

## Abstract

We present the theory of Simpl, a sequential imperative programming language. We introduce its syntax, its semantics (big and small-step operational semantics) and Hoare logics for both partial as well as total correctness. We prove soundness and completeness of the Hoare logic. We integrate and automate the Hoare logic in Isabelle/HOL to obtain a practically usable verification environment for imperative programs.

Simpl is independent of a concrete programming language but expressive enough to cover all common language features: mutually recursive procedures, abrupt termination and exceptions, runtime faults, local and global variables, pointers and heap, expressions with side effects, pointers to procedures, partial application and closures, dynamic method invocation and also unbounded nondeterminism.

— **Simpl** —

A Sequential Imperative Programming Language  
Syntax, Semantics, Hoare Logics and Verification  
Environment

Norbert W. Schirmer

February 4, 2026

**Contents**

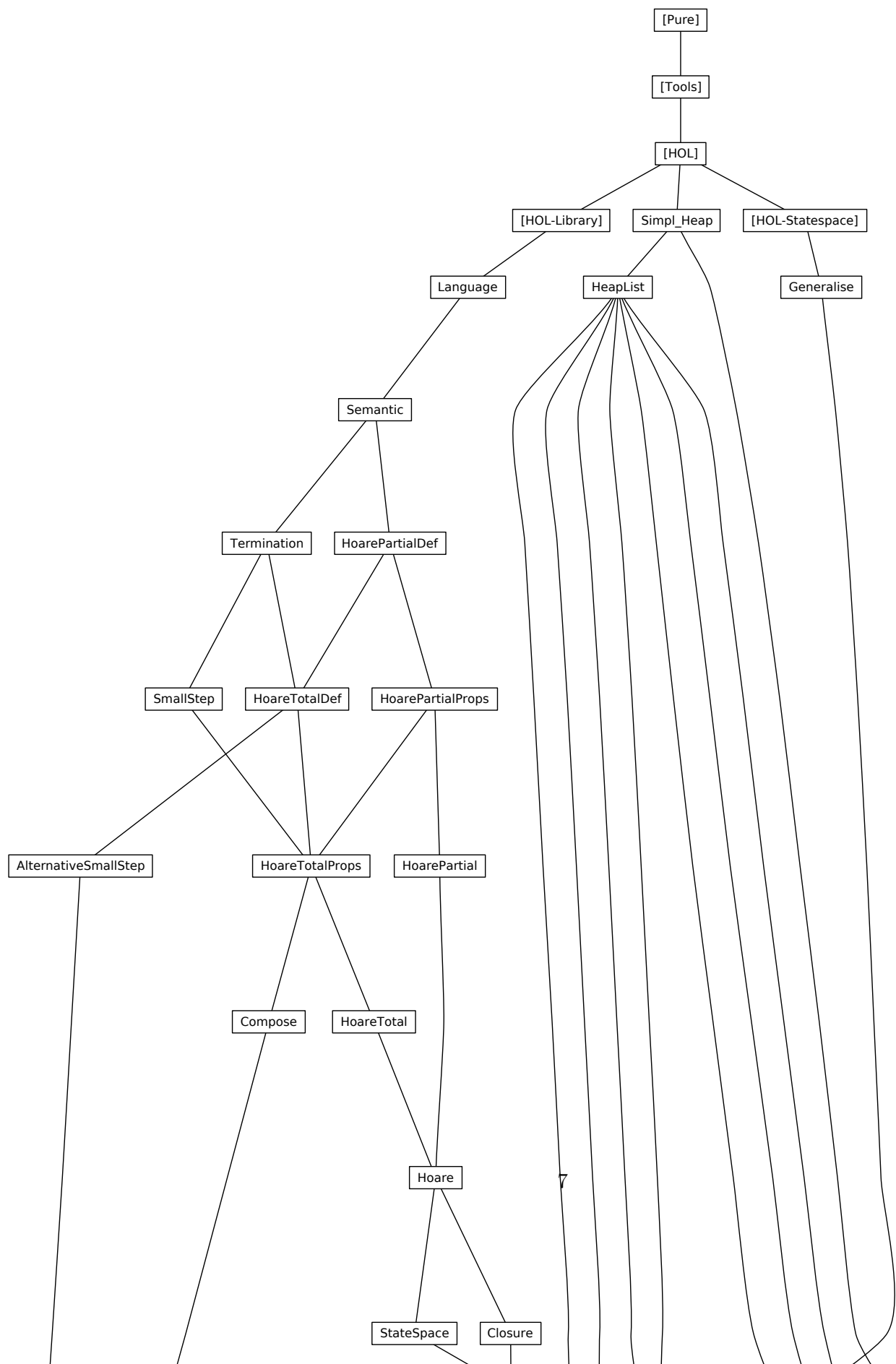
<b>1</b>	<b>Introduction</b>	<b>8</b>
<b>2</b>	<b>The Simpl Syntax</b>	<b>8</b>
2.1	The Core Language . . . . .	8
2.2	Derived Language Constructs . . . . .	8
2.3	Operations on Simpl-Syntax . . . . .	11
2.3.1	Normalisation of Sequential Composition: <i>sequence</i> , <i>flatten</i> and <i>normalize</i> . . . . .	11
2.3.2	Stripping Guards: <i>strip-guards</i> . . . . .	16
2.3.3	Marking Guards: <i>mark-guards</i> . . . . .	19
2.3.4	Merging Guards: <i>merge-guards</i> . . . . .	22
2.3.5	Intersecting Guards: $c_1 \cap_g c_2$ . . . . .	26
2.3.6	Subset on Guards: $c_1 \subseteq_g c_2$ . . . . .	29
<b>3</b>	<b>Big-Step Semantics for Simpl</b>	<b>30</b>
3.1	Big-Step Execution: $\Gamma \vdash \langle c, s \rangle \Rightarrow t$ . . . . .	31
3.2	Big-Step Execution with Recursion Limit: $\Gamma \vdash \langle c, s \rangle =_n \Rightarrow t$ . . . . .	41
3.3	Lemmas about <i>sequence</i> , <i>flatten</i> and <i>Language.normalize</i> . . . . .	55
3.4	Lemmas about $c_1 \subseteq_g c_2$ . . . . .	57
3.5	Lemmas about <i>merge-guards</i> . . . . .	57
3.6	Lemmas about <i>mark-guards</i> . . . . .	57
3.7	Lemmas about <i>strip-guards</i> . . . . .	58
3.8	Lemmas about $c_1 \cap_g c_2$ . . . . .	60
3.9	Restriction of Procedure Environment . . . . .	61
3.10	Miscellaneous . . . . .	63

<b>4</b>	<b>Hoare Logic for Partial Correctness</b>	<b>64</b>
4.1	Validity of Hoare Tuples: $\Gamma, \Theta \models_{/F} P \ c \ Q, A$ . . . . .	64
4.2	Properties of Validity . . . . .	65
4.3	The Hoare Rules: $\Gamma, \Theta \vdash_{/F} P \ c \ Q, A$ . . . . .	67
4.4	Some Derived Rules . . . . .	68
<b>5</b>	<b>Properties of Partial Correctness Hoare Logic</b>	<b>69</b>
5.1	Soundness . . . . .	69
5.2	Completeness . . . . .	69
5.3	And Now: Some Useful Rules . . . . .	72
5.3.1	Consequence . . . . .	72
5.3.2	Modify Return . . . . .	75
5.3.3	DynCall . . . . .	77
5.3.4	Conjunction of Postcondition . . . . .	79
5.3.5	Weaken Context . . . . .	80
5.3.6	Guards and Guarantees . . . . .	80
5.3.7	Restricting the Procedure Environment . . . . .	82
<b>6</b>	<b>Derived Hoare Rules for Partial Correctness</b>	<b>83</b>
6.1	Rules for Single-Step Proof . . . . .	101
<b>7</b>	<b>Terminating Programs</b>	<b>103</b>
7.1	Inductive Characterisation: $\Gamma \vdash c \downarrow s$ . . . . .	103
7.2	Lemmas about <i>sequence</i> , <i>flatten</i> and <i>Language.normalize</i> . . . . .	108
7.3	Lemmas about <i>strip-guards</i> . . . . .	109
7.4	Lemmas about $c_1 \cap_g c_2$ . . . . .	109
7.5	Lemmas about <i>mark-guards</i> . . . . .	109
7.6	Lemmas about <i>merge-guards</i> . . . . .	110
7.7	Lemmas about $c_1 \subseteq_g c_2$ . . . . .	110
7.8	Lemmas about <i>strip-guards</i> . . . . .	110
7.9	Miscellaneous . . . . .	111
<b>8</b>	<b>Small-Step Semantics and Infinite Computations</b>	<b>111</b>
8.1	Small-Step Computation: $\Gamma \vdash (c, s) \rightarrow (c', s')$ . . . . .	112
8.2	Structural Properties of Small Step Computations . . . . .	114
8.3	Equivalence between Small-Step and Big-Step Semantics . . . . .	115
8.4	Infinite Computations: $\Gamma \vdash (c, s) \rightarrow \dots(\infty)$ . . . . .	117
8.5	Equivalence between Termination and the Absence of Infinite Computations . . . . .	117
8.6	Generalised Redexes . . . . .	122
<b>9</b>	<b>Hoare Logic for Total Correctness</b>	<b>124</b>
9.1	Validity of Hoare Tuples: $\Gamma \models_{t/F} P \ c \ Q, A$ . . . . .	124
9.2	Properties of Validity . . . . .	124

9.3	The Hoare Rules: $\Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$ . . . . .	125
9.4	Some Derived Rules . . . . .	127
<b>10</b>	<b>Properties of Total Correctness Hoare Logic</b>	<b>128</b>
10.1	Soundness . . . . .	128
10.2	Completeness . . . . .	128
10.3	And Now: Some Useful Rules . . . . .	131
10.3.1	Modify Return . . . . .	131
10.3.2	DynCall . . . . .	133
10.3.3	Conjunction of Postcondition . . . . .	135
10.3.4	Guards and Guarantees . . . . .	136
10.3.5	Restricting the Procedure Environment . . . . .	138
10.3.6	Miscellaneous . . . . .	139
<b>11</b>	<b>Derived Hoare Rules for Total Correctness</b>	<b>139</b>
11.0.1	Rules for Single-Step Proof . . . . .	158
<b>12</b>	<b>Auxiliary Definitions/Lemmas to Facilitate Hoare Logic</b>	<b>159</b>
<b>13</b>	<b>State Space Template</b>	<b>166</b>
<b>14</b>	<b>Alternative Small Step Semantics</b>	<b>166</b>
14.1	Small-Step Computation: $\Gamma \vdash (cs, css, s) \rightarrow (cs', css', s')$ . . .	167
14.1.1	Structural Properties of Small Step Computations . .	169
14.1.2	Equivalence between Big and Small-Step Semantics .	172
14.2	Infinite Computations: $\text{inf } \Gamma \ cs \ css \ s$ . . . . .	173
14.3	Equivalence of Termination and Absence of Infinite Computations . . . . .	173
14.4	Completeness of Total Correctness Hoare Logic . . . . .	180
14.5	References . . . . .	182
<b>15</b>	<b>Paths and Lists in the Heap</b>	<b>182</b>
15.1	Paths in The Heap . . . . .	183
15.2	Lists on The Heap . . . . .	184
15.2.1	Relational Abstraction . . . . .	184
15.3	Functional abstraction . . . . .	185
<b>16</b>	<b>Facilitating the Hoare Logic</b>	<b>189</b>
16.1	Some Fancy Syntax . . . . .	189
<b>17</b>	<b>Examples using the Verification Environment</b>	<b>196</b>
17.1	State Spaces . . . . .	196
17.2	Basic Examples . . . . .	197
17.3	Multiplication by Addition . . . . .	199
17.4	Summing Natural Numbers . . . . .	201

17.5 SWITCH . . . . .	202
17.6 (Mutually) Recursive Procedures . . . . .	202
17.6.1 Factorial . . . . .	202
17.6.2 Odd and Even . . . . .	205
17.7 Expressions With Side Effects . . . . .	205
17.8 Global Variables and Heap . . . . .	206
17.8.1 Insertion Sort . . . . .	208
17.8.2 Memory Allocation and Deallocation . . . . .	210
17.9 Fault Avoiding Semantics . . . . .	211
17.10 Circular Lists . . . . .	214
<b>18 Examples using Statespaces</b>	<b>217</b>
18.1 State Spaces . . . . .	217
18.2 Basic Examples . . . . .	218
18.3 Multiplication by Addition . . . . .	219
18.4 Summing Natural Numbers . . . . .	221
18.5 SWITCH . . . . .	222
18.6 (Mutually) Recursive Procedures . . . . .	223
18.6.1 Factorial . . . . .	223
18.6.2 Odd and Even . . . . .	225
18.7 Expressions With Side Effects . . . . .	225
18.8 Global Variables and Heap . . . . .	226
18.8.1 Insertion Sort . . . . .	228
18.8.2 Memory Allocation and Deallocation . . . . .	230
18.9 Fault Avoiding Semantics . . . . .	231
18.10 Circular Lists . . . . .	234
<b>19 Examples for Total Correctness</b>	<b>236</b>
<b>20 Example: Quicksort on Heap Lists</b>	<b>241</b>
<b>21 Examples for Parallel Assignments</b>	<b>243</b>
<b>22 Examples for Procedures as Parameters</b>	<b>245</b>
<b>23 Examples for Procedures as Parameters using Statespaces</b>	<b>248</b>
<b>24 Experiments with Closures</b>	<b>252</b>
<b>25 Experiments on State Composition</b>	<b>257</b>
25.1 Changing the State-Space . . . . .	257
25.2 Renaming Procedures . . . . .	264

<b>26 User Guide</b>	<b>271</b>
26.1 Basics	271
26.2 Procedures	273
26.2.1 Declaration	273
26.2.2 Verification	274
26.2.3 Usage	275
26.2.4 Recursion	276
26.3 Global Variables and Heap	276
26.4 Total Correctness	279
26.5 Guards	281
26.6 Miscellaneous Techniques	282
26.6.1 Modifies Clause	282
26.6.2 Annotations	283
26.6.3 Total Correctness of Nested Loops	287
26.7 Functional Correctness, Termination and Runtime Faults	288
26.8 Procedures and Locales	289
26.9 Records	289
26.9.1 Extending State Spaces	291
26.9.2 Mapping Variables to Record Fields	291



# 1 Introduction

The work presented in these theories was developed within the German Verisoft project<sup>1</sup>. A thorough description of the core parts can be found in my PhD thesis [9]. A tutorial-like user guide is in Section 26.

Applications so far include BDD-normalisation [8], a C0 compiler [4], a page fault handler [1] and extensions towards separation logic [10].

## 2 The Simpl Syntax

**theory** *Language* **imports** *HOL-Library.Old-Recdef* **begin**

### 2.1 The Core Language

We use a shallow embedding of boolean expressions as well as assertions as sets of states.

**type-synonym** *'s bexp* = *'s set*

**type-synonym** *'s assn* = *'s set*

**datatype** (*dead 's, 'p, 'f*) *com* =  
  *Skip*  
  | *Basic 's*  $\Rightarrow$  *'s*  
  | *Spec ('s  $\times$  's)* *set*  
  | *Seq ('s, 'p, 'f) com ('s, 'p, 'f) com*  
  | *Cond 's bexp ('s, 'p, 'f) com ('s, 'p, 'f) com*  
  | *While 's bexp ('s, 'p, 'f) com*  
  | *Call 'p*  
  | *DynCom 's*  $\Rightarrow$  (*'s, 'p, 'f*) *com*  
  | *Guard 'f 's bexp ('s, 'p, 'f) com*  
  | *Throw*  
  | *Catch ('s, 'p, 'f) com ('s, 'p, 'f) com*

### 2.2 Derived Language Constructs

**definition**

*raise*:: (*'s*  $\Rightarrow$  *'s*)  $\Rightarrow$  (*'s, 'p, 'f*) *com* **where**  
*raise f* = *Seq (Basic f) Throw*

**definition**

*condCatch*:: (*'s, 'p, 'f*) *com*  $\Rightarrow$  *'s bexp*  $\Rightarrow$  (*'s, 'p, 'f*) *com*  $\Rightarrow$  (*'s, 'p, 'f*) *com* **where**  
*condCatch c<sub>1</sub> b c<sub>2</sub>* = *Catch c<sub>1</sub> (Cond b c<sub>2</sub> Throw)*

**definition**

*bind*:: (*'s*  $\Rightarrow$  *'v*)  $\Rightarrow$  (*'v*  $\Rightarrow$  (*'s, 'p, 'f*) *com*)  $\Rightarrow$  (*'s, 'p, 'f*) *com* **where**  
*bind e c* = *DynCom ( $\lambda s. c (e s)$ )*

---

<sup>1</sup><http://www.verisoft.de>

**definition**

$bseq:: ('s, 'p, 'f) com \Rightarrow ('s, 'p, 'f) com \Rightarrow ('s, 'p, 'f) com$  **where**  
 $bseq = Seq$

**definition**

$block-expr:: ['s \Rightarrow 's, ('s, 'p, 'f) com, 's \Rightarrow 's \Rightarrow 's, 's \Rightarrow 's \Rightarrow 's, 's \Rightarrow 's \Rightarrow ('s, 'p, 'f) com] \Rightarrow ('s, 'p, 'f) com$

**where**

$block-expr$  *init* *bdy* *return* *result-expr*  $c =$   
 $DynCom (\lambda s. (Seq (Catch (Seq (Basic$  *init*) *bdy*) (Seq (Basic ( $\lambda t.$  *result-expr* (return  $s$   $t$ )  $t$ )) Throw)))  
 $(DynCom (\lambda t. Seq (Basic (return  $s$ )) (c  $s$   $t$ ))))$   
 $)$

**definition**

$call-expr:: ('s \Rightarrow 's) \Rightarrow 'p \Rightarrow ('s \Rightarrow 's \Rightarrow 's) \Rightarrow ('s \Rightarrow 's \Rightarrow 's) \Rightarrow ('s \Rightarrow 's \Rightarrow ('s, 'p, 'f) com) \Rightarrow ('s, 'p, 'f) com$  **where**  
 $call-expr$  *init*  $p$  *return* *result-expr*  $c = block-expr$  *init* (Call  $p$ ) *return* *result-expr*  $c$

