟨mixfix Hoare field access⟩⟩-^.-)⟩ [65,1000] 65)
translations
  p^.f := e => f := CONST fun-upd f p e
  p^.f      => f p

```

An example due to Suzuki:

```

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

```

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

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

*<proof>*

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

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

*<proof>*

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

*<proof>*

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

*<proof>*

### 9.3.2 Lists on the heap

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

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

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

*<proof>*

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

*<proof>*

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

*<proof>*

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

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

*<proof>*

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

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

*<proof>*

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

**lemma** *List-unique*:  $\bigwedge x \ bs. \text{List } h \ x \ as \implies \text{List } h \ x \ bs \implies as = bs$

*<proof>*

**lemma** *List-unique1*:  $List\ h\ p\ as \implies \exists! as. List\ h\ p\ as$   
 $\langle proof \rangle$

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

**lemma** *List-hd-not-in-tl[simp]*:  $List\ h\ (h\ a)\ as \implies a \notin set\ as$   
 $\langle proof \rangle$

**lemma** *List-distinct[simp]*:  $\bigwedge x. List\ h\ x\ as \implies distinct\ as$   
 $\langle proof \rangle$

### 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*:  $List\ h\ p\ as = (islist\ h\ p \wedge as = list\ h\ p)$   
 $\langle proof \rangle$

**lemma** *[simp]*:  $islist\ h\ Null$   
 $\langle proof \rangle$

**lemma** *[simp]*:  $a \neq Null \implies islist\ h\ a = islist\ h\ (h\ a)$   
 $\langle proof \rangle$

**lemma** *[simp]*:  $list\ h\ Null = []$   
 $\langle proof \rangle$

**lemma** *list-Ref-conv[simp]*:  
 $\llbracket a \neq Null; islist\ h\ (h\ a) \rrbracket \implies list\ h\ a = a \# list\ h\ (h\ a)$   
 $\langle proof \rangle$

**lemma** *[simp]*:  $islist\ h\ (h\ a) \implies a \notin set(list\ h\ (h\ a))$   
 $\langle proof \rangle$

**lemma** *list-upd-conv[simp]*:  
 $islist\ h\ p \implies y \notin set(list\ h\ p) \implies list\ (h(y := q))\ p = list\ h\ p$   
 $\langle proof \rangle$

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



## 9.4 Verifications

### 9.4.1 List reversal

A short but unreadable proof:

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

A longer readable version:

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

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

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

### 9.4.2 Searching in a list

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

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

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

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

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

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

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

### 9.4.3 Merging two lists

This is still a bit rough, especially the proof.

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

**lemma** *imp-disjCL*:  $(P \mid Q \longrightarrow R) = ((P \longrightarrow R) \wedge (\sim P \longrightarrow Q \longrightarrow R))$   
 $\langle proof \rangle$

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

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