**primrec**  $guards:: ('f \times 's set) list \Rightarrow ('s, 'p, 'f) com \Rightarrow ('s, 'p, 'f) com$

**where**

$guards [] c = c$  |  
 $guards (g\#gs) c = Guard (fst g) (snd g) (guards gs c)$

**definition**  $maybe-guard:: 'f \Rightarrow 's set \Rightarrow ('s, 'p, 'f) com \Rightarrow ('s, 'p, 'f) com$

**where**

$maybe-guard f g c = (if g = UNIV then c else Guard f g c)$

**lemma**  $maybe-guard-UNIV$  [*simp*]:  $maybe-guard f UNIV c = c$

*<proof>*

**definition**

$dynCall-expr:: 'f \Rightarrow 's set \Rightarrow ('s \Rightarrow 's) \Rightarrow ('s \Rightarrow 'p) \Rightarrow ('s \Rightarrow 's \Rightarrow 's) \Rightarrow ('s \Rightarrow 's \Rightarrow 's) \Rightarrow ('s \Rightarrow 's \Rightarrow ('s, 'p, 'f) com) \Rightarrow ('s, 'p, 'f) com$  **where**

$dynCall-expr f g$  *init*  $p$  *return* *result-expr*  $c =$   
 $maybe-guard f g (DynCom (\lambda s. call-expr$  *init* ( $p$   $s$ ) *return* *result-expr*  $c$ ))

**definition**

$block:: ['s \Rightarrow 's, ('s, 'p, 'f) com, 's \Rightarrow 's \Rightarrow 's, 's \Rightarrow 's \Rightarrow ('s, 'p, 'f) com] \Rightarrow ('s, 'p, 'f) com$

**where**

$block$  *init* *bdy* *return*  $c = block-expr$  *init* *bdy* *return* ( $\lambda s t. s$ )  $c$

**definition**

$call:: ('s \Rightarrow 's) \Rightarrow 'p \Rightarrow ('s \Rightarrow 's \Rightarrow 's) \Rightarrow ('s \Rightarrow 's \Rightarrow ('s, 'p, 'f) com) \Rightarrow ('s, 'p, 'f) com$

**where**

$call\ init\ p\ return\ c = block\ init\ (Call\ p)\ return\ c$

**definition**

$dynCall:: ('s \Rightarrow 's) \Rightarrow ('s \Rightarrow 'p) \Rightarrow$   
 $('s \Rightarrow 's \Rightarrow 's) \Rightarrow ('s \Rightarrow 's \Rightarrow ('s, 'p, 'f)\ com) \Rightarrow ('s, 'p, 'f)\ com$  **where**  
 $dynCall\ init\ p\ return\ c = DynCom\ (\lambda s. call\ init\ (p\ s)\ return\ c)$

**definition**

$fcall:: ('s \Rightarrow 's) \Rightarrow 'p \Rightarrow ('s \Rightarrow 's \Rightarrow 's) \Rightarrow ('s \Rightarrow 'v) \Rightarrow ('v \Rightarrow ('s, 'p, 'f)\ com)$   
 $\Rightarrow ('s, 'p, 'f)\ com$  **where**  
 $fcall\ init\ p\ return\ result\ c = call\ init\ p\ return\ (\lambda s\ t. c\ (result\ t))$

**definition**

$lem:: 'x \Rightarrow ('s, 'p, 'f)\ com \Rightarrow ('s, 'p, 'f)\ com$  **where**  
 $lem\ x\ c = c$

**primrec**  $switch:: ('s \Rightarrow 'v) \Rightarrow ('v\ set \times ('s, 'p, 'f)\ com)\ list \Rightarrow ('s, 'p, 'f)\ com$

**where**

$switch\ v\ [] = Skip\ |$   
 $switch\ v\ (Vc\ \#\ vs) = Cond\ \{s. v\ s \in fst\ Vc\}\ (snd\ Vc)\ (switch\ v\ vs)$

**definition**  $guaranteeStrip:: 'f \Rightarrow 's\ set \Rightarrow ('s, 'p, 'f)\ com \Rightarrow ('s, 'p, 'f)\ com$

**where**  $guaranteeStrip\ f\ g\ c = Guard\ f\ g\ c$

**definition**  $guaranteeStripPair:: 'f \Rightarrow 's\ set \Rightarrow ('f \times 's\ set)$

**where**  $guaranteeStripPair\ f\ g = (f, g)$

**definition**

$while:: ('f \times 's\ set)\ list \Rightarrow 's\ bexp \Rightarrow ('s, 'p, 'f)\ com \Rightarrow ('s, 'p, 'f)\ com$

**where**

$while\ gs\ b\ c = guards\ gs\ (While\ b\ (Seq\ c\ (guards\ gs\ Skip)))$

**definition**

$whileAnno::$   
 $'s\ bexp \Rightarrow 's\ assn \Rightarrow ('s \times 's)\ assn \Rightarrow ('s, 'p, 'f)\ com \Rightarrow ('s, 'p, 'f)\ com$  **where**  
 $whileAnno\ b\ I\ V\ c = While\ b\ c$

**definition**

$whileAnnoG::$   
 $('f \times 's\ set)\ list \Rightarrow 's\ bexp \Rightarrow 's\ assn \Rightarrow ('s \times 's)\ assn \Rightarrow$   
 $('s, 'p, 'f)\ com \Rightarrow ('s, 'p, 'f)\ com$  **where**  
 $whileAnnoG\ gs\ b\ I\ V\ c = while\ gs\ b\ c$

**definition**

*specAnno*:: ('a ⇒ 's assn) ⇒ ('a ⇒ ('s,'p,'f) com) ⇒  
('a ⇒ 's assn) ⇒ ('a ⇒ 's assn) ⇒ ('s,'p,'f) com  
**where** *specAnno* P c Q A = (c undefined)

**definition**

*whileAnnoFix*::  
's bexp ⇒ ('a ⇒ 's assn) ⇒ ('a ⇒ ('s × 's) assn) ⇒ ('a ⇒ ('s,'p,'f) com) ⇒  
('s,'p,'f) com **where**  
*whileAnnoFix* b I V c = While b (c undefined)

**definition**

*whileAnnoGFix*::  
('f × 's set) list ⇒ 's bexp ⇒ ('a ⇒ 's assn) ⇒ ('a ⇒ ('s × 's) assn) ⇒  
('a ⇒ ('s,'p,'f) com) ⇒ ('s,'p,'f) com **where**  
*whileAnnoGFix* gs b I V c = while gs b (c undefined)

**definition** *if-rel*::('s ⇒ bool) ⇒ ('s ⇒ 's) ⇒ ('s ⇒ 's) ⇒ ('s ⇒ 's) ⇒ ('s × 's) set  
**where** *if-rel* b f g h = {(s,t). if b s then t = f s else t = g s ∨ t = h s}

**lemma** *fst-guaranteeStripPair*: fst (guaranteeStripPair f g) = f  
⟨proof⟩

**lemma** *snd-guaranteeStripPair*: snd (guaranteeStripPair f g) = g  
⟨proof⟩

**lemma** *call-call-exn*: call init p return result = call-exn init p return (λs t. s) result  
⟨proof⟩

**lemma** *dynCall-dynCall-exn*: dynCall init p return result = dynCall-exn undefined  
UNIV init p return (λs t. s) result  
⟨proof⟩

## 2.3 Operations on Simpl-Syntax

### 2.3.1 Normalisation of Sequential Composition: *sequence*, *flatten* and *normalize*

**primrec** *flatten*:: ('s,'p,'f) com ⇒ ('s,'p,'f) com list  
**where**

*flatten* Skip = [Skip] |  
*flatten* (Basic f) = [Basic f] |  
*flatten* (Spec r) = [Spec r] |  
*flatten* (Seq c<sub>1</sub> c<sub>2</sub>) = *flatten* c<sub>1</sub> @ *flatten* c<sub>2</sub> |  
*flatten* (Cond b c<sub>1</sub> c<sub>2</sub>) = [Cond b c<sub>1</sub> c<sub>2</sub>] |  
*flatten* (While b c) = [While b c] |  
*flatten* (Call p) = [Call p] |  
*flatten* (DynCom c) = [DynCom c] |  
*flatten* (Guard f g c) = [Guard f g c] |  
*flatten* Throw = [Throw] |

$flatten (Catch\ c_1\ c_2) = [Catch\ c_1\ c_2]$

**primrec** *sequence*::  $((s,p,f)\ com \Rightarrow (s,p,f)\ com \Rightarrow (s,p,f)\ com) \Rightarrow$   
 $(s,p,f)\ com\ list \Rightarrow (s,p,f)\ com$

**where**

$sequence\ seq\ [] = Skip\ |$   
 $sequence\ seq\ (c\#\ cs) = (case\ cs\ of\ [] \Rightarrow c$   
 $\quad | - \Rightarrow seq\ c\ (sequence\ seq\ cs))$

**primrec** *normalize*::  $(s,p,f)\ com \Rightarrow (s,p,f)\ com$

**where**

$normalize\ Skip = Skip\ |$   
 $normalize\ (Basic\ f) = Basic\ f\ |$   
 $normalize\ (Spec\ r) = Spec\ r\ |$   
 $normalize\ (Seq\ c_1\ c_2) = sequence\ Seq$   
 $\quad ((flatten\ (normalize\ c_1))\ @\ (flatten\ (normalize\ c_2)))\ |$   
 $normalize\ (Cond\ b\ c_1\ c_2) = Cond\ b\ (normalize\ c_1)\ (normalize\ c_2)\ |$   
 $normalize\ (While\ b\ c) = While\ b\ (normalize\ c)\ |$   
 $normalize\ (Call\ p) = Call\ p\ |$   
 $normalize\ (DynCom\ c) = DynCom\ (\lambda s. (normalize\ (c\ s)))\ |$   
 $normalize\ (Guard\ f\ g\ c) = Guard\ f\ g\ (normalize\ c)\ |$   
 $normalize\ Throw = Throw\ |$   
 $normalize\ (Catch\ c_1\ c_2) = Catch\ (normalize\ c_1)\ (normalize\ c_2)$

**lemma** *flatten-nonEmpty*:  $flatten\ c \neq []$   
*<proof>*

**lemma** *flatten-single*:  $\forall c \in set\ (flatten\ c').\ flatten\ c = [c]$   
*<proof>*

**lemma** *flatten-sequence-id*:

$[[cs \neq []]; \forall c \in set\ cs.\ flatten\ c = [c]] \Longrightarrow flatten\ (sequence\ Seq\ cs) = cs$   
*<proof>*

**lemma** *flatten-app*:

$flatten\ (sequence\ Seq\ (flatten\ c1\ @\ flatten\ c2)) = flatten\ c1\ @\ flatten\ c2$   
*<proof>*

**lemma** *flatten-sequence-flatten*:  $flatten\ (sequence\ Seq\ (flatten\ c)) = flatten\ c$   
*<proof>*

**lemma** *sequence-flatten-normalize*:  $sequence\ Seq\ (flatten\ (normalize\ c)) = normalize\ c$

*<proof>*

**lemma** *flatten-normalize*:  $\bigwedge x xs. \text{flatten} (\text{normalize } c) = x\#xs$   
 $\implies (\text{case } xs \text{ of } [] \Rightarrow \text{normalize } c = x$   
 $\quad | (x'\#xs') \Rightarrow \text{normalize } c = \text{Seq } x (\text{sequence } \text{Seq } xs))$

*<proof>*

**lemma** *flatten-raise* [*simp*]:  $\text{flatten} (\text{raise } f) = [\text{Basic } f, \text{Throw}]$   
*<proof>*

**lemma** *flatten-condCatch* [*simp*]:  $\text{flatten} (\text{condCatch } c1 \ b \ c2) = [\text{condCatch } c1 \ b \ c2]$   
*<proof>*

**lemma** *flatten-bind* [*simp*]:  $\text{flatten} (\text{bind } e \ c) = [\text{bind } e \ c]$   
*<proof>*

**lemma** *flatten-bseq* [*simp*]:  $\text{flatten} (\text{bseq } c1 \ c2) = \text{flatten } c1 \ @ \ \text{flatten } c2$   
*<proof>*

**lemma** *flatten-block-exn* [*simp*]:  
 $\text{flatten} (\text{block-exn } \text{init } \text{bdy } \text{return } \text{result-exn } \text{result}) = [\text{block-exn } \text{init } \text{bdy } \text{return } \text{result-exn } \text{result}]$   
*<proof>*

**lemma** *flatten-block* [*simp*]:  
 $\text{flatten} (\text{block } \text{init } \text{bdy } \text{return } \text{result}) = [\text{block } \text{init } \text{bdy } \text{return } \text{result}]$   
*<proof>*

**lemma** *flatten-call* [*simp*]:  $\text{flatten} (\text{call } \text{init } p \ \text{return } \text{result}) = [\text{call } \text{init } p \ \text{return } \text{result}]$   
*<proof>*

**lemma** *flatten-dynCall* [*simp*]:  $\text{flatten} (\text{dynCall } \text{init } p \ \text{return } \text{result}) = [\text{dynCall } \text{init } p \ \text{return } \text{result}]$   
*<proof>*

**lemma** *flatten-call-exn* [*simp*]:  $\text{flatten} (\text{call-exn } \text{init } p \ \text{return } \text{result-exn } \text{result}) = [\text{call-exn } \text{init } p \ \text{return } \text{result-exn } \text{result}]$   
*<proof>*

**lemma** *flatten-dynCall-exn* [*simp*]:  $\text{flatten} (\text{dynCall-exn } f \ g \ \text{init } p \ \text{return } \text{result-exn } \text{result}) = [\text{dynCall-exn } f \ g \ \text{init } p \ \text{return } \text{result-exn } \text{result}]$   
*<proof>*

**lemma** *flatten-fcall* [*simp*]:  $\text{flatten} (\text{fcall } \text{init } p \ \text{return } \text{result } c) = [\text{fcall } \text{init } p \ \text{return } \text{result } c]$   
*<proof>*

**lemma** *flatten-switch* [*simp*]:  $flatten (switch\ v\ Vcs) = [switch\ v\ Vcs]$   
 ⟨*proof*⟩

**lemma** *flatten-guaranteeStrip* [*simp*]:  
 $flatten (guaranteeStrip\ f\ g\ c) = [guaranteeStrip\ f\ g\ c]$   
 ⟨*proof*⟩

**lemma** *flatten-while* [*simp*]:  $flatten (while\ gs\ b\ c) = [while\ gs\ b\ c]$   
 ⟨*proof*⟩

**lemma** *flatten-whileAnno* [*simp*]:  
 $flatten (whileAnno\ b\ I\ V\ c) = [whileAnno\ b\ I\ V\ c]$   
 ⟨*proof*⟩

**lemma** *flatten-whileAnnoG* [*simp*]:  
 $flatten (whileAnnoG\ gs\ b\ I\ V\ c) = [whileAnnoG\ gs\ b\ I\ V\ c]$   
 ⟨*proof*⟩

**lemma** *flatten-specAnno* [*simp*]:  
 $flatten (specAnno\ P\ c\ Q\ A) = flatten (c\ undefined)$   
 ⟨*proof*⟩

**lemmas** *flatten-simps* = *flatten.simps* *flatten-raise* *flatten-condCatch* *flatten-bind*  
*flatten-block* *flatten-call* *flatten-dynCall* *flatten-fcall* *flatten-switch*  
*flatten-guaranteeStrip*  
*flatten-while* *flatten-whileAnno* *flatten-whileAnnoG* *flatten-specAnno*

**lemma** *normalize-raise* [*simp*]:  
 $normalize (raise\ f) = raise\ f$   
 ⟨*proof*⟩

**lemma** *normalize-condCatch* [*simp*]:  
 $normalize (condCatch\ c1\ b\ c2) = condCatch (normalize\ c1)\ b (normalize\ c2)$   
 ⟨*proof*⟩

**lemma** *normalize-bind* [*simp*]:  
 $normalize (bind\ e\ c) = bind\ e (\lambda v. normalize (c\ v))$   
 ⟨*proof*⟩

**lemma** *normalize-bseq* [*simp*]:  
 $normalize (bseq\ c1\ c2) = sequence\ bseq$   
 $((flatten (normalize\ c1)) @ (flatten (normalize\ c2)))$   
 ⟨*proof*⟩

**lemma** *normalize-block-exn* [*simp*]:  $normalize (block-exn\ init\ bdy\ return\ result-exn\ c) =$   
 $block-exn\ init (normalize\ bdy)\ return\ result-exn (\lambda s\ t. normalize$   
 $(c\ s\ t))$

*<proof>*

**lemma** *normalize-block* [*simp*]:  $normalize (block\ init\ bdy\ return\ c) =$   
 $block\ init\ (normalize\ bdy)\ return\ (\lambda s\ t.\ normalize\ (c\ s\ t))$   
*<proof>*

**lemma** *normalize-call* [*simp*]:  
 $normalize (call\ init\ p\ return\ c) = call\ init\ p\ return\ (\lambda i\ t.\ normalize\ (c\ i\ t))$   
*<proof>*

**lemma** *normalize-call-exn* [*simp*]:  
 $normalize (call-exn\ init\ p\ return\ result-exn\ c) = call-exn\ init\ p\ return\ result-exn$   
 $(\lambda i\ t.\ normalize\ (c\ i\ t))$   
*<proof>*

**lemma** *normalize-dynCall* [*simp*]:  
 $normalize (dynCall\ init\ p\ return\ c) =$   
 $dynCall\ init\ p\ return\ (\lambda s\ t.\ normalize\ (c\ s\ t))$   
*<proof>*

**lemma** *normalize-guards* [*simp*]:  
 $normalize (guards\ gs\ c) = guards\ gs\ (normalize\ c)$   
*<proof>*

**lemma** *normalize-dynCall-exn* [*simp*]:  
 $normalize (dynCall-exn\ f\ g\ init\ p\ return\ result-exn\ c) =$   
 $dynCall-exn\ f\ g\ init\ p\ return\ result-exn\ (\lambda s\ t.\ normalize\ (c\ s\ t))$   
*<proof>*

**lemma** *normalize-fcall* [*simp*]:  
 $normalize (fcall\ init\ p\ return\ result\ c) =$   
 $fcall\ init\ p\ return\ result\ (\lambda v.\ normalize\ (c\ v))$   
*<proof>*

**lemma** *normalize-switch* [*simp*]:  
 $normalize (switch\ v\ Vcs) = switch\ v\ (map\ (\lambda(V,c).\ (V,normalize\ c))\ Vcs)$   
*<proof>*

**lemma** *normalize-guaranteeStrip* [*simp*]:  
 $normalize (guaranteeStrip\ f\ g\ c) = guaranteeStrip\ f\ g\ (normalize\ c)$   
*<proof>*

Sequential composition with guards in the body is not preserved by normalize

**lemma** *normalize-while* [*simp*]:  
 $normalize (while\ gs\ b\ c) = guards\ gs$   
 $(While\ b\ (sequence\ Seq\ (flatten\ (normalize\ c)\ @\ flatten\ (guards\ gs\ Skip))))$   
*<proof>*

**lemma** *normalize-whileAnno* [simp]:

*normalize (whileAnno b I V c) = whileAnno b I V (normalize c)*

*<proof>*

**lemma** *normalize-whileAnnoG* [simp]:

*normalize (whileAnnoG gs b I V c) = guards gs*

*(While b (sequence Seq (flatten (normalize c) @ flatten (guards gs Skip))))*

*<proof>*

**lemma** *normalize-specAnno* [simp]:

*normalize (specAnno P c Q A) = specAnno P (λs. normalize (c undefined)) Q A*

*<proof>*

**lemmas** *normalize-simps* =

*normalize.simps normalize-raise normalize-condCatch normalize-bind*

*normalize-block normalize-call normalize-dynCall normalize-fcall normalize-switch*

*normalize-guaranteeStrip normalize-guards*

*normalize-while normalize-whileAnno normalize-whileAnnoG normalize-specAnno*

### 2.3.2 Stripping Guards: *strip-guards*

**primrec** *strip-guards*:: 'f set ⇒ ('s,'p,'f) com ⇒ ('s,'p,'f) com

**where**

*strip-guards F Skip = Skip |*

*strip-guards F (Basic f) = Basic f |*

*strip-guards F (Spec r) = Spec r |*

*strip-guards F (Seq c<sub>1</sub> c<sub>2</sub>) = (Seq (strip-guards F c<sub>1</sub>) (strip-guards F c<sub>2</sub>)) |*

*strip-guards F (Cond b c<sub>1</sub> c<sub>2</sub>) = Cond b (strip-guards F c<sub>1</sub>) (strip-guards F c<sub>2</sub>) |*

*strip-guards F (While b c) = While b (strip-guards F c) |*

*strip-guards F (Call p) = Call p |*

*strip-guards F (DynCom c) = DynCom (λs. (strip-guards F (c s))) |*

*strip-guards F (Guard f g c) = (if f ∈ F then strip-guards F c*

*else Guard f g (strip-guards F c)) |*

*strip-guards F Throw = Throw |*

*strip-guards F (Catch c<sub>1</sub> c<sub>2</sub>) = Catch (strip-guards F c<sub>1</sub>) (strip-guards F c<sub>2</sub>)*

**definition** *strip*:: 'f set ⇒

*('p ⇒ ('s,'p,'f) com option) ⇒ ('p ⇒ ('s,'p,'f) com option)*

**where** *strip F Γ = (λp. map-option (strip-guards F) (Γ p))*

**lemma** *strip-simp* [simp]: *(strip F Γ) p = map-option (strip-guards F) (Γ p)*

*<proof>*

**lemma** *dom-strip*: *dom (strip F Γ) = dom Γ*

*<proof>*

**lemma** *strip-guards-idem*: *strip-guards F (strip-guards F c) = strip-guards F c*

*<proof>*

**lemma** *strip-idem*:  $strip\ F\ (strip\ F\ \Gamma) = strip\ F\ \Gamma$   
*<proof>*

**lemma** *strip-guards-raise* [*simp*]:  
 $strip\ guards\ F\ (raise\ f) = raise\ f$   
*<proof>*

**lemma** *strip-guards-condCatch* [*simp*]:  
 $strip\ guards\ F\ (condCatch\ c1\ b\ c2) =$   
 $condCatch\ (strip\ guards\ F\ c1)\ b\ (strip\ guards\ F\ c2)$   
*<proof>*

**lemma** *strip-guards-bind* [*simp*]:  
 $strip\ guards\ F\ (bind\ e\ c) = bind\ e\ (\lambda v. strip\ guards\ F\ (c\ v))$   
*<proof>*

**lemma** *strip-guards-bseq* [*simp*]:  
 $strip\ guards\ F\ (bseq\ c1\ c2) = bseq\ (strip\ guards\ F\ c1)\ (strip\ guards\ F\ c2)$   
*<proof>*

**lemma** *strip-guards-block-exn* [*simp*]:  
 $strip\ guards\ F\ (block\ exn\ init\ bdy\ return\ result\ exn\ c) =$   
 $block\ exn\ init\ (strip\ guards\ F\ bdy)\ return\ result\ exn\ (\lambda s\ t. strip\ guards\ F\ (c\ s\ t))$   
*<proof>*

**lemma** *strip-guards-block* [*simp*]:  
 $strip\ guards\ F\ (block\ init\ bdy\ return\ c) =$   
 $block\ init\ (strip\ guards\ F\ bdy)\ return\ (\lambda s\ t. strip\ guards\ F\ (c\ s\ t))$   
*<proof>*

**lemma** *strip-guards-call* [*simp*]:  
 $strip\ guards\ F\ (call\ init\ p\ return\ c) =$   
 $call\ init\ p\ return\ (\lambda s\ t. strip\ guards\ F\ (c\ s\ t))$   
*<proof>*

**lemma** *strip-guards-call-exn* [*simp*]:  
 $strip\ guards\ F\ (call\ exn\ init\ p\ return\ result\ exn\ c) =$   
 $call\ exn\ init\ p\ return\ result\ exn\ (\lambda s\ t. strip\ guards\ F\ (c\ s\ t))$   
*<proof>*

**lemma** *strip-guards-dynCall* [*simp*]:  
 $strip\ guards\ F\ (dynCall\ init\ p\ return\ c) =$   
 $dynCall\ init\ p\ return\ (\lambda s\ t. strip\ guards\ F\ (c\ s\ t))$   
*<proof>*

**lemma** *strip-guards-guards* [*simp*]:  $strip\ guards\ F\ (guards\ gs\ c) =$   
 $guards\ (filter\ (\lambda(f,g). f\ \notin\ F)\ gs)\ (strip\ guards\ F\ c)$

$\langle \text{proof} \rangle$

**lemma** *strip-guards-dynCall-exn* [simp]:

$\text{strip-guards } F (\text{dynCall-exn } f \ g \ \text{init } p \ \text{return } \text{result-exn } c) =$   
 $\text{dynCall-exn } f \ (\text{if } f \in F \ \text{then } UNIV \ \text{else } g) \ \text{init } p \ \text{return } \text{result-exn } (\lambda s \ t.$   
 $\text{strip-guards } F \ (c \ s \ t))$   
 $\langle \text{proof} \rangle$

**lemma** *strip-guards-fcall* [simp]:

$\text{strip-guards } F (\text{fcall } \text{init } p \ \text{return } \text{result } c) =$   
 $\text{fcall } \text{init } p \ \text{return } \text{result } (\lambda v. \text{strip-guards } F \ (c \ v))$   
 $\langle \text{proof} \rangle$

**lemma** *strip-guards-switch* [simp]:

$\text{strip-guards } F (\text{switch } v \ Vc) =$   
 $\text{switch } v \ (\text{map } (\lambda (V, c). (V, \text{strip-guards } F \ c)) \ Vc)$   
 $\langle \text{proof} \rangle$

**lemma** *strip-guards-guaranteeStrip* [simp]:

$\text{strip-guards } F (\text{guaranteeStrip } f \ g \ c) =$   
 $(\text{if } f \in F \ \text{then } \text{strip-guards } F \ c$   
 $\ \text{else } \text{guaranteeStrip } f \ g \ (\text{strip-guards } F \ c))$   
 $\langle \text{proof} \rangle$

**lemma** *guaranteeStripPair-split-conv* [simp]:  $\text{case-prod } c \ (\text{guaranteeStripPair } f \ g)$   
 $= c \ f \ g$   
 $\langle \text{proof} \rangle$

**lemma** *strip-guards-while* [simp]:

$\text{strip-guards } F (\text{while } gs \ b \ c) =$   
 $\text{while } (\text{filter } (\lambda (f, g). f \notin F) \ gs) \ b \ (\text{strip-guards } F \ c)$   
 $\langle \text{proof} \rangle$

**lemma** *strip-guards-whileAnno* [simp]:

$\text{strip-guards } F (\text{whileAnno } b \ I \ V \ c) = \text{whileAnno } b \ I \ V \ (\text{strip-guards } F \ c)$   
 $\langle \text{proof} \rangle$

**lemma** *strip-guards-whileAnnoG* [simp]:

$\text{strip-guards } F (\text{whileAnnoG } gs \ b \ I \ V \ c) =$   
 $\text{whileAnnoG } (\text{filter } (\lambda (f, g). f \notin F) \ gs) \ b \ I \ V \ (\text{strip-guards } F \ c)$   
 $\langle \text{proof} \rangle$

**lemma** *strip-guards-specAnno* [simp]:

$\text{strip-guards } F (\text{specAnno } P \ c \ Q \ A) =$   
 $\text{specAnno } P \ (\lambda s. \text{strip-guards } F \ (c \ \text{undefined})) \ Q \ A$   
 $\langle \text{proof} \rangle$

**lemmas** *strip-guards-simps* = *strip-guards.simps strip-guards-raise*  
*strip-guards-condCatch strip-guards-bind strip-guards-bseq strip-guards-block*  
*strip-guards-dynCall strip-guards-fcall strip-guards-switch*  
*strip-guards-guaranteeStrip guaranteeStripPair-split-conv strip-guards-guards*  
*strip-guards-while strip-guards-whileAnno strip-guards-whileAnnoG*  
*strip-guards-specAnno*

### 2.3.3 Marking Guards: *mark-guards*

**primrec** *mark-guards*:: '*f* ⇒ ('*s*, '*p*, '*g*) *com* ⇒ ('*s*, '*p*, '*f*) *com*

**where**

*mark-guards f Skip* = *Skip* |  
*mark-guards f (Basic g)* = *Basic g* |  
*mark-guards f (Spec r)* = *Spec r* |  
*mark-guards f (Seq c<sub>1</sub> c<sub>2</sub>)* = (*Seq (mark-guards f c<sub>1</sub>) (mark-guards f c<sub>2</sub>)*) |  
*mark-guards f (Cond b c<sub>1</sub> c<sub>2</sub>)* = *Cond b (mark-guards f c<sub>1</sub>) (mark-guards f c<sub>2</sub>)* |  
*mark-guards f (While b c)* = *While b (mark-guards f c)* |  
*mark-guards f (Call p)* = *Call p* |  
*mark-guards f (DynCom c)* = *DynCom (λs. (mark-guards f (c s)))* |  
*mark-guards f (Guard f' g c)* = *Guard f g (mark-guards f c)* |  
*mark-guards f Throw* = *Throw* |  
*mark-guards f (Catch c<sub>1</sub> c<sub>2</sub>)* = *Catch (mark-guards f c<sub>1</sub>) (mark-guards f c<sub>2</sub>)*

**lemma** *mark-guards-raise*: *mark-guards f (raise g) = raise g*  
 ⟨*proof*⟩

**lemma** *mark-guards-condCatch [simp]*:  
*mark-guards f (condCatch c1 b c2) =*  
*condCatch (mark-guards f c1) b (mark-guards f c2)*  
 ⟨*proof*⟩

**lemma** *mark-guards-bind [simp]*:  
*mark-guards f (bind e c) = bind e (λv. mark-guards f (c v))*  
 ⟨*proof*⟩

**lemma** *mark-guards-bseq [simp]*:  
*mark-guards f (bseq c1 c2) = bseq (mark-guards f c1) (mark-guards f c2)*  
 ⟨*proof*⟩

**lemma** *mark-guards-block-exn [simp]*:  
*mark-guards f (block-exn init bdy return result-exn c) =*  
*block-exn init (mark-guards f bdy) return result-exn (λs t. mark-guards f (c s*  
*t))*  
 ⟨*proof*⟩

**lemma** *mark-guards-block [simp]*:  
*mark-guards f (block init bdy return c) =*  
*block init (mark-guards f bdy) return (λs t. mark-guards f (c s t))*  
 ⟨*proof*⟩

**lemma** *mark-guards-call* [*simp*]:

$mark-guards\ f\ (call\ init\ p\ return\ c) =$   
 $call\ init\ p\ return\ (\lambda s\ t.\ mark-guards\ f\ (c\ s\ t))$   
(*proof*)

**lemma** *mark-guards-call-exn* [*simp*]:

$mark-guards\ f\ (call-exn\ init\ p\ return\ result-exn\ c) =$   
 $call-exn\ init\ p\ return\ result-exn\ (\lambda s\ t.\ mark-guards\ f\ (c\ s\ t))$   
(*proof*)

**lemma** *mark-guards-dynCall* [*simp*]:

$mark-guards\ f\ (dynCall\ init\ p\ return\ c) =$   
 $dynCall\ init\ p\ return\ (\lambda s\ t.\ mark-guards\ f\ (c\ s\ t))$   
(*proof*)

**lemma** *mark-guards-guards* [*simp*]:

$mark-guards\ f\ (guards\ gs\ c) = guards\ (map\ (\lambda(f',g).\ (f,g))\ gs)\ (mark-guards\ f\ c)$   
(*proof*)

**lemma** *mark-guards-dynCall-exn* [*simp*]:

$mark-guards\ f\ (dynCall-exn\ f'\ g\ init\ p\ return\ result-exn\ c) =$   
 $dynCall-exn\ f\ g\ init\ p\ return\ result-exn\ (\lambda s\ t.\ mark-guards\ f\ (c\ s\ t))$   
(*proof*)

**lemma** *mark-guards-fcall* [*simp*]:

$mark-guards\ f\ (fcall\ init\ p\ return\ result\ c) =$   
 $fcall\ init\ p\ return\ result\ (\lambda v.\ mark-guards\ f\ (c\ v))$   
(*proof*)

**lemma** *mark-guards-switch* [*simp*]:

$mark-guards\ f\ (switch\ v\ vs) =$   
 $switch\ v\ (map\ (\lambda(V,c).\ (V,mark-guards\ f\ c))\ vs)$   
(*proof*)

**lemma** *mark-guards-guaranteeStrip* [*simp*]:

$mark-guards\ f\ (guaranteeStrip\ f'\ g\ c) = guaranteeStrip\ f\ g\ (mark-guards\ f\ c)$   
(*proof*)

**lemma** *mark-guards-while* [*simp*]:

$mark-guards\ f\ (while\ gs\ b\ c) =$   
 $while\ (map\ (\lambda(f',g).\ (f,g))\ gs)\ b\ (mark-guards\ f\ c)$   
(*proof*)

**lemma** *mark-guards-whileAnno* [*simp*]:

$mark-guards\ f\ (whileAnno\ b\ I\ V\ c) = whileAnno\ b\ I\ V\ (mark-guards\ f\ c)$   
(*proof*)

**lemma** *mark-guards-whileAnnoG* [simp]:  
*mark-guards f (whileAnnoG gs b I V c) =*  
*whileAnnoG (map (λ(f',g). (f,g)) gs) b I V (mark-guards f c)*  
 ⟨proof⟩

**lemma** *mark-guards-specAnno* [simp]:  
*mark-guards f (specAnno P c Q A) =*  
*specAnno P (λs. mark-guards f (c undefined)) Q A*  
 ⟨proof⟩

**lemmas** *mark-guards-simps = mark-guards.simps mark-guards-raise*  
*mark-guards-condCatch mark-guards-bind mark-guards-bseq mark-guards-block*  
*mark-guards-dynCall mark-guards-fcall mark-guards-switch*  
*mark-guards-guaranteeStrip guaranteeStripPair-split-conv mark-guards-guards*  
*mark-guards-while mark-guards-whileAnno mark-guards-whileAnnoG*  
*mark-guards-specAnno*

**definition** *is-Guard*:: ('s,'p,'f) com ⇒ bool  
**where** *is-Guard c = (case c of Guard f g c' ⇒ True | - ⇒ False)*

**lemma** *is-Guard-basic-simps* [simp]:  
*is-Guard (guards (pg# pgs) c) = True*  
*is-Guard Skip = False*  
*is-Guard (Basic f) = False*  
*is-Guard (Spec r) = False*  
*is-Guard (Seq c1 c2) = False*  
*is-Guard (Cond b c1 c2) = False*  
*is-Guard (While b c) = False*  
*is-Guard (Call p) = False*  
*is-Guard (DynCom C) = False*  
*is-Guard (Guard F g c) = True*  
*is-Guard (Throw) = False*  
*is-Guard (Catch c1 c2) = False*  
*is-Guard (raise f) = False*  
*is-Guard (condCatch c1 b c2) = False*  
*is-Guard (bind e cv) = False*  
*is-Guard (bseq c1 c2) = False*  
*is-Guard (block-exn init bdy return result-exn cont) = False*  
*is-Guard (block init bdy return cont) = False*  
*is-Guard (call init p return cont) = False*  
*is-Guard (dynCall init P return cont) = False*  
*is-Guard (call-exn init p return result-exn cont) = False*  
*is-Guard (dynCall-exn f UNIV init P return result-exn cont) = False*  
*is-Guard (fcall init p return result cont') = False*  
*is-Guard (whileAnno b I V c) = False*  
*is-Guard (guaranteeStrip F g c) = True*  
 ⟨proof⟩

**lemma** *is-Guard-switch* [simp]:

*is-Guard* (switch  $v$   $Vc$ ) = *False*  
 ⟨*proof*⟩

**lemmas** *is-Guard-simps* = *is-Guard-basic-simps is-Guard-switch*

**primrec** *dest-Guard*:: ('s,'p,'f) *com*  $\Rightarrow$  ('f  $\times$  's *set*  $\times$  ('s,'p,'f) *com*)  
**where** *dest-Guard* (*Guard*  $f$   $g$   $c$ ) = ( $f,g,c$ )

**lemma** *dest-Guard-guaranteeStrip* [*simp*]: *dest-Guard* (*guaranteeStrip*  $f$   $g$   $c$ ) =  
 ( $f,g,c$ )  
 ⟨*proof*⟩

**lemmas** *dest-Guard-simps* = *dest-Guard.simps dest-Guard-guaranteeStrip*

### 2.3.4 Merging Guards: *merge-guards*

**primrec** *merge-guards*:: ('s,'p,'f) *com*  $\Rightarrow$  ('s,'p,'f) *com*  
**where**

*merge-guards* *Skip* = *Skip* |  
*merge-guards* (*Basic*  $g$ ) = *Basic*  $g$  |  
*merge-guards* (*Spec*  $r$ ) = *Spec*  $r$  |  
*merge-guards* (*Seq*  $c_1$   $c_2$ ) = (*Seq* (*merge-guards*  $c_1$ ) (*merge-guards*  $c_2$ )) |  
*merge-guards* (*Cond*  $b$   $c_1$   $c_2$ ) = *Cond*  $b$  (*merge-guards*  $c_1$ ) (*merge-guards*  $c_2$ ) |  
*merge-guards* (*While*  $b$   $c$ ) = *While*  $b$  (*merge-guards*  $c$ ) |  
*merge-guards* (*Call*  $p$ ) = *Call*  $p$  |  
*merge-guards* (*DynCom*  $c$ ) = *DynCom* ( $\lambda s. (*merge-guards* (c s))$ ) |