```

    THEN  $s^{\wedge}.tl := p$ ;  $p := p^{\wedge}.tl$  ELSE  $s^{\wedge}.tl := q$ ;  $q := q^{\wedge}.tl$  FI;
     $s := s^{\wedge}.tl$ 
  OD
  {List  $tl$   $r$  (merge( $Ps, Qs, \lambda x y. hd\ x \leq hd\ y$ ))}
  <proof>

```

#### 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*:  
 $\llbracket \sim finite(UNIV::('a::ref)\ set); finite(A::'a\ set); B \subseteq A \rrbracket \Longrightarrow$   
 $new\ (A) \notin B \ \&\ new\ A \neq Null$   
 <proof>

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

**end**

## 10 Pointers, heaps and heap abstractions

See the paper by Mehta and Nipkow.

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

### 10.1 References

**datatype**  $'a\ ref = Null \mid Ref\ 'a$

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

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

**primrec**  $addr :: 'a\ ref \Rightarrow 'a$  **where**

$$\text{addr } (\text{Ref } a) = a$$

## 10.2 The heap

### 10.2.1 Paths in the heap

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

**lemma**  $[\text{iff}]$ :  $\text{Path } h \ \text{Null } xs \ y = (xs = [] \wedge y = \text{Null})$   
 $\langle \text{proof} \rangle$

**lemma**  $[\text{simp}]$ :  $\text{Path } h \ (\text{Ref } a) \ as \ z =$   
 $(as = [] \wedge z = \text{Ref } a \vee (\exists bs. as = a \# bs \wedge \text{Path } h \ (h \ a) \ bs \ z))$   
 $\langle \text{proof} \rangle$

**lemma**  $[\text{simp}]$ :  $\bigwedge x. \text{Path } f \ x \ (as @ bs) \ z = (\exists y. \text{Path } f \ x \ as \ y \wedge \text{Path } f \ y \ bs \ z)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{Path-upd}[\text{simp}]$ :  
 $\bigwedge x. u \notin \text{set } as \implies \text{Path } (f(u := v)) \ x \ as \ y = \text{Path } f \ x \ as \ y$   
 $\langle \text{proof} \rangle$

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

### 10.2.2 Non-repeating paths

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

The term  $\text{distPath } h \ x \ as \ y$  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**  $\text{neq-dP}$ :  $p \neq q \implies \text{Path } h \ p \ Ps \ q \implies \text{distinct } Ps \implies$   
 $\exists a \ Qs. p = \text{Ref } a \wedge Ps = a \# Qs \wedge a \notin \text{set } Qs$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{neq-dP-disp}$ :  $\llbracket p \neq q; \text{distPath } h \ p \ Ps \ q \rrbracket \implies$   
 $\exists a \ Qs. p = \text{Ref } a \wedge Ps = a \# Qs \wedge a \notin \text{set } Qs$   
 $\langle \text{proof} \rangle$

### 10.2.3 Lists on the heap

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

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

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

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

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

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

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

**lemma**  $List-unique$ :  $\bigwedge x\ bs.\ List\ h\ x\ as \implies List\ h\ x\ bs \implies as = bs$   
 $\langle proof \rangle$

**lemma**  $List-unique1$ :  $List\ h\ p\ as \implies \exists! as.\ List\ h\ p\ as$   
 $\langle proof \rangle$

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

**lemma**  $List-hd-not-in-tl[simp]$ :  $List\ h\ (h\ a)\ as \implies a \notin set\ as$   
 $\langle proof \rangle$

**lemma**  $List-distinct[simp]$ :  $\bigwedge x.\ List\ h\ x\ as \implies distinct\ as$   
 $\langle proof \rangle$

**lemma**  $Path-is-List$ :  
 $\llbracket Path\ h\ b\ Ps\ (Ref\ a); a \notin set\ Ps \rrbracket \implies List\ (h(a := Null))\ b\ (Ps\ @\ [a])$   
 $\langle proof \rangle$

#### 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**  $List-conv-islist-list$ :  $List\ h\ p\ as = (islist\ h\ p \wedge as = list\ h\ p)$   
 $\langle proof \rangle$

**lemma**  $[simp]$ :  $islist\ h\ Null$

⟨proof⟩

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

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

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

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

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

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

end

## 11 Heap syntax

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

### 11.1 Field access and update

**syntax**

-refupdate ::  $('a \Rightarrow 'b) \Rightarrow 'a\ ref \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b)$   
   $\langle \langle \langle open-block\ notation = \langle mixfix\ Hoare\ ref\ update \rangle \rangle - / '((- \rightarrow -)') \rangle \rangle [1000, 0]\ 900)$   
-fassign ::  $'a\ ref \Rightarrow id \Rightarrow 'v \Rightarrow 's\ com$   
   $\langle \langle \langle indent = 2\ notation = \langle mixfix\ Hoare\ ref\ assignment \rangle \rangle - \hat{.} - := / - \rangle \rangle [70, 1000, 65]$   
61)  
-faccess ::  $'a\ ref \Rightarrow ('a\ ref \Rightarrow 'v) \Rightarrow 'v$   
   $\langle \langle \langle open-block\ notation = \langle infix\ Hoare\ ref\ access \rangle \rangle - \hat{.} - \rangle \rangle [65, 1000]\ 65)$

**translations**

$f(r \rightarrow v) == f(CONST\ addr\ r := v)$   
 $p \hat{.} f := e \Rightarrow f := f(p \rightarrow e)$   
 $p \hat{.} f \Rightarrow f(CONST\ addr\ p)$

**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}
⟨proof⟩

end

```

## 12 Examples of verifications of pointer programs

```

theory Pointer-Examples
  imports HeapSyntax
begin

```

**axiomatization** where *unproven*: *PROP* *A*

### 12.1 Verifications

#### 12.1.1 List reversal

A short but unreadable proof:

```

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

```

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

```

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

```

A longer readable version:

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

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

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

### 12.1.2 Searching in a list

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

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

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

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

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

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



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

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

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

### 12.1.3 Splicing two lists

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

### 12.1.4 Merging two lists

This is still a bit rough, especially the proof.