*merge-guards* (*Guard*  $f$   $g$   $c$ ) =  
 (let  $c' = (*merge-guards* c)$   
 in if *is-Guard*  $c'$   
 then let ( $f',g',c''$ ) = *dest-Guard*  $c'$   
   in if  $f=f'$  then *Guard*  $f$  ( $g \cap g'$ )  $c''$   
   else *Guard*  $f$   $g$  (*Guard*  $f'$   $g'$   $c''$ )  
 else *Guard*  $f$   $g$   $c'$ ) |  
*merge-guards* *Throw* = *Throw* |  
*merge-guards* (*Catch*  $c_1$   $c_2$ ) = *Catch* (*merge-guards*  $c_1$ ) (*merge-guards*  $c_2$ )

**lemma** *merge-guards-res-Skip*: *merge-guards*  $c = *Skip*$   $\Longrightarrow$   $c = *Skip*$   
 ⟨*proof*⟩

**lemma** *merge-guards-res-Basic*: *merge-guards*  $c = *Basic* f$   $\Longrightarrow$   $c = *Basic* f$   
 ⟨*proof*⟩

**lemma** *merge-guards-res-Spec*: *merge-guards*  $c = *Spec* r$   $\Longrightarrow$   $c = *Spec* r$   
 ⟨*proof*⟩

**lemma** *merge-guards-res-Seq*: *merge-guards*  $c = *Seq* c1 c2$   $\Longrightarrow$   
 $\exists c1' c2'. c = *Seq* c1' c2' \wedge *merge-guards* c1' = c1 \wedge *merge-guards* c2' = c2$

*<proof>*

**lemma** *merge-guards-res-Cond*:  $\text{merge-guards } c = \text{Cond } b \ c1 \ c2 \implies$   
 $\exists c1' \ c2'. \ c = \text{Cond } b \ c1' \ c2' \wedge \text{merge-guards } c1' = c1 \wedge \text{merge-guards } c2' =$   
 $c2$   
*<proof>*

**lemma** *merge-guards-res-While*:  $\text{merge-guards } c = \text{While } b \ c' \implies$   
 $\exists c''. \ c = \text{While } b \ c'' \wedge \text{merge-guards } c'' = c'$   
*<proof>*

**lemma** *merge-guards-res-Call*:  $\text{merge-guards } c = \text{Call } p \implies c = \text{Call } p$   
*<proof>*

**lemma** *merge-guards-res-DynCom*:  $\text{merge-guards } c = \text{DynCom } c' \implies$   
 $\exists c''. \ c = \text{DynCom } c'' \wedge (\lambda s. (\text{merge-guards } (c'' \ s))) = c'$   
*<proof>*

**lemma** *merge-guards-res-Throw*:  $\text{merge-guards } c = \text{Throw} \implies c = \text{Throw}$   
*<proof>*

**lemma** *merge-guards-res-Catch*:  $\text{merge-guards } c = \text{Catch } c1 \ c2 \implies$   
 $\exists c1' \ c2'. \ c = \text{Catch } c1' \ c2' \wedge \text{merge-guards } c1' = c1 \wedge \text{merge-guards } c2' = c2$   
*<proof>*

**lemma** *merge-guards-res-Guard*:  
 $\text{merge-guards } c = \text{Guard } f \ g \ c' \implies \exists c'' \ f' \ g'. \ c = \text{Guard } f' \ g' \ c''$   
*<proof>*

**lemmas** *merge-guards-res-simps = merge-guards-res-Skip merge-guards-res-Basic*  
*merge-guards-res-Spec merge-guards-res-Seq merge-guards-res-Cond*  
*merge-guards-res-While merge-guards-res-Call*  
*merge-guards-res-DynCom merge-guards-res-Throw merge-guards-res-Catch*  
*merge-guards-res-Guard*

**lemma** *merge-guards-guards-empty*:  $\text{merge-guards } (\text{guards } [] \ c) = \text{merge-guards } c$   
*<proof>*

**lemma** *merge-guards-raise*:  $\text{merge-guards } (\text{raise } g) = \text{raise } g$   
*<proof>*

**lemma** *merge-guards-condCatch [simp]*:  
 $\text{merge-guards } (\text{condCatch } c1 \ b \ c2) =$   
 $\text{condCatch } (\text{merge-guards } c1) \ b \ (\text{merge-guards } c2)$   
*<proof>*

**lemma** *merge-guards-bind [simp]*:  
 $\text{merge-guards } (\text{bind } e \ c) = \text{bind } e \ (\lambda v. \text{merge-guards } (c \ v))$   
*<proof>*

**lemma** *merge-guards-bseq* [*simp*]:

*merge-guards* (*bseq* *c1* *c2*) = *bseq* (*merge-guards* *c1*) (*merge-guards* *c2*)  
(*proof*)

**lemma** *merge-guards-block-exn* [*simp*]:

*merge-guards* (*block-exn* *init* *bdy* *return* *result-exn* *c*) =  
*block-exn* *init* (*merge-guards* *bdy*) *return* *result-exn* ( $\lambda s t. \text{merge-guards } (c\ s\ t)$ )  
(*proof*)

**lemma** *merge-guards-block* [*simp*]:

*merge-guards* (*block* *init* *bdy* *return* *c*) =  
*block* *init* (*merge-guards* *bdy*) *return* ( $\lambda s t. \text{merge-guards } (c\ s\ t)$ )  
(*proof*)

**lemma** *merge-guards-call* [*simp*]:

*merge-guards* (*call* *init* *p* *return* *c*) =  
*call* *init* *p* *return* ( $\lambda s t. \text{merge-guards } (c\ s\ t)$ )  
(*proof*)

**lemma** *merge-guards-call-exn* [*simp*]:

*merge-guards* (*call-exn* *init* *p* *return* *result-exn* *c*) =  
*call-exn* *init* *p* *return* *result-exn* ( $\lambda s t. \text{merge-guards } (c\ s\ t)$ )  
(*proof*)

**lemma** *merge-guards-dynCall* [*simp*]:

*merge-guards* (*dynCall* *init* *p* *return* *c*) =  
*dynCall* *init* *p* *return* ( $\lambda s t. \text{merge-guards } (c\ s\ t)$ )  
(*proof*)

**lemma** *merge-guards-fcall* [*simp*]:

*merge-guards* (*fcall* *init* *p* *return* *result* *c*) =  
*fcall* *init* *p* *return* *result* ( $\lambda v. \text{merge-guards } (c\ v)$ )  
(*proof*)

**lemma** *merge-guards-switch* [*simp*]:

*merge-guards* (*switch* *v* *vs*) =  
*switch* *v* (*map* ( $\lambda (V, c). (V, \text{merge-guards } c)$ ) *vs*)  
(*proof*)

**lemma** *merge-guards-guaranteeStrip* [*simp*]:

*merge-guards* (*guaranteeStrip* *f* *g* *c*) =  
(*let* *c'* = (*merge-guards* *c*)  
  *in if* *is-Guard* *c'*  
    *then let* (*f', g', c'*) = *dest-Guard* *c'*  
      *in if* *f=f'* *then* *Guard* *f* (*g*  $\cap$  *g'*) *c'*  
      *else* *Guard* *f* *g* (*Guard* *f'* *g'* *c'*)  
    *else* *Guard* *f* *g* *c'*)  
(*proof*)

**lemma** *merge-guards-whileAnno* [*simp*]:  
 $\text{merge-guards } (\text{whileAnno } b \ I \ V \ c) = \text{whileAnno } b \ I \ V \ (\text{merge-guards } c)$   
 ⟨*proof*⟩

**lemma** *merge-guards-specAnno* [*simp*]:  
 $\text{merge-guards } (\text{specAnno } P \ c \ Q \ A) =$   
 $\text{specAnno } P \ (\lambda s. \text{merge-guards } (c \ \text{undefined})) \ Q \ A$   
 ⟨*proof*⟩

*merge-guards* for guard-lists as in *guards*, *while* and *whileAnnoG* may have funny effects since the guard-list has to be merged with the body statement too.

**lemmas** *merge-guards-simps* = *merge-guards.simps* *merge-guards-raise*  
*merge-guards-condCatch* *merge-guards-bind* *merge-guards-bseq* *merge-guards-block*  
*merge-guards-dynCall* *merge-guards-fcall* *merge-guards-switch*  
*merge-guards-block-exn* *merge-guards-call-exn*  
*merge-guards-guaranteeStrip* *merge-guards-whileAnno* *merge-guards-specAnno*

**primrec** *noguards*:: ('s,'p,'f) com  $\Rightarrow$  bool

**where**

*noguards* *Skip* = *True* |  
*noguards* (*Basic* *f*) = *True* |  
*noguards* (*Spec* *r*) = *True* |  
*noguards* (*Seq* *c*<sub>1</sub> *c*<sub>2</sub>) = (*noguards* *c*<sub>1</sub>  $\wedge$  *noguards* *c*<sub>2</sub>) |  
*noguards* (*Cond* *b* *c*<sub>1</sub> *c*<sub>2</sub>) = (*noguards* *c*<sub>1</sub>  $\wedge$  *noguards* *c*<sub>2</sub>) |  
*noguards* (*While* *b* *c*) = (*noguards* *c*) |  
*noguards* (*Call* *p*) = *True* |  
*noguards* (*DynCom* *c*) = ( $\forall s. \text{noguards } (c \ s)$ ) |  
*noguards* (*Guard* *f* *g* *c*) = *False* |  
*noguards* *Throw* = *True* |  
*noguards* (*Catch* *c*<sub>1</sub> *c*<sub>2</sub>) = (*noguards* *c*<sub>1</sub>  $\wedge$  *noguards* *c*<sub>2</sub>)

**lemma** *noguards-strip-guards*: *noguards* (*strip-guards* *UNIV* *c*)  
 ⟨*proof*⟩

**primrec** *nothrows*:: ('s,'p,'f) com  $\Rightarrow$  bool

**where**

*nothrows* *Skip* = *True* |  
*nothrows* (*Basic* *f*) = *True* |  
*nothrows* (*Spec* *r*) = *True* |  
*nothrows* (*Seq* *c*<sub>1</sub> *c*<sub>2</sub>) = (*nothrows* *c*<sub>1</sub>  $\wedge$  *nothrows* *c*<sub>2</sub>) |  
*nothrows* (*Cond* *b* *c*<sub>1</sub> *c*<sub>2</sub>) = (*nothrows* *c*<sub>1</sub>  $\wedge$  *nothrows* *c*<sub>2</sub>) |  
*nothrows* (*While* *b* *c*) = *nothrows* *c* |  
*nothrows* (*Call* *p*) = *True* |  
*nothrows* (*DynCom* *c*) = ( $\forall s. \text{nothrows } (c \ s)$ ) |  
*nothrows* (*Guard* *f* *g* *c*) = *nothrows* *c* |  
*nothrows* *Throw* = *False* |  
*nothrows* (*Catch* *c*<sub>1</sub> *c*<sub>2</sub>) = (*nothrows* *c*<sub>1</sub>  $\wedge$  *nothrows* *c*<sub>2</sub>)

### 2.3.5 Intersecting Guards: $c_1 \cap_g c_2$

**inductive-set** *com-rel* :: ((*'s,'p,'f*) *com* × (*'s,'p,'f*) *com*) *set*

**where**

(*c1*, *Seq c1 c2*) ∈ *com-rel*  
 | (*c2*, *Seq c1 c2*) ∈ *com-rel*  
 | (*c1*, *Cond b c1 c2*) ∈ *com-rel*  
 | (*c2*, *Cond b c1 c2*) ∈ *com-rel*  
 | (*c*, *While b c*) ∈ *com-rel*  
 | (*c x*, *DynCom c*) ∈ *com-rel*  
 | (*c*, *Guard f g c*) ∈ *com-rel*  
 | (*c1*, *Catch c1 c2*) ∈ *com-rel*  
 | (*c2*, *Catch c1 c2*) ∈ *com-rel*

**inductive-cases** *com-rel-elim-cases*:

(*c*, *Skip*) ∈ *com-rel*  
 (*c*, *Basic f*) ∈ *com-rel*  
 (*c*, *Spec r*) ∈ *com-rel*  
 (*c*, *Seq c1 c2*) ∈ *com-rel*  
 (*c*, *Cond b c1 c2*) ∈ *com-rel*  
 (*c*, *While b c1*) ∈ *com-rel*  
 (*c*, *Call p*) ∈ *com-rel*  
 (*c*, *DynCom c1*) ∈ *com-rel*  
 (*c*, *Guard f g c1*) ∈ *com-rel*  
 (*c*, *Throw*) ∈ *com-rel*  
 (*c*, *Catch c1 c2*) ∈ *com-rel*

**lemma** *wf-com-rel*: *wf com-rel*

*<proof>*

**consts** *inter-guards*:: (*'s,'p,'f*) *com* × (*'s,'p,'f*) *com* ⇒ (*'s,'p,'f*) *com option*

**abbreviation**

*inter-guards-syntax* :: (*'s,'p,'f*) *com* ⇒ (*'s,'p,'f*) *com* ⇒ (*'s,'p,'f*) *com option*  
 (←  $\cap_g$  → [20,20] 19)

**where**  $c \cap_g d == \text{inter-guards } (c,d)$

**recdef** *inter-guards inv-image com-rel fst*

(*Skip*  $\cap_g$  *Skip*) = *Some Skip*  
 (*Basic f1*  $\cap_g$  *Basic f2*) = (if *f1* = *f2* then *Some (Basic f1)* else *None*)  
 (*Spec r1*  $\cap_g$  *Spec r2*) = (if *r1* = *r2* then *Some (Spec r1)* else *None*)  
 (*Seq a1 a2*  $\cap_g$  *Seq b1 b2*) =  
 (case *a1*  $\cap_g$  *b1* of  
   *None* ⇒ *None*  
   | *Some c1* ⇒ (case *a2*  $\cap_g$  *b2* of  
     *None* ⇒ *None*  
     | *Some c2* ⇒ *Some (Seq c1 c2)*)))  
 (*Cond cnd1 t1 e1*  $\cap_g$  *Cond cnd2 t2 e2*) =  
 (if *cnd1* = *cnd2*  
   then (case *t1*  $\cap_g$  *t2* of

$None \Rightarrow None$   
 $| Some\ t \Rightarrow (case\ e1\ \cap_g\ e2\ of$   
 $\quad None \Rightarrow None$   
 $\quad | Some\ e \Rightarrow Some\ (Cond\ cnd1\ t\ e)))$   
 $else\ None)$   
 $(While\ cnd1\ c1\ \cap_g\ While\ cnd2\ c2) =$   
 $(if\ cnd1 = cnd2$   
 $\quad then\ (case\ c1\ \cap_g\ c2\ of$   
 $\quad\quad None \Rightarrow None$   
 $\quad\quad | Some\ c \Rightarrow Some\ (While\ cnd1\ c))$   
 $\quad else\ None)$   
 $(Call\ p1\ \cap_g\ Call\ p2) =$   
 $(if\ p1 = p2$   
 $\quad then\ Some\ (Call\ p1)$   
 $\quad else\ None)$   
 $(DynCom\ P1\ \cap_g\ DynCom\ P2) =$   
 $(if\ (\forall\ s.\ (P1\ s\ \cap_g\ P2\ s) \neq None)$   
 $\quad then\ Some\ (DynCom\ (\lambda s.\ the\ (P1\ s\ \cap_g\ P2\ s)))$   
 $\quad else\ None)$   
 $(Guard\ m1\ g1\ c1\ \cap_g\ Guard\ m2\ g2\ c2) =$   
 $(if\ m1 = m2\ then$   
 $\quad (case\ c1\ \cap_g\ c2\ of$   
 $\quad\quad None \Rightarrow None$   
 $\quad\quad | Some\ c \Rightarrow Some\ (Guard\ m1\ (g1\ \cap\ g2)\ c))$   
 $\quad else\ None)$   
 $(Throw\ \cap_g\ Throw) = Some\ Throw$   
 $(Catch\ a1\ a2\ \cap_g\ Catch\ b1\ b2) =$   
 $(case\ a1\ \cap_g\ b1\ of$   
 $\quad None \Rightarrow None$   
 $\quad | Some\ c1 \Rightarrow (case\ a2\ \cap_g\ b2\ of$   
 $\quad\quad None \Rightarrow None$   
 $\quad\quad | Some\ c2 \Rightarrow Some\ (Catch\ c1\ c2)))$   
 $(c\ \cap_g\ d) = None$   
**(hints** *cong add: option.case-cong if-cong*  
*recdef-wf: wf-com-rel simp: com-rel.intros*)

**lemma** *inter-guards-strip-eq:*

$\bigwedge c.\ (c1\ \cap_g\ c2) = Some\ c \implies$   
 $(strip-guards\ UNIV\ c = strip-guards\ UNIV\ c1) \wedge$   
 $(strip-guards\ UNIV\ c = strip-guards\ UNIV\ c2)$   
 $\langle proof \rangle$

**lemma** *inter-guards-sym:*  $\bigwedge c.\ (c1\ \cap_g\ c2) = Some\ c \implies (c2\ \cap_g\ c1) = Some\ c$   
 $\langle proof \rangle$

**lemma** *inter-guards-Skip:*  $(Skip\ \cap_g\ c2) = Some\ c = (c2=Skip \wedge c=Skip)$   
 $\langle proof \rangle$

**lemma** *inter-guards-Basic*:  
 $((Basic\ f) \cap_g c2) = Some\ c = (c2=Basic\ f \wedge c=Basic\ f)$   
 $\langle proof \rangle$

**lemma** *inter-guards-Spec*:  
 $((Spec\ r) \cap_g c2) = Some\ c = (c2=Spec\ r \wedge c=Spec\ r)$   
 $\langle proof \rangle$

**lemma** *inter-guards-Seq*:  
 $(Seq\ a1\ a2 \cap_g c2) = Some\ c =$   
 $(\exists b1\ b2\ d1\ d2. c2=Seq\ b1\ b2 \wedge (a1 \cap_g b1) = Some\ d1 \wedge$   
 $(a2 \cap_g b2) = Some\ d2 \wedge c=Seq\ d1\ d2)$   
 $\langle proof \rangle$

**lemma** *inter-guards-Cond*:  
 $(Cond\ cnd\ t1\ e1 \cap_g c2) = Some\ c =$   
 $(\exists t2\ e2\ t\ e. c2=Cond\ cnd\ t2\ e2 \wedge (t1 \cap_g t2) = Some\ t \wedge$   
 $(e1 \cap_g e2) = Some\ e \wedge c=Cond\ cnd\ t\ e)$   
 $\langle proof \rangle$

**lemma** *inter-guards-While*:  
 $(While\ cnd\ bdy1 \cap_g c2) = Some\ c =$   
 $(\exists bdy2\ bdy. c2=While\ cnd\ bdy2 \wedge (bdy1 \cap_g bdy2) = Some\ bdy \wedge$   
 $c=While\ cnd\ bdy)$   
 $\langle proof \rangle$

**lemma** *inter-guards-Call*:  
 $(Call\ p \cap_g c2) = Some\ c =$   
 $(c2=Call\ p \wedge c=Call\ p)$   
 $\langle proof \rangle$

**lemma** *inter-guards-DynCom*:  
 $(DynCom\ f1 \cap_g c2) = Some\ c =$   
 $(\exists f2. c2=DynCom\ f2 \wedge (\forall s. ((f1\ s) \cap_g (f2\ s)) \neq None) \wedge$   
 $c=DynCom\ (\lambda s. the\ ((f1\ s) \cap_g (f2\ s))))$   
 $\langle proof \rangle$

**lemma** *inter-guards-Guard*:  
 $(Guard\ f\ g1\ bdy1 \cap_g c2) = Some\ c =$   
 $(\exists g2\ bdy2\ bdy. c2=Guard\ f\ g2\ bdy2 \wedge (bdy1 \cap_g bdy2) = Some\ bdy \wedge$   
 $c=Guard\ f\ (g1 \cap_g g2)\ bdy)$   
 $\langle proof \rangle$

**lemma** *inter-guards-Throw*:  
 $(Throw \cap_g c2) = Some\ c = (c2=Throw \wedge c=Throw)$   
 $\langle proof \rangle$

**lemma** *inter-guards-Catch*:

$(\text{Catch } a1 \ a2 \ \cap_g \ c2) = \text{Some } c =$   
 $(\exists b1 \ b2 \ d1 \ d2. c2 = \text{Catch } b1 \ b2 \wedge (a1 \ \cap_g \ b1) = \text{Some } d1 \ \wedge$   
 $(a2 \ \cap_g \ b2) = \text{Some } d2 \ \wedge c = \text{Catch } d1 \ d2)$   
 $\langle \text{proof} \rangle$

**lemmas** *inter-guards-simps* = *inter-guards-Skip* *inter-guards-Basic* *inter-guards-Spec*  
*inter-guards-Seq* *inter-guards-Cond* *inter-guards-While* *inter-guards-Call*  
*inter-guards-DynCom* *inter-guards-Guard* *inter-guards-Throw*  
*inter-guards-Catch*

### 2.3.6 Subset on Guards: $c_1 \subseteq_g c_2$

**inductive** *subseteq-guards* :: ('s,'p,'f) com  $\Rightarrow$  ('s,'p,'f) com  $\Rightarrow$  bool

( $\langle \_ \subseteq_g \_ \rangle$  [20,20] 19) **where**

$\text{Skip} \subseteq_g \text{Skip}$   
 $| f1 = f2 \Longrightarrow \text{Basic } f1 \subseteq_g \text{Basic } f2$   
 $| r1 = r2 \Longrightarrow \text{Spec } r1 \subseteq_g \text{Spec } r2$   
 $| a1 \subseteq_g b1 \Longrightarrow a2 \subseteq_g b2 \Longrightarrow \text{Seq } a1 \ a2 \subseteq_g \text{Seq } b1 \ b2$   
 $| \text{cnd1} = \text{cnd2} \Longrightarrow t1 \subseteq_g t2 \Longrightarrow e1 \subseteq_g e2 \Longrightarrow \text{Cond } \text{cnd1 } t1 \ e1 \subseteq_g \text{Cond } \text{cnd2}$   
 $t2 \ e2$   
 $| \text{cnd1} = \text{cnd2} \Longrightarrow c1 \subseteq_g c2 \Longrightarrow \text{While } \text{cnd1 } c1 \subseteq_g \text{While } \text{cnd2 } c2$   
 $| p1 = p2 \Longrightarrow \text{Call } p1 \subseteq_g \text{Call } p2$   
 $| (\bigwedge s. P1 \ s \subseteq_g P2 \ s) \Longrightarrow \text{DynCom } P1 \subseteq_g \text{DynCom } P2$   
 $| m1 = m2 \Longrightarrow g1 = g2 \Longrightarrow c1 \subseteq_g c2 \Longrightarrow \text{Guard } m1 \ g1 \ c1 \subseteq_g \text{Guard } m2 \ g2 \ c2$   
 $| c1 \subseteq_g c2 \Longrightarrow c1 \subseteq_g \text{Guard } m2 \ g2 \ c2$   
 $| \text{Throw} \subseteq_g \text{Throw}$   
 $| a1 \subseteq_g b1 \Longrightarrow a2 \subseteq_g b2 \Longrightarrow \text{Catch } a1 \ a2 \subseteq_g \text{Catch } b1 \ b2$

**lemma** *subseteq-guards-Skip*:

$c = \text{Skip}$  **if**  $c \subseteq_g \text{Skip}$

$\langle \text{proof} \rangle$

**lemma** *subseteq-guards-Basic*:

$c = \text{Basic } f$  **if**  $c \subseteq_g \text{Basic } f$

$\langle \text{proof} \rangle$

**lemma** *subseteq-guards-Spec*:

$c = \text{Spec } r$  **if**  $c \subseteq_g \text{Spec } r$

$\langle \text{proof} \rangle$

**lemma** *subseteq-guards-Seq*:

$\exists c1' \ c2'. c = \text{Seq } c1' \ c2' \wedge (c1' \subseteq_g c1) \wedge (c2' \subseteq_g c2)$  **if**  $c \subseteq_g \text{Seq } c1 \ c2$

$\langle \text{proof} \rangle$

**lemma** *subseteq-guards-Cond*:

$\exists c1' \ c2'. c = \text{Cond } b \ c1' \ c2' \wedge (c1' \subseteq_g c1) \wedge (c2' \subseteq_g c2)$  **if**  $c \subseteq_g \text{Cond } b \ c1 \ c2$

$\langle \text{proof} \rangle$

**lemma** *subseteq-guards-While*:

$\exists c''. c = \text{While } b \ c'' \wedge (c'' \subseteq_g c') \text{ if } c \subseteq_g \text{While } b \ c'$   
*<proof>*

**lemma** *subseteq-guards-Call*:

$c = \text{Call } p \text{ if } c \subseteq_g \text{Call } p$   
*<proof>*

**lemma** *subseteq-guards-DynCom*:

$\exists C'. c = \text{DynCom } C' \wedge (\forall s. C' \ s \subseteq_g C \ s) \text{ if } c \subseteq_g \text{DynCom } C$   
*<proof>*

**lemma** *subseteq-guards-Guard*:

$(c \subseteq_g c') \vee (\exists c''. c = \text{Guard } f \ g \ c'' \wedge (c'' \subseteq_g c')) \text{ if } c \subseteq_g \text{Guard } f \ g \ c'$   
*<proof>*

**lemma** *subseteq-guards-Throw*:

$c = \text{Throw} \text{ if } c \subseteq_g \text{Throw}$   
*<proof>*

**lemma** *subseteq-guards-Catch*:

$\exists c1' \ c2'. c = \text{Catch } c1' \ c2' \wedge (c1' \subseteq_g c1) \wedge (c2' \subseteq_g c2) \text{ if } c \subseteq_g \text{Catch } c1 \ c2$   
*<proof>*

**lemmas** *subseteq-guardsD = subseteq-guards-Skip subseteq-guards-Basic  
subseteq-guards-Spec subseteq-guards-Seq subseteq-guards-Cond subseteq-guards-While  
subseteq-guards-Call subseteq-guards-DynCom subseteq-guards-Guard  
subseteq-guards-Throw subseteq-guards-Catch*

**lemma** *subseteq-guards-Guard'*:

$\exists f' \ b' \ c'. d = \text{Guard } f' \ b' \ c' \text{ if } \text{Guard } f \ b \ c \subseteq_g d$   
*<proof>*

**lemma** *subseteq-guards-refl*:  $c \subseteq_g c$

*<proof>*

end

### 3 Big-Step Semantics for Simpl

**theory** *Semantic imports Language begin*

**notation**

*restrict-map*  $\langle \cdot | \cdot \rangle [90, 91] \ 90$

**datatype**  $(s, f) \ xstate = \text{Normal } 's \ | \ \text{Abrupt } 's \ | \ \text{Fault } 'f \ | \ \text{Stuck}$

**definition**  $isAbr::('s,'f) \text{ xstate} \Rightarrow \text{bool}$   
**where**  $isAbr S = (\exists s. S=Abrupt s)$

**lemma**  $isAbr\text{-simps}$   $[simp]$ :  
 $isAbr (Normal s) = False$   
 $isAbr (Abrupt s) = True$   
 $isAbr (Fault f) = False$   
 $isAbr Stuck = False$   
 $\langle proof \rangle$

**lemma**  $isAbrE$   $[consumes\ 1, elim?]$ :  $\llbracket isAbr S; \bigwedge s. S=Abrupt s \implies P \rrbracket \implies P$   
 $\langle proof \rangle$

**lemma**  $not\text{-}isAbrD$ :  
 $\neg isAbr s \implies (\exists s'. s=Normal s') \vee s = Stuck \vee (\exists f. s=Fault f)$   
 $\langle proof \rangle$

**definition**  $isFault::('s,'f) \text{ xstate} \Rightarrow \text{bool}$   
**where**  $isFault S = (\exists f. S=Fault f)$

**lemma**  $isFault\text{-simps}$   $[simp]$ :  
 $isFault (Normal s) = False$   
 $isFault (Abrupt s) = False$   
 $isFault (Fault f) = True$   
 $isFault Stuck = False$   
 $\langle proof \rangle$

**lemma**  $isFaultE$   $[consumes\ 1, elim?]$ :  $\llbracket isFault s; \bigwedge f. s=Fault f \implies P \rrbracket \implies P$   
 $\langle proof \rangle$

**lemma**  $not\text{-}isFault\text{-}iff$ :  $(\neg isFault t) = (\forall f. t \neq Fault f)$   
 $\langle proof \rangle$

### 3.1 Big-Step Execution: $\Gamma \vdash \langle c, s \rangle \Rightarrow t$

The procedure environment

**type-synonym**  $('s,'p,'f) \text{ body} = 'p \Rightarrow ('s,'p,'f) \text{ com option}$

**inductive**  
 $exec::('s,'p,'f) \text{ body}, ('s,'p,'f) \text{ com}, ('s,'f) \text{ xstate}, ('s,'f) \text{ xstate}$   
 $\Rightarrow \text{bool} (\langle \vdash \langle -, - \rangle \Rightarrow \rightarrow [60,20,98,98] 89)$

**for**  $\Gamma::('s,'p,'f) \text{ body}$

**where**

$Skip: \Gamma \vdash \langle Skip, Normal s \rangle \Rightarrow Normal s$

|  $Guard: \llbracket s \in g; \Gamma \vdash \langle c, Normal s \rangle \Rightarrow t \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle Guard f g c, Normal s \rangle \Rightarrow t$

$$\begin{aligned}
& | \text{GuardFault}: s \notin g \implies \Gamma \vdash \langle \text{Guard } f \ g \ c, \text{Normal } s \rangle \Rightarrow \text{Fault } f \\
& | \text{FaultProp} [\text{intro}, \text{simp}]: \Gamma \vdash \langle c, \text{Fault } f \rangle \Rightarrow \text{Fault } f \\
& | \text{Basic}: \Gamma \vdash \langle \text{Basic } f, \text{Normal } s \rangle \Rightarrow \text{Normal } (f \ s) \\
& | \text{Spec}: (s, t) \in r \\
& \quad \implies \\
& \quad \Gamma \vdash \langle \text{Spec } r, \text{Normal } s \rangle \Rightarrow \text{Normal } t \\
& | \text{SpecStuck}: \forall t. (s, t) \notin r \\
& \quad \implies \\
& \quad \Gamma \vdash \langle \text{Spec } r, \text{Normal } s \rangle \Rightarrow \text{Stuck} \\
& | \text{Seq}: [\Gamma \vdash \langle c_1, \text{Normal } s \rangle \Rightarrow s'; \Gamma \vdash \langle c_2, s' \rangle \Rightarrow t] \\
& \quad \implies \\
& \quad \Gamma \vdash \langle \text{Seq } c_1 \ c_2, \text{Normal } s \rangle \Rightarrow t \\
& | \text{CondTrue}: [s \in b; \Gamma \vdash \langle c_1, \text{Normal } s \rangle \Rightarrow t] \\
& \quad \implies \\
& \quad \Gamma \vdash \langle \text{Cond } b \ c_1 \ c_2, \text{Normal } s \rangle \Rightarrow t \\
& | \text{CondFalse}: [s \notin b; \Gamma \vdash \langle c_2, \text{Normal } s \rangle \Rightarrow t] \\
& \quad \implies \\
& \quad \Gamma \vdash \langle \text{Cond } b \ c_1 \ c_2, \text{Normal } s \rangle \Rightarrow t \\
& | \text{WhileTrue}: [s \in b; \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow s'; \Gamma \vdash \langle \text{While } b \ c, s' \rangle \Rightarrow t] \\
& \quad \implies \\
& \quad \Gamma \vdash \langle \text{While } b \ c, \text{Normal } s \rangle \Rightarrow t \\
& | \text{WhileFalse}: [s \notin b] \\
& \quad \implies \\
& \quad \Gamma \vdash \langle \text{While } b \ c, \text{Normal } s \rangle \Rightarrow \text{Normal } s \\
& | \text{Call}: [\Gamma \ p = \text{Some } \text{bdy}; \Gamma \vdash \langle \text{bdy}, \text{Normal } s \rangle \Rightarrow t] \\
& \quad \implies \\
& \quad \Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle \Rightarrow t \\
& | \text{CallUndefined}: [\Gamma \ p = \text{None}] \\
& \quad \implies \\
& \quad \Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle \Rightarrow \text{Stuck} \\
& | \text{StuckProp} [\text{intro}, \text{simp}]: \Gamma \vdash \langle c, \text{Stuck} \rangle \Rightarrow \text{Stuck} \\
& | \text{DynCom}: [\Gamma \vdash \langle (c \ s), \text{Normal } s \rangle \Rightarrow t] \\
& \quad \implies \\
& \quad \Gamma \vdash \langle \text{DynCom } c, \text{Normal } s \rangle \Rightarrow t
\end{aligned}$$

| *Throw*:  $\Gamma \vdash \langle \text{Throw}, \text{Normal } s \rangle \Rightarrow \text{Abrupt } s$

| *AbruptProp* [*intro, simp*]:  $\Gamma \vdash \langle c, \text{Abrupt } s \rangle \Rightarrow \text{Abrupt } s$

| *CatchMatch*:  $\llbracket \Gamma \vdash \langle c_1, \text{Normal } s \rangle \Rightarrow \text{Abrupt } s'; \Gamma \vdash \langle c_2, \text{Normal } s' \rangle \Rightarrow t \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{Catch } c_1 \ c_2, \text{Normal } s \rangle \Rightarrow t$

| *CatchMiss*:  $\llbracket \Gamma \vdash \langle c_1, \text{Normal } s \rangle \Rightarrow t; \neg \text{isAbr } t \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{Catch } c_1 \ c_2, \text{Normal } s \rangle \Rightarrow t$

**inductive-cases** *exec-elim-cases* [*cases set*]:

$\Gamma \vdash \langle c, \text{Fault } f \rangle \Rightarrow t$   
 $\Gamma \vdash \langle c, \text{Stuck} \rangle \Rightarrow t$   
 $\Gamma \vdash \langle c, \text{Abrupt } s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Skip}, s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Seq } c_1 \ c_2, s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Guard } f \ g \ c, s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Basic } f, s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Spec } r, s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Cond } b \ c_1 \ c_2, s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{While } b \ c, s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Call } p, s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{DynCom } c, s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Throw}, s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Catch } c_1 \ c_2, s \rangle \Rightarrow t$

**inductive-cases** *exec-Normal-elim-cases* [*cases set*]:

$\Gamma \vdash \langle c, \text{Fault } f \rangle \Rightarrow t$   
 $\Gamma \vdash \langle c, \text{Stuck} \rangle \Rightarrow t$   
 $\Gamma \vdash \langle c, \text{Abrupt } s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Skip}, \text{Normal } s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Guard } f \ g \ c, \text{Normal } s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Basic } f, \text{Normal } s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Spec } r, \text{Normal } s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Seq } c_1 \ c_2, \text{Normal } s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Cond } b \ c_1 \ c_2, \text{Normal } s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{While } b \ c, \text{Normal } s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{DynCom } c, \text{Normal } s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Throw}, \text{Normal } s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle \text{Catch } c_1 \ c_2, \text{Normal } s \rangle \Rightarrow t$

**lemma** *exec-block-exn*:

$\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle \Rightarrow \text{Normal } t; \Gamma \vdash \langle c \ s \ t, \text{Normal } (\text{return } s \ t) \rangle \Rightarrow u \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{block-exn } \text{init } \text{bdy } \text{return } \text{result-exn } c, \text{Normal } s \rangle \Rightarrow u$

$\langle proof \rangle$

**lemma** *exec-block*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle bdy, Normal (init\ s) \rangle \Rightarrow Normal\ t; \Gamma \vdash \langle c\ s\ t, Normal (return\ s\ t) \rangle \Rightarrow u \rrbracket \\ & \implies \\ & \Gamma \vdash \langle block\ init\ bdy\ return\ c, Normal\ s \rangle \Rightarrow u \\ & \langle proof \rangle \end{aligned}$$

**lemma** *exec-block-exnAbrupt*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle bdy, Normal (init\ s) \rangle \Rightarrow Abrupt\ t \rrbracket \\ & \implies \\ & \Gamma \vdash \langle block\ exn\ init\ bdy\ return\ result\ exn\ c, Normal\ s \rangle \Rightarrow Abrupt\ (result\ exn \\ & (return\ s\ t)\ t) \\ & \langle proof \rangle \end{aligned}$$

**lemma** *exec-blockAbrupt*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle bdy, Normal (init\ s) \rangle \Rightarrow Abrupt\ t \rrbracket \\ & \implies \\ & \Gamma \vdash \langle block\ init\ bdy\ return\ c, Normal\ s \rangle \Rightarrow Abrupt\ (return\ s\ t) \\ & \langle proof \rangle \end{aligned}$$

**lemma** *exec-block-exnFault*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle bdy, Normal (init\ s) \rangle \Rightarrow Fault\ f \rrbracket \\ & \implies \\ & \Gamma \vdash \langle block\ exn\ init\ bdy\ return\ result\ exn\ c, Normal\ s \rangle \Rightarrow Fault\ f \\ & \langle proof \rangle \end{aligned}$$