**definition** *cor* :: *bool*  $\Rightarrow$  *bool*  $\Rightarrow$  *bool*  
**where** *cor* *P* *Q*  $\longleftrightarrow (if\ P\ then\ True\ else\ Q)$

**definition** *cand* :: *bool*  $\Rightarrow$  *bool*  $\Rightarrow$  *bool*  
**where** *cand* *P* *Q*  $\longleftrightarrow (if\ P\ then\ Q\ else\ False)$

**fun** *merge* :: *'a* *list* \* *'a* *list* \* (*'a*  $\Rightarrow$  *'a*  $\Rightarrow$  *bool*)  $\Rightarrow$  *'a* *list*

**where**

$merge(x\#xs, y\#ys, f) = (if\ f\ x\ y\ then\ x\ \# \ merge(xs, y\#ys, f)$   
 $\quad \quad \quad else\ y\ \# \ merge(x\#xs, ys, f))$   
 $| \ merge(x\#xs, [], f) = x\ \# \ merge(xs, [], f)$   
 $| \ merge([], y\#ys, f) = y\ \# \ merge([], ys, f)$   
 $| \ merge([], [], f) = []$

Simplifies the proof a little:

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

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

And now with ghost variables:

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

The proof is a LOT simpler because it does not need instantiations any-more, 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*

⟨*proof*⟩

**lemma** [*simp*]: *path* *f* *p* *p* = []

⟨*proof*⟩

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

⟨*proof*⟩

**lemma** [*simp*]: *ispath* *f* *p* *q* ⇒ *a* ∉ set(*path* *f* *p* *q*) ⇒

*path* (*f* (*a* := *r*)) *p* *q* = *path* *f* *p* *q*

⟨*proof*⟩

Some more specific lemmas needed by the example:

**lemma** [*simp*]: *ispath* (*f* (*a* := *q*)) *p* (*Ref* *a*) ⇒ *ispath* (*f* (*a* := *q*)) *p* *q*

⟨*proof*⟩

**lemma** [*simp*]: *ispath* (*f* (*a* := *q*)) *p* (*Ref* *a*) ⇒

*path* (*f* (*a* := *q*)) *p* *q* = *path* (*f* (*a* := *q*)) *p* (*Ref* *a*) @ [*a*]

⟨*proof*⟩

**lemma** [*simp*]: *ispath* *f* *p* (*Ref* *a*) ⇒ *f* *a* = *Ref* *b* ⇒

*b* ∉ set (*path* *f* *p* (*Ref* *a*))

⟨*proof*⟩

**lemma** [*simp*]: *ispath* *f* *p* (*Ref* *a*) ⇒ *f* *a* = *Null* ⇒ *islist* *f* *p*

⟨*proof*⟩

**lemma** [*simp*]: *ispath* *f* *p* (*Ref* *a*) ⇒ *f* *a* = *Null* ⇒ *list* *f* *p* = *path* *f* *p* (*Ref* *a*) @

[*a*]

⟨*proof*⟩

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

⟨*proof*⟩

**lemma** *VARs* *hd* *tl* *p* *q* *r* *s*

{ *islist* *tl* *p* ∧ *Ps* = *list* *tl* *p* ∧ *islist* *tl* *q* ∧ *Qs* = *list* *tl* *q* ∧

set *Ps* ∩ set *Qs* = {} ∧

(*p* ≠ *Null* ∨ *q* ≠ *Null*) }

*IF* *cor* (*q* = *Null*) (*cand* (*p* ≠ *Null*) (*p*^*hd* ≤ *q*^*hd*))

*THEN* *r* := *p*; *p* := *p*^*tl* *ELSE* *r* := *q*; *q* := *q*^*tl* *FI*;

*s* := *r*;