**lemma** *exec-blockFault*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle bdy, Normal (init\ s) \rangle \Rightarrow Fault\ f \rrbracket \\ & \implies \\ & \Gamma \vdash \langle block\ init\ bdy\ return\ c, Normal\ s \rangle \Rightarrow Fault\ f \\ & \langle proof \rangle \end{aligned}$$

**lemma** *exec-block-exnStuck*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle bdy, Normal (init\ s) \rangle \Rightarrow Stuck \rrbracket \\ & \implies \\ & \Gamma \vdash \langle block\ exn\ init\ bdy\ return\ result\ exn\ c, Normal\ s \rangle \Rightarrow Stuck \\ & \langle proof \rangle \end{aligned}$$

**lemma** *exec-blockStuck*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle bdy, Normal (init\ s) \rangle \Rightarrow Stuck \rrbracket \\ & \implies \\ & \Gamma \vdash \langle block\ init\ bdy\ return\ c, Normal\ s \rangle \Rightarrow Stuck \\ & \langle proof \rangle \end{aligned}$$

**lemma** *exec-call*:

$$\begin{aligned} & \llbracket \Gamma\ p = Some\ bdy; \Gamma \vdash \langle bdy, Normal (init\ s) \rangle \Rightarrow Normal\ t; \Gamma \vdash \langle c\ s\ t, Normal (return\ s \\ & t) \rangle \Rightarrow u \rrbracket \\ & \implies \end{aligned}$$

$\Gamma \vdash \langle \text{call init } p \text{ return } c, \text{Normal } s \rangle \Rightarrow u$   
 <proof>

**lemma** *exec-callAbrupt*:

$\llbracket \Gamma \text{ } p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init \ s) \rangle \Rightarrow \text{Abrupt } t \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{call init } p \text{ return } c, \text{Normal } s \rangle \Rightarrow \text{Abrupt } (\text{return } s \ t)$   
 <proof>

**lemma** *exec-callFault*:

$\llbracket \Gamma \text{ } p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init \ s) \rangle \Rightarrow \text{Fault } f \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{call init } p \text{ return } c, \text{Normal } s \rangle \Rightarrow \text{Fault } f$   
 <proof>

**lemma** *exec-callStuck*:

$\llbracket \Gamma \text{ } p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init \ s) \rangle \Rightarrow \text{Stuck} \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{call init } p \text{ return } c, \text{Normal } s \rangle \Rightarrow \text{Stuck}$   
 <proof>

**lemma** *exec-callUndefined*:

$\llbracket \Gamma \text{ } p = \text{None} \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{call init } p \text{ return } c, \text{Normal } s \rangle \Rightarrow \text{Stuck}$   
 <proof>

**lemma** *exec-call-exn*:

$\llbracket \Gamma \text{ } p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init \ s) \rangle \Rightarrow \text{Normal } t; \Gamma \vdash \langle c \ s \ t, \text{Normal } (\text{return } s \ t) \rangle \Rightarrow u \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{call-exn init } p \text{ return result-exn } c, \text{Normal } s \rangle \Rightarrow u$   
 <proof>

**lemma** *exec-call-exnAbrupt*:

$\llbracket \Gamma \text{ } p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init \ s) \rangle \Rightarrow \text{Abrupt } t \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{call-exn init } p \text{ return result-exn } c, \text{Normal } s \rangle \Rightarrow \text{Abrupt } (\text{result-exn } (\text{return } s \ t) \ t)$   
 <proof>

**lemma** *exec-call-exnFault*:

$\llbracket \Gamma \text{ } p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init \ s) \rangle \Rightarrow \text{Fault } f \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{call-exn init } p \text{ return result-exn } c, \text{Normal } s \rangle \Rightarrow \text{Fault } f$   
 <proof>

**lemma** *exec-call-exnStuck*:

$\llbracket \Gamma \text{ } p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init \ s) \rangle \Rightarrow \text{Stuck} \rrbracket$

$\implies$   
 $\Gamma \vdash \langle \text{call-exn init } p \text{ return result-exn } c, \text{Normal } s \rangle \Rightarrow \text{Stuck}$   
 <proof>

**lemma** *exec-call-exnUndefined*:

$\llbracket \Gamma \text{ } p = \text{None} \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{call-exn init } p \text{ return result-exn } c, \text{Normal } s \rangle \Rightarrow \text{Stuck}$   
 <proof>

**lemma** *Fault-end*: **assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$  **and**  $s = \text{Fault } f$   
**shows**  $t = \text{Fault } f$   
 <proof>

**lemma** *Stuck-end*: **assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$  **and**  $s = \text{Stuck}$   
**shows**  $t = \text{Stuck}$   
 <proof>

**lemma** *Abrupt-end*: **assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$  **and**  $s = \text{Abrupt } s'$   
**shows**  $t = \text{Abrupt } s'$   
 <proof>

**lemma** *exec-Call-body-aux*:

$\Gamma \text{ } p = \text{Some } \text{bdy} \implies$   
 $\Gamma \vdash \langle \text{Call } p, s \rangle \Rightarrow t = \Gamma \vdash \langle \text{bdy}, s \rangle \Rightarrow t$   
 <proof>

**lemma** *exec-Call-body'*:

$p \in \text{dom } \Gamma \implies$   
 $\Gamma \vdash \langle \text{Call } p, s \rangle \Rightarrow t = \Gamma \vdash \langle \text{the } (\Gamma \text{ } p), s \rangle \Rightarrow t$   
 <proof>

**lemma** *exec-block-exn-Normal-elim* [*consumes 1*]:

**assumes** *exec-block*:  $\Gamma \vdash \langle \text{block-exn init } \text{bdy} \text{ return result-exn } c, \text{Normal } s \rangle \Rightarrow t$

**assumes** *Normal*:

$\wedge t'$ .  
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle \Rightarrow \text{Normal } t';$   
 $\Gamma \vdash \langle c \text{ } s \text{ } t', \text{Normal } (\text{return } s \text{ } t') \rangle \Rightarrow t \rrbracket$   
 $\implies P$

**assumes** *Abrupt*:

$\wedge t'$ .  
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle \Rightarrow \text{Abrupt } t';$   
 $t = \text{Abrupt } (\text{result-exn } (\text{return } s \text{ } t') \text{ } t') \rrbracket$   
 $\implies P$

**assumes** *Fault*:

$\wedge f$ .  
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle \Rightarrow \text{Fault } f;$

$t = \text{Fault } f]$   
 $\implies P$   
**assumes** *Stuck*:  
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle \Rightarrow \text{Stuck};$   
 $t = \text{Stuck} \rrbracket$   
 $\implies P$   
**assumes**  
 $\llbracket \Gamma \text{ } p = \text{None}; t = \text{Stuck} \rrbracket \implies P$   
**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *exec-block-Normal-elim* [*consumes 1*]:  
**assumes** *exec-block*:  $\Gamma \vdash \langle \text{block } \text{init } \text{bdy } \text{return } c, \text{Normal } s \rangle \Rightarrow t$   
**assumes** *Normal*:  
 $\wedge t'$ .  
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle \Rightarrow \text{Normal } t';$   
 $\Gamma \vdash \langle c \text{ } s \text{ } t', \text{Normal } (\text{return } s \text{ } t') \rangle \Rightarrow t \rrbracket$   
 $\implies P$   
**assumes** *Abrupt*:  
 $\wedge t'$ .  
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle \Rightarrow \text{Abrupt } t';$   
 $t = \text{Abrupt } (\text{return } s \text{ } t') \rrbracket$   
 $\implies P$   
**assumes** *Fault*:  
 $\wedge f$ .  
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle \Rightarrow \text{Fault } f;$   
 $t = \text{Fault } f \rrbracket$   
 $\implies P$   
**assumes** *Stuck*:  
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle \Rightarrow \text{Stuck};$   
 $t = \text{Stuck} \rrbracket$   
 $\implies P$   
**assumes**  
 $\text{Undef}: \llbracket \Gamma \text{ } p = \text{None}; t = \text{Stuck} \rrbracket \implies P$   
**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *exec-call-exn-Normal-elim* [*consumes 1*]:  
**assumes** *exec-call*:  $\Gamma \vdash \langle \text{call-exn } \text{init } p \text{ } \text{return } \text{result-exn } c, \text{Normal } s \rangle \Rightarrow t$   
**assumes** *Normal*:  
 $\wedge \text{bdy } t'$ .  
 $\llbracket \Gamma \text{ } p = \text{Some } \text{bdy}; \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle \Rightarrow \text{Normal } t';$   
 $\Gamma \vdash \langle c \text{ } s \text{ } t', \text{Normal } (\text{return } s \text{ } t') \rangle \Rightarrow t \rrbracket$   
 $\implies P$   
**assumes** *Abrupt*:  
 $\wedge \text{bdy } t'$ .  
 $\llbracket \Gamma \text{ } p = \text{Some } \text{bdy}; \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle \Rightarrow \text{Abrupt } t';$   
 $t = \text{Abrupt } (\text{result-exn } (\text{return } s \text{ } t') \text{ } t') \rrbracket$

$\implies P$   
**assumes** *Fault*:  
 $\wedge bdy\ f.$   
 $\llbracket \Gamma\ p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init\ s) \rangle \Rightarrow \text{Fault } f;$   
 $t = \text{Fault } f \rrbracket$   
 $\implies P$   
**assumes** *Stuck*:  
 $\wedge bdy.$   
 $\llbracket \Gamma\ p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init\ s) \rangle \Rightarrow \text{Stuck};$   
 $t = \text{Stuck} \rrbracket$   
 $\implies P$   
**assumes** *Undef*:  
 $\llbracket \Gamma\ p = \text{None}; t = \text{Stuck} \rrbracket \implies P$   
**shows**  $P$   
 $\langle proof \rangle$

**lemma** *exec-call-Normal-elim* [*consumes 1*]:  
**assumes** *exec-call*:  $\Gamma \vdash \langle call\ init\ p\ return\ c, \text{Normal } s \rangle \Rightarrow t$   
**assumes** *Normal*:  
 $\wedge bdy\ t'.$   
 $\llbracket \Gamma\ p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init\ s) \rangle \Rightarrow \text{Normal } t';$   
 $\Gamma \vdash \langle c\ s\ t', \text{Normal } (return\ s\ t') \rangle \Rightarrow t \rrbracket$   
 $\implies P$   
**assumes** *Abrupt*:  
 $\wedge bdy\ t'.$   
 $\llbracket \Gamma\ p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init\ s) \rangle \Rightarrow \text{Abrupt } t';$   
 $t = \text{Abrupt } (return\ s\ t') \rrbracket$   
 $\implies P$   
**assumes** *Fault*:  
 $\wedge bdy\ f.$   
 $\llbracket \Gamma\ p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init\ s) \rangle \Rightarrow \text{Fault } f;$   
 $t = \text{Fault } f \rrbracket$   
 $\implies P$   
**assumes** *Stuck*:  
 $\wedge bdy.$   
 $\llbracket \Gamma\ p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init\ s) \rangle \Rightarrow \text{Stuck};$   
 $t = \text{Stuck} \rrbracket$   
 $\implies P$   
**assumes** *Undef*:  
 $\llbracket \Gamma\ p = \text{None}; t = \text{Stuck} \rrbracket \implies P$   
**shows**  $P$   
 $\langle proof \rangle$

**lemma** *exec-dynCall*:  
 $\llbracket \Gamma \vdash \langle call\ init\ (p\ s)\ return\ c, \text{Normal } s \rangle \Rightarrow t \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle dynCall\ init\ p\ return\ c, \text{Normal } s \rangle \Rightarrow t$   
 $\langle proof \rangle$

**lemma** *exec-dynCall-exn*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle \text{call-exn init } (p \ s) \ \text{return result-exn } c, \text{Normal } s \rangle \Rightarrow t \rrbracket \\ & \implies \\ & \Gamma \vdash \langle \text{dynCall-exn } f \ \text{UNIV init } p \ \text{return result-exn } c, \text{Normal } s \rangle \Rightarrow t \end{aligned}$$

*<proof>*

**lemma** *exec-dynCall-Normal-elim*:

**assumes** *exec*:  $\Gamma \vdash \langle \text{dynCall init } p \ \text{return } c, \text{Normal } s \rangle \Rightarrow t$   
**assumes** *call*:  $\Gamma \vdash \langle \text{call init } (p \ s) \ \text{return } c, \text{Normal } s \rangle \Rightarrow t \implies P$   
**shows**  $P$   
*<proof>*

**lemma** *exec-guards-Normal-elim-cases* [*consumes 1, case-names noFault someFault*]:

**assumes** *exec-guards*:  $\Gamma \vdash \langle \text{guards } gs \ c, \text{Normal } s \rangle \Rightarrow t$   
**assumes** *noFault*:  $\forall f \ g. (f, g) \in \text{set } gs \longrightarrow s \in g \implies \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow t \implies P$   
**assumes** *someFault*:  $\bigwedge f \ g. \text{find } (\lambda(f, g). s \notin g) \ gs = \text{Some } (f, g) \implies t = \text{Fault } f \implies P$   
**shows**  $P$   
*<proof>*

**lemma** *exec-guards-noFault*:

**assumes** *exec*:  $\Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow t$   
**assumes** *noFault*:  $\forall f \ g. (f, g) \in \text{set } gs \longrightarrow s \in g$   
**shows**  $\Gamma \vdash \langle \text{guards } gs \ c, \text{Normal } s \rangle \Rightarrow t$   
*<proof>*

**lemma** *exec-guards-Fault*:

**assumes** *Fault*:  $\text{find } (\lambda(f, g). s \notin g) \ gs = \text{Some } (f, g)$   
**shows**  $\Gamma \vdash \langle \text{guards } gs \ c, \text{Normal } s \rangle \Rightarrow \text{Fault } f$   
*<proof>*

**lemma** *exec-guards-DynCom*:

**assumes** *exec-c*:  $\Gamma \vdash \langle \text{guards } gs \ (c \ s), \text{Normal } s \rangle \Rightarrow t$   
**shows**  $\Gamma \vdash \langle \text{guards } gs \ (\text{DynCom } c), \text{Normal } s \rangle \Rightarrow t$   
*<proof>*

**lemma** *exec-guards-DynCom-Normal-elim*:

**assumes** *exec*:  $\Gamma \vdash \langle \text{guards } gs \ (\text{DynCom } c), \text{Normal } s \rangle \Rightarrow t$   
**assumes** *call*:  $\Gamma \vdash \langle \text{guards } gs \ (c \ s), \text{Normal } s \rangle \Rightarrow t \implies P$   
**shows**  $P$   
*<proof>*

**lemma** *exec-maybe-guard-DynCom*:

**assumes** *exec-c*:  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ (c \ s), \text{Normal } s \rangle \Rightarrow t$   
**shows**  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ (\text{DynCom } c), \text{Normal } s \rangle \Rightarrow t$   
*<proof>*

**lemma** *exec-maybe-guard-Normal-elim-cases* [*consumes 1*, *case-names noFault someFault*]:

**assumes** *exec-guards*:  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ c, \text{Normal } s \rangle \Rightarrow t$   
**assumes** *noFault*:  $s \in g \Longrightarrow \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow t \Longrightarrow P$   
**assumes** *someFault*:  $s \notin g \Longrightarrow t = \text{Fault } f \Longrightarrow P$   
**shows**  $P$   
*<proof>*

**lemma** *exec-maybe-guard-noFault*:

**assumes** *exec*:  $\Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow t$   
**assumes** *noFault*:  $s \in g$   
**shows**  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ c, \text{Normal } s \rangle \Rightarrow t$   
*<proof>*

**lemma** *exec-maybe-guard-Fault*:

**assumes** *Fault*:  $s \notin g$   
**shows**  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ c, \text{Normal } s \rangle \Rightarrow \text{Fault } f$   
*<proof>*

**lemma** *exec-maybe-guard-DynCom-Normal-elim*:

**assumes** *exec*:  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ (\text{DynCom } c), \text{Normal } s \rangle \Rightarrow t$   
**assumes** *call*:  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ (c \ s), \text{Normal } s \rangle \Rightarrow t \Longrightarrow P$   
**shows**  $P$   
*<proof>*

**lemma** *exec-dynCall-exn-Normal-elim*:

**assumes** *exec*:  $\Gamma \vdash \langle \text{dynCall-exn } f \ g \ \text{init } p \ \text{return result-exn } c, \text{Normal } s \rangle \Rightarrow t$   
**assumes** *call*:  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ (\text{call-exn } \text{init } (p \ s) \ \text{return result-exn } c), \text{Normal } s \rangle \Rightarrow t \Longrightarrow P$   
**shows**  $P$   
*<proof>*

**lemma** *exec-Call-body*:

$\Gamma \ p = \text{Some } \text{bdy} \Longrightarrow$   
 $\Gamma \vdash \langle \text{Call } p, s \rangle \Rightarrow t = \Gamma \vdash \langle \text{the } (\Gamma \ p), s \rangle \Rightarrow t$   
*<proof>*

**lemma** *exec-Seq'*:  $\llbracket \Gamma \vdash \langle c1, s \rangle \Rightarrow s'; \Gamma \vdash \langle c2, s' \rangle \Rightarrow s'' \rrbracket$

$\Longrightarrow$   
 $\Gamma \vdash \langle \text{Seq } c1 \ c2, s \rangle \Rightarrow s''$   
*<proof>*

**lemma** *exec-assoc*:  $\Gamma \vdash \langle \text{Seq } c1 \ (\text{Seq } c2 \ c3), s \rangle \Rightarrow t = \Gamma \vdash \langle \text{Seq } (\text{Seq } c1 \ c2) \ c3, s \rangle \Rightarrow t$

*<proof>*

### 3.2 Big-Step Execution with Recursion Limit: $\Gamma \vdash \langle c, s \rangle =n \Rightarrow t$

**inductive** *execn*::[(*'s, 'p, 'f*) *body*, (*'s, 'p, 'f*) *com*, (*'s, 'f*) *xstate*, *nat*, (*'s, 'f*) *xstate*]  
 $\Rightarrow \text{bool } (\vdash \langle -, - \rangle =n \Rightarrow -)$  [60,20,98,65,98] 89)

**for**  $\Gamma :: (\text{'s, 'p, 'f}) \text{ body}$

**where**

*Skip*:  $\Gamma \vdash \langle \text{Skip}, \text{Normal } s \rangle =n \Rightarrow \text{Normal } s$

| *Guard*:  $\llbracket s \in g; \Gamma \vdash \langle c, \text{Normal } s \rangle =n \Rightarrow t \rrbracket$

$\Longrightarrow$

$\Gamma \vdash \langle \text{Guard } f \ g \ c, \text{Normal } s \rangle =n \Rightarrow t$

| *GuardFault*:  $s \notin g \Longrightarrow \Gamma \vdash \langle \text{Guard } f \ g \ c, \text{Normal } s \rangle =n \Rightarrow \text{Fault } f$

| *FaultProp* [*intro, simp*]:  $\Gamma \vdash \langle c, \text{Fault } f \rangle =n \Rightarrow \text{Fault } f$

| *Basic*:  $\Gamma \vdash \langle \text{Basic } f, \text{Normal } s \rangle =n \Rightarrow \text{Normal } (f \ s)$

| *Spec*:  $(s, t) \in r$

$\Longrightarrow$

$\Gamma \vdash \langle \text{Spec } r, \text{Normal } s \rangle =n \Rightarrow \text{Normal } t$

| *SpecStuck*:  $\forall t. (s, t) \notin r$

$\Longrightarrow$

$\Gamma \vdash \langle \text{Spec } r, \text{Normal } s \rangle =n \Rightarrow \text{Stuck}$

| *Seq*:  $\llbracket \Gamma \vdash \langle c_1, \text{Normal } s \rangle =n \Rightarrow s'; \Gamma \vdash \langle c_2, s' \rangle =n \Rightarrow t \rrbracket$

$\Longrightarrow$

$\Gamma \vdash \langle \text{Seq } c_1 \ c_2, \text{Normal } s \rangle =n \Rightarrow t$

| *CondTrue*:  $\llbracket s \in b; \Gamma \vdash \langle c_1, \text{Normal } s \rangle =n \Rightarrow t \rrbracket$

$\Longrightarrow$

$\Gamma \vdash \langle \text{Cond } b \ c_1 \ c_2, \text{Normal } s \rangle =n \Rightarrow t$

| *CondFalse*:  $\llbracket s \notin b; \Gamma \vdash \langle c_2, \text{Normal } s \rangle =n \Rightarrow t \rrbracket$

$\Longrightarrow$

$\Gamma \vdash \langle \text{Cond } b \ c_1 \ c_2, \text{Normal } s \rangle =n \Rightarrow t$

| *WhileTrue*:  $\llbracket s \in b; \Gamma \vdash \langle c, \text{Normal } s \rangle =n \Rightarrow s';$

$\Gamma \vdash \langle \text{While } b \ c, s' \rangle =n \Rightarrow t \rrbracket$

$\Longrightarrow$

$\Gamma \vdash \langle \text{While } b \ c, \text{Normal } s \rangle =n \Rightarrow t$

| *WhileFalse*:  $\llbracket s \notin b \rrbracket$

$\Longrightarrow$

$\Gamma \vdash \langle \text{While } b \ c, \text{Normal } s \rangle =n \Rightarrow \text{Normal } s$

| *Call*:  $\llbracket \Gamma \ p = \text{Some } \text{bdy}; \Gamma \vdash \langle \text{bdy}, \text{Normal } s \rangle =n \Rightarrow t \rrbracket$

$\Longrightarrow$

$\Gamma \vdash \langle \text{Call } p \ , \text{Normal } s \rangle =\text{Suc } n \Rightarrow t$

| *CallUndefined*:  $\llbracket \Gamma \ p=None \rrbracket$   
 $\implies$   
 $\Gamma \langle \text{Call } p, \text{Normal } s \rangle = \text{Suc } n \Rightarrow \text{Stuck}$

| *StuckProp* [*intro,simp*]:  $\Gamma \langle c, \text{Stuck} \rangle = n \Rightarrow \text{Stuck}$

| *DynCom*:  $\llbracket \Gamma \langle (c \ s), \text{Normal } s \rangle = n \Rightarrow t \rrbracket$   
 $\implies$   
 $\Gamma \langle \text{DynCom } c, \text{Normal } s \rangle = n \Rightarrow t$

| *Throw*:  $\Gamma \langle \text{Throw}, \text{Normal } s \rangle = n \Rightarrow \text{Abrupt } s$

| *AbruptProp* [*intro,simp*]:  $\Gamma \langle c, \text{Abrupt } s \rangle = n \Rightarrow \text{Abrupt } s$

| *CatchMatch*:  $\llbracket \Gamma \langle c_1, \text{Normal } s \rangle = n \Rightarrow \text{Abrupt } s'; \Gamma \langle c_2, \text{Normal } s \rangle = n \Rightarrow t \rrbracket$   
 $\implies$   
 $\Gamma \langle \text{Catch } c_1 \ c_2, \text{Normal } s \rangle = n \Rightarrow t$

| *CatchMiss*:  $\llbracket \Gamma \langle c_1, \text{Normal } s \rangle = n \Rightarrow t; \neg \text{isAbr } t \rrbracket$   
 $\implies$   
 $\Gamma \langle \text{Catch } c_1 \ c_2, \text{Normal } s \rangle = n \Rightarrow t$

**inductive-cases** *execn-elim-cases* [*cases set*]:

$\Gamma \langle c, \text{Fault } f \rangle = n \Rightarrow t$   
 $\Gamma \langle c, \text{Stuck} \rangle = n \Rightarrow t$   
 $\Gamma \langle c, \text{Abrupt } s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{Skip}, s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{Seq } c_1 \ c_2, s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{Guard } f \ g \ c, s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{Basic } f, s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{Spec } r, s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{Cond } b \ c_1 \ c_2, s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{While } b \ c, s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{Call } p, s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{DynCom } c, s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{Throw}, s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{Catch } c_1 \ c_2, s \rangle = n \Rightarrow t$

**inductive-cases** *execn-Normal-elim-cases* [*cases set*]:

$\Gamma \langle c, \text{Fault } f \rangle = n \Rightarrow t$   
 $\Gamma \langle c, \text{Stuck} \rangle = n \Rightarrow t$   
 $\Gamma \langle c, \text{Abrupt } s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{Skip}, \text{Normal } s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{Guard } f \ g \ c, \text{Normal } s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{Basic } f, \text{Normal } s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{Spec } r, \text{Normal } s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{Seq } c_1 \ c_2, \text{Normal } s \rangle = n \Rightarrow t$   
 $\Gamma \langle \text{Cond } b \ c_1 \ c_2, \text{Normal } s \rangle = n \Rightarrow t$

$\Gamma \vdash \langle \text{While } b \ c, \text{Normal } s \rangle = n \Rightarrow t$   
 $\Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle = n \Rightarrow t$   
 $\Gamma \vdash \langle \text{DynCom } c, \text{Normal } s \rangle = n \Rightarrow t$   
 $\Gamma \vdash \langle \text{Throw}, \text{Normal } s \rangle = n \Rightarrow t$   
 $\Gamma \vdash \langle \text{Catch } c1 \ c2, \text{Normal } s \rangle = n \Rightarrow t$

**lemma** *execn-Skip'*:  $\Gamma \vdash \langle \text{Skip}, t \rangle = n \Rightarrow t$   
 ⟨proof⟩

**lemma** *execn-Fault-end*: **assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$  **and**  $s = \text{Fault } f$   
**shows**  $t = \text{Fault } f$   
 ⟨proof⟩

**lemma** *execn-Stuck-end*: **assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$  **and**  $s = \text{Stuck}$   
**shows**  $t = \text{Stuck}$   
 ⟨proof⟩

**lemma** *execn-Abrupt-end*: **assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$  **and**  $s = \text{Abrupt } s'$   
**shows**  $t = \text{Abrupt } s'$   
 ⟨proof⟩

**lemma** *execn-block-exn*:  
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle = n \Rightarrow \text{Normal } t; \Gamma \vdash \langle c \ s \ t, \text{Normal } (\text{return } s \ t) \rangle = n \Rightarrow u \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{block-exn } \text{init } \text{bdy } \text{return } \text{result-exn } c, \text{Normal } s \rangle = n \Rightarrow u$   
 ⟨proof⟩

**lemma** *execn-block*:  
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle = n \Rightarrow \text{Normal } t; \Gamma \vdash \langle c \ s \ t, \text{Normal } (\text{return } s \ t) \rangle = n \Rightarrow u \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{block } \text{init } \text{bdy } \text{return } c, \text{Normal } s \rangle = n \Rightarrow u$   
 ⟨proof⟩

**lemma** *execn-block-exnAbrupt*:  
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle = n \Rightarrow \text{Abrupt } t \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{block-exn } \text{init } \text{bdy } \text{return } \text{result-exn } c, \text{Normal } s \rangle = n \Rightarrow \text{Abrupt } (\text{result-exn } (\text{return } s \ t) \ t)$   
 ⟨proof⟩

**lemma** *execn-blockAbrupt*:  
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle = n \Rightarrow \text{Abrupt } t \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{block } \text{init } \text{bdy } \text{return } c, \text{Normal } s \rangle = n \Rightarrow \text{Abrupt } (\text{return } s \ t)$   
 ⟨proof⟩

**lemma** *execn-block-exnFault*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle bdy, Normal (init\ s) \rangle = n \Rightarrow Fault\ f \rrbracket \\ & \implies \\ & \Gamma \vdash \langle block\text{-}exn\ init\ bdy\ return\ result\text{-}exn\ c, Normal\ s \rangle = n \Rightarrow Fault\ f \\ & \langle proof \rangle \end{aligned}$$

**lemma** *execn-blockFault*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle bdy, Normal (init\ s) \rangle = n \Rightarrow Fault\ f \rrbracket \\ & \implies \\ & \Gamma \vdash \langle block\ init\ bdy\ return\ c, Normal\ s \rangle = n \Rightarrow Fault\ f \\ & \langle proof \rangle \end{aligned}$$

**lemma** *execn-block-exnStuck*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle bdy, Normal (init\ s) \rangle = n \Rightarrow Stuck \rrbracket \\ & \implies \\ & \Gamma \vdash \langle block\text{-}exn\ init\ bdy\ return\ result\text{-}exn\ c, Normal\ s \rangle = n \Rightarrow Stuck \\ & \langle proof \rangle \end{aligned}$$

**lemma** *execn-blockStuck*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle bdy, Normal (init\ s) \rangle = n \Rightarrow Stuck \rrbracket \\ & \implies \\ & \Gamma \vdash \langle block\ init\ bdy\ return\ c, Normal\ s \rangle = n \Rightarrow Stuck \\ & \langle proof \rangle \end{aligned}$$

**lemma** *execn-call*:

$$\begin{aligned} & \llbracket \Gamma\ p = Some\ bdy; \Gamma \vdash \langle bdy, Normal (init\ s) \rangle = n \Rightarrow Normal\ t; \\ & \quad \Gamma \vdash \langle c\ s\ t, Normal (return\ s\ t) \rangle = Suc\ n \Rightarrow u \rrbracket \\ & \implies \\ & \Gamma \vdash \langle call\ init\ p\ return\ c, Normal\ s \rangle = Suc\ n \Rightarrow u \\ & \langle proof \rangle \end{aligned}$$

**lemma** *execn-call-exn*:

$$\begin{aligned} & \llbracket \Gamma\ p = Some\ bdy; \Gamma \vdash \langle bdy, Normal (init\ s) \rangle = n \Rightarrow Normal\ t; \\ & \quad \Gamma \vdash \langle c\ s\ t, Normal (return\ s\ t) \rangle = Suc\ n \Rightarrow u \rrbracket \\ & \implies \\ & \Gamma \vdash \langle call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c, Normal\ s \rangle = Suc\ n \Rightarrow u \\ & \langle proof \rangle \end{aligned}$$

**lemma** *execn-callAbrupt*:

$$\begin{aligned} & \llbracket \Gamma\ p = Some\ bdy; \Gamma \vdash \langle bdy, Normal (init\ s) \rangle = n \Rightarrow Abrupt\ t \rrbracket \\ & \implies \\ & \Gamma \vdash \langle call\ init\ p\ return\ c, Normal\ s \rangle = Suc\ n \Rightarrow Abrupt (return\ s\ t) \\ & \langle proof \rangle \end{aligned}$$

**lemma** *execn-call-exnAbrupt*:

$$\begin{aligned} & \llbracket \Gamma\ p = Some\ bdy; \Gamma \vdash \langle bdy, Normal (init\ s) \rangle = n \Rightarrow Abrupt\ t \rrbracket \\ & \implies \\ & \Gamma \vdash \langle call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c, Normal\ s \rangle = Suc\ n \Rightarrow Abrupt (result\text{-}exn \end{aligned}$$

(return s t) t  
 ⟨proof⟩

**lemma** *execn-callFault*:

$$\begin{aligned} & \llbracket \Gamma \text{ p=Some bdy}; \Gamma \vdash \langle \text{bdy}, \text{Normal} (\text{init } s) \rangle =n \Rightarrow \text{Fault } f \rrbracket \\ & \quad \Longrightarrow \\ & \Gamma \vdash \langle \text{call init } p \text{ return } c, \text{Normal } s \rangle =\text{Suc } n \Rightarrow \text{Fault } f \end{aligned}$$

⟨proof⟩

**lemma** *execn-call-exnFault*:

$$\begin{aligned} & \llbracket \Gamma \text{ p=Some bdy}; \Gamma \vdash \langle \text{bdy}, \text{Normal} (\text{init } s) \rangle =n \Rightarrow \text{Fault } f \rrbracket \\ & \quad \Longrightarrow \\ & \Gamma \vdash \langle \text{call-exn init } p \text{ return result-exn } c, \text{Normal } s \rangle =\text{Suc } n \Rightarrow \text{Fault } f \end{aligned}$$

⟨proof⟩

**lemma** *execn-callStuck*:

$$\begin{aligned} & \llbracket \Gamma \text{ p=Some bdy}; \Gamma \vdash \langle \text{bdy}, \text{Normal} (\text{init } s) \rangle =n \Rightarrow \text{Stuck} \rrbracket \\ & \quad \Longrightarrow \\ & \Gamma \vdash \langle \text{call init } p \text{ return } c, \text{Normal } s \rangle =\text{Suc } n \Rightarrow \text{Stuck} \end{aligned}$$

⟨proof⟩

**lemma** *execn-call-exnStuck*:

$$\begin{aligned} & \llbracket \Gamma \text{ p=Some bdy}; \Gamma \vdash \langle \text{bdy}, \text{Normal} (\text{init } s) \rangle =n \Rightarrow \text{Stuck} \rrbracket \\ & \quad \Longrightarrow \\ & \Gamma \vdash \langle \text{call-exn init } p \text{ return result-exn } c, \text{Normal } s \rangle =\text{Suc } n \Rightarrow \text{Stuck} \end{aligned}$$

⟨proof⟩

**lemma** *execn-callUndefined*:

$$\begin{aligned} & \llbracket \Gamma \text{ p=None} \rrbracket \\ & \quad \Longrightarrow \\ & \Gamma \vdash \langle \text{call init } p \text{ return } c, \text{Normal } s \rangle =\text{Suc } n \Rightarrow \text{Stuck} \end{aligned}$$

⟨proof⟩

**lemma** *execn-call-exnUndefined*:

$$\begin{aligned} & \llbracket \Gamma \text{ p=None} \rrbracket \\ & \quad \Longrightarrow \\ & \Gamma \vdash \langle \text{call-exn init } p \text{ return result-exn } c, \text{Normal } s \rangle =\text{Suc } n \Rightarrow \text{Stuck} \end{aligned}$$

⟨proof⟩

**lemma** *execn-block-exn-Normal-elim* [consumes 1]:

**assumes** *execn-block*:  $\Gamma \vdash \langle \text{block-exn init bdy return result-exn } c, \text{Normal } s \rangle =n \Rightarrow t$

**assumes** *Normal*:

$$\begin{aligned} & \wedge t'. \\ & \llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal} (\text{init } s) \rangle =n \Rightarrow \text{Normal } t'; \\ & \Gamma \vdash \langle c \text{ s } t', \text{Normal} (\text{return } s \text{ } t') \rangle =n \Rightarrow t \rrbracket \\ & \quad \Longrightarrow P \end{aligned}$$

**assumes** *Abrupt*:

$$\begin{aligned} & \wedge t'. \\ & \llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal} (\text{init } s) \rangle =n \Rightarrow \text{Abrupt } t'; \end{aligned}$$

$t = \text{Abrupt } (\text{result-exn } (\text{return } s \ t') \ t')$   
 $\implies P$

**assumes** *Fault*:  
 $\wedge f.$   
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle = n \Rightarrow \text{Fault } f; \ t = \text{Fault } f \rrbracket$   
 $\implies P$

**assumes** *Stuck*:  
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle = n \Rightarrow \text{Stuck}; \ t = \text{Stuck} \rrbracket$   
 $\implies P$

**assumes** *Undef*:  
 $\llbracket \Gamma \ p = \text{None}; \ t = \text{Stuck} \rrbracket \implies P$

**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *execn-block-Normal-elim* [*consumes 1*]:  
**assumes** *execn-block*:  $\Gamma \vdash \langle \text{block } \text{init } \text{bdy } \text{return } c, \text{Normal } s \rangle = n \Rightarrow t$   
**assumes** *Normal*:  
 $\wedge t'.$   
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle = n \Rightarrow \text{Normal } t'; \ \Gamma \vdash \langle c \ s \ t', \text{Normal } (\text{return } s \ t') \rangle = n \Rightarrow t \rrbracket$   
 $\implies P$

**assumes** *Abrupt*:  
 $\wedge t'.$   
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle = n \Rightarrow \text{Abrupt } t'; \ t = \text{Abrupt } (\text{return } s \ t') \rrbracket$   
 $\implies P$

**assumes** *Fault*:  
 $\wedge f.$   
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle = n \Rightarrow \text{Fault } f; \ t = \text{Fault } f \rrbracket$   
 $\implies P$

**assumes** *Stuck*:  
 $\llbracket \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle = n \Rightarrow \text{Stuck}; \ t = \text{Stuck} \rrbracket$   
 $\implies P$

**assumes** *Undef*:  
 $\llbracket \Gamma \ p = \text{None}; \ t = \text{Stuck} \rrbracket \implies P$

**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *execn-call-exn-Normal-elim* [*consumes 1*]:  
**assumes** *exec-call*:  $\Gamma \vdash \langle \text{call-exn } \text{init } p \ \text{return } \text{result-exn } c, \text{Normal } s \rangle = n \Rightarrow t$   
**assumes** *Normal*:  
 $\wedge \text{bdy } i \ t'.$   
 $\llbracket \Gamma \ p = \text{Some } \text{bdy}; \ \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle = i \Rightarrow \text{Normal } t'; \ \Gamma \vdash \langle c \ s \ t', \text{Normal } (\text{return } s \ t') \rangle = \text{Suc } i \Rightarrow t; \ n = \text{Suc } i \rrbracket$   
 $\implies P$

**assumes** *Abrupt*:

$\wedge bdy\ i\ t'$ .