*WHILE* *p* ≠ *Null* ∨ *q* ≠ *Null*

*INV* { ∃ *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))}
⟨proof⟩

```

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 }
⟨proof⟩

```

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

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

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

**lemma** *circular-list-rev-II*:

```

  VARS next p q tmp
  {p = Ref r ∧ distPath next p (r # Ps) p}

```

```

q:=Null;
WHILE p ≠ Null
INV
{ ((q = Null) → (∃ ps. distPath next p (ps) (Ref r) ∧ ps = r#Ps)) ∧
  ((q ≠ Null) → (∃ ps qs. distPath next q (qs) (Ref r) ∧ List next p ps ∧
    set ps ∩ set qs = {} ∧ rev qs @ ps = Ps@[r])) ∧
  ¬ (p = Null ∧ q = Null) }
DO tmp := p; p := p^.next; tmp^.next := q; q:=tmp OD
{q = Ref r ∧ distPath next q (r # rev Ps) q}
⟨proof⟩

```

### 12.1.6 Storage allocation

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

**lemma** *new-notin*:

$\llbracket \sim finite(UNIV :: 'a \text{ set}); finite(A :: 'a \text{ set}); B \subseteq A \rrbracket \Longrightarrow new(A) \notin B$   
 ⟨proof⟩

**lemma**  $\sim finite(UNIV :: 'a \text{ set}) \Longrightarrow$

```

  VARS xs elem next alloc p q
  {Xs = xs ∧ p = (Null :: 'a ref)}
  WHILE xs ≠ []
  INV {islist next p ∧ set(list next p) ⊆ set alloc ∧
    map elem (rev(list next p)) @ xs = Xs}
  DO q := Ref(new(set alloc)); alloc := (addr q)#alloc;
    q^.next := p; q^.elem := hd xs; xs := tl xs; p := q
  OD
  {islist next p ∧ map elem (rev(list next p)) = Xs}
⟨proof⟩

```

**end**

## 13 Heap syntax (abort)

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

### 13.1 Field access and update

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

free addresses around. This is needed anyway as soon as we want to reason about storage allocation/deallocation.

#### **syntax**

```
-refupdate :: ('a ⇒ 'b) ⇒ 'a ref ⇒ 'b ⇒ ('a ⇒ 'b)
  (⟨⟨open-block notation=⟨mixfix Hoare ref update⟩⟩-/((- → -)')⟩ [1000,0] 900)
-fassign :: 'a ref => id => 'v => 's com
  (⟨⟨indent=2 notation=⟨mixfix Hoare ref assignment⟩⟩-^.- :=/ -)⟩ [70,1000,65]
61)
-faccess :: 'a ref => ('a ref ⇒ 'v) => 'v
  (⟨⟨open-block notation=⟨infix Hoare ref access⟩⟩-^.-)⟩ [65,1000] 65)
```

#### **translations**

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

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

An example due to Suzuki:

**lemma** *VARs v n*

```
{w = Ref w0 & x = Ref x0 & y = Ref y0 & z = Ref z0 &
  distinct[w0,x0,y0,z0]}
w^v := (1::int); w^n := x;
x^v := 2; x^n := y;
y^v := 3; y^n := z;
z^v := 4; z^n := z
{w^n.n^n.v = 4}
⟨proof⟩
```

**end**

## **14 Examples of verifications of pointer programs**

**theory** *Pointer-ExamplesAbort*

**imports** *HeapSyntaxAbort*

**begin**

### **14.1 Verifications**

#### **14.1.1 List reversal**

Interestingly, this proof is the same as for the unguarded program:

**lemma** *VARs tl p q r*

```
{List tl p Ps ∧ List tl q Qs ∧ set Ps ∩ set Qs = {}}
WHILE p ≠ Null
INV {∃ ps qs. List tl p ps ∧ List tl q qs ∧ set ps ∩ set qs = {}} ∧
  rev ps @ qs = rev Ps @ Qs}
DO r := p; (p ≠ Null → p := p^tl); r^tl := q; q := r OD
```

$\{List\ tl\ q\ (rev\ Ps\ @\ Qs)\}$   
 $\langle proof \rangle$

end

## 15 Proof of the Schorr-Waite graph marking algorithm

**theory** *SchorrWaite*  
**imports** *HeapSyntax*  
**begin**

### 15.1 Machinery for the Schorr-Waite proof

**definition**

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

**definition**

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

**definition**

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

**definition**

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

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

Rewrite rules for relations induced by a mapping

**lemma** *self-reachable*:  $b \in B \Longrightarrow b \in R^* \text{ `` } B$   
 $\langle proof \rangle$

**lemma** *oneStep-reachable*:  $b \in R \text{ `` } B \Longrightarrow b \in R^* \text{ `` } B$   
 $\langle proof \rangle$

**lemma** *still-reachable*:  $\llbracket B \subseteq Ra^* \text{ `` } A; \forall (x,y) \in Rb-Ra. y \in (Ra^* \text{ `` } A) \rrbracket \Longrightarrow Rb^* \text{ `` } B \subseteq Ra^* \text{ `` } A$   
 $\langle proof \rangle$

**lemma** *still-reachable-eq*:  $\llbracket A \subseteq Rb^* \text{ `` } B; B \subseteq Ra^* \text{ `` } A; \forall (x,y) \in Ra-Rb. y \in (Rb^* \text{ `` } B); \forall (x,y) \in Rb-Ra. y \in (Ra^* \text{ `` } A) \rrbracket \Longrightarrow Ra^* \text{ `` } A = Rb^* \text{ `` } B$   
 $\langle proof \rangle$

**lemma** *reachable-null*: *reachable mS {Null} = {}*  
 $\langle proof \rangle$

**lemma** *reachable-empty*: *reachable mS {} = {}*  
 $\langle proof \rangle$

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

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

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

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

### definition

— Restriction of a relation

*restr :: ('a  $\times$  'a) set  $\Rightarrow$  ('a  $\Rightarrow$  bool)  $\Rightarrow$  ('a  $\times$  'a) set*  
 $\langle \langle notation = \langle \text{mixfix relation restriction} \rangle \rangle - / | - \rangle [50, 51] 50 \rangle$

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

Rewrite rules for the restriction of a relation

**lemma** *restr-identity[simp]*:  
 $(\forall x. \neg m\ x) \implies (R | m) = R$   
 $\langle proof \rangle$

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

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

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

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

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

### 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**  $[rule\text{-}format, simp]:$   
 $\forall p. a \notin \text{set } stack \longrightarrow \text{List } (S \ c \ l \ r) \ p \ stack = \text{List } (S \ (c(a:=x)) \ (l(a:=y)) \ (r(a:=z))) \ p \ stack$   
 $\langle proof \rangle$

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

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

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

#### primrec

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

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

**where**

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

|  $stkOk\text{-}cons:$

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

Rewrite rules for stkOk

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

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

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

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

**lemma**  $[simp]$ :  $\bigwedge x. x \notin \text{set } xs \implies$   
 $\text{stkOk } c \ (l(x := g)) \ r \ iL \ iR \ (\text{Ref } x) \ xs = \text{stkOk } c \ l \ r \ iL \ iR \ (\text{Ref } x) \ xs$   
 $\langle \text{proof} \rangle$

**lemma**  $[simp]$ :  $\bigwedge x. x \notin \text{set } xs \implies$   
 $\text{stkOk } (c(x := g)) \ l \ r \ iL \ iR \ (\text{Ref } x) \ xs = \text{stkOk } c \ l \ r \ iL \ iR \ (\text{Ref } x) \ xs$   
 $\langle \text{proof} \rangle$

## 15.2 The Schorr-Waite algorithm

**theorem** *SchorrWaiteAlgorithm*:

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

**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**  $[iff]$ :  $Path\ h\ 0\ xs\ y = (xs = [] \wedge y = 0)$   
 $\langle proof \rangle$

**lemma**  $[simp]$ :  $x \neq 0 \implies Path\ h\ x\ as\ z =$   
 $(as = [] \wedge z = x \vee (\exists y\ bs. as = x\#\!bs \wedge h\ x = Some\ y \ \&\ Path\ h\ y\ bs\ z))$   
 $\langle proof \rangle$

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

**lemma**  $Path\text{-}upd[simp]$ :  
 $\bigwedge x. u \notin set\ as \implies Path\ (f(u := v))\ x\ as\ y = Path\ f\ x\ as\ y$   
 $\langle proof \rangle$

## 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**  $[simp]$ :  $List\ h\ x\ [] = (x = 0)$   
 $\langle proof \rangle$

**lemma**  $[simp]$ :  
 $List\ h\ x\ (a\#\!as) = (x \neq 0 \wedge a = x \wedge (\exists y. h\ x = Some\ y \wedge List\ h\ y\ as))$   
 $\langle proof \rangle$

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

**lemma**  $List\text{-}non\text{-}null$ :  $a \neq 0 \implies$   
 $List\ h\ a\ as = (\exists b\ bs. as = a\#\!bs \wedge h\ a = Some\ b \wedge List\ h\ b\ bs)$   
 $\langle proof \rangle$

**theorem**  $notin\text{-}List\text{-}update[simp]$ :  
 $\bigwedge x. a \notin set\ as \implies List\ (h(a := y))\ x\ as = List\ h\ x\ as$   
 $\langle proof \rangle$

**lemma**  $List\text{-}unique$ :  $\bigwedge x\ bs. List\ h\ x\ as \implies List\ h\ x\ bs \implies as = bs$

$\langle proof \rangle$

**lemma** *List-unique1*:  $List\ h\ p\ as \implies \exists! as. List\ h\ p\ as$   
 $\langle proof \rangle$

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

**lemma** *List-hd-not-in-tl*[simp]:  $List\ h\ b\ as \implies h\ a = Some\ b \implies a \notin set\ as$   
 $\langle proof \rangle$

**lemma** *List-distinct*[simp]:  $\bigwedge x. List\ h\ x\ as \implies distinct\ as$   
 $\langle proof \rangle$

**lemma** *list-in-heap*:  $\bigwedge p. List\ h\ p\ ps \implies set\ ps \subseteq dom\ h$   
 $\langle proof \rangle$

**lemma** *list-ortho-sum1*[simp]:  
 $\bigwedge p. \llbracket List\ h1\ p\ ps; dom\ h1 \cap dom\ h2 = \{\} \rrbracket \implies List\ (h1++h2)\ p\ ps$   
 $\langle proof \rangle$

**lemma** *list-ortho-sum2*[simp]:  
 $\bigwedge p. \llbracket List\ h2\ p\ ps; dom\ h1 \cap dom\ h2 = \{\} \rrbracket \implies List\ (h1++h2)\ p\ ps$   
 $\langle proof \rangle$

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**  $\langle \perp \rangle$  55)  
**where**  $h1 \perp h2 \longleftrightarrow dom\ h1 \cap dom\ h2 = \{\}$

**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 ≠ z}
⟨proof⟩
```

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

```
syntax
-emp :: bool (⟨emp⟩)
-singl :: nat ⇒ nat ⇒ bool (⟨⟨open-block notation=⟨mixfix singl⟩[- ↦ -]⟩⟩)
-star :: bool ⇒ bool ⇒ bool (infixl ⟨**⟩ 60)
-wand :: bool ⇒ bool ⇒ bool (infixl ⟨-∗⟩ 60)
```

```
syntax-consts
-emp ⇐ is-empty and
-singl ⇐ singl and
-star ⇐ star and
-wand ⇐ wand
```

⟨ML⟩

Now it looks much better:

```
lemma VARS  $H\ x\ y\ z\ w$ 
{[ $x \mapsto y$ ] ** [ $z \mapsto w$ ]}
SKIP
{x ≠ z}
⟨proof⟩
```

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

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

⟨ML⟩

Now the intermediate proof states are also readable:

```
lemma VARS  $H\ x\ y\ z\ w$ 
{[ $x \mapsto y$ ] ** [ $z \mapsto w$ ]}
 $y := w$ 
{x ≠ z}
```

$\langle proof \rangle$

**lemma** *VARs*  $H\ x\ y\ z\ w$   
 $\{emp\ **\ emp\}$   
*SKIP*  
 $\{emp\}$   
 $\langle proof \rangle$

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

**lemma** *star-comm*:  $P\ **\ Q = Q\ **\ P$   
 $\langle proof \rangle$

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

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

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

**end**

## References

- [1] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, volume 2741 of *LNCS*, pages 121–135. Springer, 2003.
- [2] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005.