$\llbracket \Gamma\ p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init\ s) \rangle = i \Rightarrow \text{Abrupt } t'; n = \text{Suc } i;$   
 $t = \text{Abrupt } (\text{result-exn } (\text{return } s\ t')\ t') \rrbracket$   
 $\Rightarrow P$

**assumes** *Fault*:

$\wedge bdy\ i\ f$ .

$\llbracket \Gamma\ p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init\ s) \rangle = i \Rightarrow \text{Fault } f; n = \text{Suc } i;$   
 $t = \text{Fault } f \rrbracket$   
 $\Rightarrow P$

**assumes** *Stuck*:

$\wedge bdy\ i$ .

$\llbracket \Gamma\ p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init\ s) \rangle = i \Rightarrow \text{Stuck}; n = \text{Suc } i;$   
 $t = \text{Stuck} \rrbracket$   
 $\Rightarrow P$

**assumes** *Undef*:

$\wedge i. \llbracket \Gamma\ p = \text{None}; n = \text{Suc } i; t = \text{Stuck} \rrbracket \Longrightarrow P$

**shows**  $P$

$\langle \text{proof} \rangle$

**lemma** *execn-call-Normal-elim* [*consumes 1*]:

**assumes** *exec-call*:  $\Gamma \vdash \langle \text{call } init\ p\ \text{return } c, \text{Normal } s \rangle = n \Rightarrow t$

**assumes** *Normal*:

$\wedge bdy\ i\ t'$ .

$\llbracket \Gamma\ p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init\ s) \rangle = i \Rightarrow \text{Normal } t';$   
 $\Gamma \vdash \langle c\ s\ t', \text{Normal } (\text{return } s\ t') \rangle = \text{Suc } i \Rightarrow t; n = \text{Suc } i \rrbracket$   
 $\Rightarrow P$

**assumes** *Abrupt*:

$\wedge bdy\ i\ t'$ .

$\llbracket \Gamma\ p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init\ s) \rangle = i \Rightarrow \text{Abrupt } t'; n = \text{Suc } i;$   
 $t = \text{Abrupt } (\text{return } s\ t') \rrbracket$   
 $\Rightarrow P$

**assumes** *Fault*:

$\wedge bdy\ i\ f$ .

$\llbracket \Gamma\ p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init\ s) \rangle = i \Rightarrow \text{Fault } f; n = \text{Suc } i;$   
 $t = \text{Fault } f \rrbracket$   
 $\Rightarrow P$

**assumes** *Stuck*:

$\wedge bdy\ i$ .

$\llbracket \Gamma\ p = \text{Some } bdy; \Gamma \vdash \langle bdy, \text{Normal } (init\ s) \rangle = i \Rightarrow \text{Stuck}; n = \text{Suc } i;$   
 $t = \text{Stuck} \rrbracket$   
 $\Rightarrow P$

**assumes** *Undef*:

$\wedge i. \llbracket \Gamma\ p = \text{None}; n = \text{Suc } i; t = \text{Stuck} \rrbracket \Longrightarrow P$

**shows**  $P$

$\langle \text{proof} \rangle$

**lemma** *execn-dynCall*:

$\llbracket \Gamma \vdash \langle \text{call init } (p \ s) \ \text{return } c, \text{Normal } s \rangle = n \Rightarrow t \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{dynCall init } p \ \text{return } c, \text{Normal } s \rangle = n \Rightarrow t$   
*<proof>*

**lemma** *execn-dynCall-exn*:

$\llbracket \Gamma \vdash \langle \text{call-exn init } (p \ s) \ \text{return result-exn } c, \text{Normal } s \rangle = n \Rightarrow t \rrbracket$   
 $\implies$   
 $\Gamma \vdash \langle \text{dynCall-exn } f \ \text{UNIV init } p \ \text{return result-exn } c, \text{Normal } s \rangle = n \Rightarrow t$   
*<proof>*

**lemma** *execn-dynCall-Normal-elim*:

**assumes** *exec*:  $\Gamma \vdash \langle \text{dynCall init } p \ \text{return } c, \text{Normal } s \rangle = n \Rightarrow t$   
**assumes**  $\Gamma \vdash \langle \text{call init } (p \ s) \ \text{return } c, \text{Normal } s \rangle = n \Rightarrow t \implies P$   
**shows**  $P$   
*<proof>*

**lemma** *execn-guards-Normal-elim-cases* [*consumes 1, case-names noFault someFault*]:

**assumes** *exec-guards*:  $\Gamma \vdash \langle \text{guards } gs \ c, \text{Normal } s \rangle = n \Rightarrow t$   
**assumes** *noFault*:  $\forall f \ g. (f, g) \in \text{set } gs \longrightarrow s \in g \implies \Gamma \vdash \langle c, \text{Normal } s \rangle = n \Rightarrow t$   
 $\implies P$   
**assumes** *someFault*:  $\bigwedge f \ g. \text{find } (\lambda(f, g). s \notin g) \ gs = \text{Some } (f, g) \implies t = \text{Fault } f$   
 $\implies P$   
**shows**  $P$   
*<proof>*

**lemma** *execn-maybe-guard-Normal-elim-cases* [*consumes 1, case-names noFault someFault*]:

**assumes** *exec-guards*:  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ c, \text{Normal } s \rangle = n \Rightarrow t$   
**assumes** *noFault*:  $s \in g \implies \Gamma \vdash \langle c, \text{Normal } s \rangle = n \Rightarrow t \implies P$   
**assumes** *someFault*:  $s \notin g \implies t = \text{Fault } f \implies P$   
**shows**  $P$   
*<proof>*

**lemma** *execn-guards-noFault*:

**assumes** *exec*:  $\Gamma \vdash \langle c, \text{Normal } s \rangle = n \Rightarrow t$   
**assumes** *noFault*:  $\forall f \ g. (f, g) \in \text{set } gs \longrightarrow s \in g$   
**shows**  $\Gamma \vdash \langle \text{guards } gs \ c, \text{Normal } s \rangle = n \Rightarrow t$   
*<proof>*

**lemma** *execn-guards-Fault*:

**assumes** *Fault*:  $\text{find } (\lambda(f, g). s \notin g) \ gs = \text{Some } (f, g)$   
**shows**  $\Gamma \vdash \langle \text{guards } gs \ c, \text{Normal } s \rangle = n \Rightarrow \text{Fault } f$   
*<proof>*

**lemma** *execn-maybe-guard-noFault*:

**assumes** *exec*:  $\Gamma \vdash \langle c, \text{Normal } s \rangle = n \Rightarrow t$

**assumes** *noFault*:  $s \in g$   
**shows**  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ c, \text{Normal } s \rangle = n \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *execn-maybe-guard-Fault*:

**assumes** *Fault*:  $s \notin g$   
**shows**  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ c, \text{Normal } s \rangle = n \Rightarrow \text{Fault } f$   
 $\langle \text{proof} \rangle$

**lemma** *execn-guards-DynCom-Normal-elim*:

**assumes** *exec*:  $\Gamma \vdash \langle \text{guards } gs \ (\text{DynCom } c), \text{Normal } s \rangle = n \Rightarrow t$   
**assumes** *call*:  $\Gamma \vdash \langle \text{guards } gs \ (c \ s), \text{Normal } s \rangle = n \Rightarrow t \Longrightarrow P$   
**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *execn-maybe-guard-DynCom-Normal-elim*:

**assumes** *exec*:  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ (\text{DynCom } c), \text{Normal } s \rangle = n \Rightarrow t$   
**assumes** *call*:  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ (c \ s), \text{Normal } s \rangle = n \Rightarrow t \Longrightarrow P$   
**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *execn-guards-DynCom*:

**assumes** *exec-c*:  $\Gamma \vdash \langle \text{guards } gs \ (c \ s), \text{Normal } s \rangle = n \Rightarrow t$   
**shows**  $\Gamma \vdash \langle \text{guards } gs \ (\text{DynCom } c), \text{Normal } s \rangle = n \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *execn-maybe-guard-DynCom*:

**assumes** *exec-c*:  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ (c \ s), \text{Normal } s \rangle = n \Rightarrow t$   
**shows**  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ (\text{DynCom } c), \text{Normal } s \rangle = n \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *execn-dynCall-exn-Normal-elim*:

**assumes** *exec*:  $\Gamma \vdash \langle \text{dynCall-exn } f \ g \ \text{init } p \ \text{return result-exn } c, \text{Normal } s \rangle = n \Rightarrow t$   
**assumes**  $\Gamma \vdash \langle \text{maybe-guard } f \ g \ (\text{call-exn } \text{init } (p \ s) \ \text{return result-exn } c), \text{Normal } s \rangle = n \Rightarrow t \Longrightarrow P$   
**shows**  $P$   
 $\langle \text{proof} \rangle$

**lemma** *execn-Seq'*:

$\llbracket \Gamma \vdash \langle c1, s \rangle = n \Rightarrow s'; \Gamma \vdash \langle c2, s' \rangle = n \Rightarrow s'' \rrbracket$   
 $\Longrightarrow$   
 $\Gamma \vdash \langle \text{Seq } c1 \ c2, s \rangle = n \Rightarrow s''$   
 $\langle \text{proof} \rangle$

**lemma** *execn-mono*:

**assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
**shows**  $\bigwedge m. n \leq m \Longrightarrow \Gamma \vdash \langle c, s \rangle = m \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *execn-Suc*:

$\Gamma \vdash \langle c, s \rangle = n \Rightarrow t \implies \Gamma \vdash \langle c, s \rangle = \text{Suc } n \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *execn-assoc*:

$\Gamma \vdash \langle \text{Seq } c1 (\text{Seq } c2 c3), s \rangle = n \Rightarrow t = \Gamma \vdash \langle \text{Seq } (\text{Seq } c1 c2) c3, s \rangle = n \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *execn-to-exec*:

**assumes** *execn*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
**shows**  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *exec-to-execn*:

**assumes** *execn*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
**shows**  $\exists n. \Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
 $\langle \text{proof} \rangle$

**theorem** *exec-iff-execn*:  $(\Gamma \vdash \langle c, s \rangle \Rightarrow t) = (\exists n. \Gamma \vdash \langle c, s \rangle = n \Rightarrow t)$   
 $\langle \text{proof} \rangle$

**definition** *nfinal-notin*::  $(s, p, f) \text{ body} \Rightarrow (s, p, f) \text{ com} \Rightarrow (s, f) \text{ xstate} \Rightarrow \text{nat}$   
 $\Rightarrow (s, f) \text{ xstate set} \Rightarrow \text{bool}$

$(\langle + \langle -, - \rangle \Rightarrow \notin \rightarrow [60, 20, 98, 65, 60] 89) \text{ where}$   
 $\Gamma \vdash \langle c, s \rangle = n \Rightarrow \notin T = (\forall t. \Gamma \vdash \langle c, s \rangle = n \Rightarrow t \rightarrow t \notin T)$

**definition** *final-notin*::  $(s, p, f) \text{ body} \Rightarrow (s, p, f) \text{ com} \Rightarrow (s, f) \text{ xstate}$   
 $\Rightarrow (s, f) \text{ xstate set} \Rightarrow \text{bool}$

$(\langle + \langle -, - \rangle \Rightarrow \notin \rightarrow [60, 20, 98, 60] 89) \text{ where}$   
 $\Gamma \vdash \langle c, s \rangle \Rightarrow \notin T = (\forall t. \Gamma \vdash \langle c, s \rangle \Rightarrow t \rightarrow t \notin T)$

**lemma** *final-notinI*:  $\llbracket \bigwedge t. \Gamma \vdash \langle c, s \rangle \Rightarrow t \implies t \notin T \rrbracket \implies \Gamma \vdash \langle c, s \rangle \Rightarrow \notin T$   
 $\langle \text{proof} \rangle$

**lemma** *noFaultStuck-Call-body'*:  $p \in \text{dom } \Gamma \implies$

$\Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle \Rightarrow \notin (\{\text{Stuck}\} \cup \text{Fault } '(-F)) =$   
 $\Gamma \vdash \langle \text{the } (\Gamma \text{ } p), \text{Normal } s \rangle \Rightarrow \notin (\{\text{Stuck}\} \cup \text{Fault } '(-F))$   
 $\langle \text{proof} \rangle$

**lemma** *noFault-startn*:

**assumes** *execn*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$  **and**  $t: t \neq \text{Fault } f$   
**shows**  $s \neq \text{Fault } f$   
 $\langle \text{proof} \rangle$

**lemma** *noFault-start*:

**assumes**  $exec: \Gamma \vdash \langle c, s \rangle \Rightarrow t$  **and**  $t: t \neq Fault\ f$   
**shows**  $s \neq Fault\ f$   
 $\langle proof \rangle$

**lemma** *noStuck-startn*:  
**assumes**  $execn: \Gamma \vdash \langle c, s \rangle = n \Rightarrow t$  **and**  $t: t \neq Stuck$   
**shows**  $s \neq Stuck$   
 $\langle proof \rangle$

**lemma** *noStuck-start*:  
**assumes**  $exec: \Gamma \vdash \langle c, s \rangle \Rightarrow t$  **and**  $t: t \neq Stuck$   
**shows**  $s \neq Stuck$   
 $\langle proof \rangle$

**lemma** *noAbrupt-startn*:  
**assumes**  $execn: \Gamma \vdash \langle c, s \rangle = n \Rightarrow t$  **and**  $t: \forall t'. t \neq Abrupt\ t'$   
**shows**  $s \neq Abrupt\ s'$   
 $\langle proof \rangle$

**lemma** *noAbrupt-start*:  
**assumes**  $exec: \Gamma \vdash \langle c, s \rangle \Rightarrow t$  **and**  $t: \forall t'. t \neq Abrupt\ t'$   
**shows**  $s \neq Abrupt\ s'$   
 $\langle proof \rangle$

**lemma** *noFaultn-startD*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow Normal\ t \Longrightarrow s \neq Fault\ f$   
 $\langle proof \rangle$

**lemma** *noFaultn-startD'*:  $t \neq Fault\ f \Longrightarrow \Gamma \vdash \langle c, s \rangle = n \Rightarrow t \Longrightarrow s \neq Fault\ f$   
 $\langle proof \rangle$

**lemma** *noFault-startD*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow Normal\ t \Longrightarrow s \neq Fault\ f$   
 $\langle proof \rangle$

**lemma** *noFault-startD'*:  $t \neq Fault\ f \Longrightarrow \Gamma \vdash \langle c, s \rangle \Rightarrow t \Longrightarrow s \neq Fault\ f$   
 $\langle proof \rangle$

**lemma** *noStuckn-startD*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow Normal\ t \Longrightarrow s \neq Stuck$   
 $\langle proof \rangle$

**lemma** *noStuckn-startD'*:  $t \neq Stuck \Longrightarrow \Gamma \vdash \langle c, s \rangle = n \Rightarrow t \Longrightarrow s \neq Stuck$   
 $\langle proof \rangle$

**lemma** *noStuck-startD*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow Normal\ t \Longrightarrow s \neq Stuck$   
 $\langle proof \rangle$

**lemma** *noStuck-startD'*:  $t \neq Stuck \Longrightarrow \Gamma \vdash \langle c, s \rangle \Rightarrow t \Longrightarrow s \neq Stuck$   
 $\langle proof \rangle$

**lemma** *noAbruptn-startD*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow Normal\ t \Longrightarrow s \neq Abrupt\ s'$

*<proof>*

**lemma** *noAbrupt-startD*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow \text{Normal } t \Longrightarrow s \neq \text{Abrupt } s'$   
*<proof>*

**lemma** *noFaultnI*:  $\llbracket \bigwedge t. \Gamma \vdash \langle c, s \rangle = n \Rightarrow t \Longrightarrow t \neq \text{Fault } f \rrbracket \Longrightarrow \Gamma \vdash \langle c, s \rangle = n \Rightarrow \notin \{ \text{Fault } f \}$   
*<proof>*

**lemma** *noFaultnI'*:  
**assumes** *contr*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow \text{Fault } f \Longrightarrow \text{False}$   
**shows**  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow \notin \{ \text{Fault } f \}$   
*<proof>*

**lemma** *noFaultn-def'*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow \notin \{ \text{Fault } f \} = (\neg \Gamma \vdash \langle c, s \rangle = n \Rightarrow \text{Fault } f)$   
*<proof>*

**lemma** *noStucknI*:  $\llbracket \bigwedge t. \Gamma \vdash \langle c, s \rangle = n \Rightarrow t \Longrightarrow t \neq \text{Stuck} \rrbracket \Longrightarrow \Gamma \vdash \langle c, s \rangle = n \Rightarrow \notin \{ \text{Stuck} \}$   
*<proof>*

**lemma** *noStucknI'*:  
**assumes** *contr*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow \text{Stuck} \Longrightarrow \text{False}$   
**shows**  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow \notin \{ \text{Stuck} \}$   
*<proof>*

**lemma** *noStuckn-def'*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow \notin \{ \text{Stuck} \} = (\neg \Gamma \vdash \langle c, s \rangle = n \Rightarrow \text{Stuck})$   
*<proof>*

**lemma** *noFaultI*:  $\llbracket \bigwedge t. \Gamma \vdash \langle c, s \rangle \Rightarrow t \Longrightarrow t \neq \text{Fault } f \rrbracket \Longrightarrow \Gamma \vdash \langle c, s \rangle \Rightarrow \notin \{ \text{Fault } f \}$   
*<proof>*

**lemma** *noFaultI'*:  
**assumes** *contr*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow \text{Fault } f \Longrightarrow \text{False}$   
**shows**  $\Gamma \vdash \langle c, s \rangle \Rightarrow \notin \{ \text{Fault } f \}$   
*<proof>*

**lemma** *noFaultE*:  
 $\llbracket \Gamma \vdash \langle c, s \rangle \Rightarrow \notin \{ \text{Fault } f \}; \Gamma \vdash \langle c, s \rangle \Rightarrow \text{Fault } f \rrbracket \Longrightarrow P$   
*<proof>*

**lemma** *noFault-def'*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow \notin \{ \text{Fault } f \} = (\neg \Gamma \vdash \langle c, s \rangle \Rightarrow \text{Fault } f)$   
*<proof>*

**lemma** *noStuckI*:  $\llbracket \bigwedge t. \Gamma \vdash \langle c, s \rangle \Rightarrow t \Longrightarrow t \neq \text{Stuck} \rrbracket \Longrightarrow \Gamma \vdash \langle c, s \rangle \Rightarrow \notin \{ \text{Stuck} \}$   
*<proof>*

**lemma** *noStuckI'*:

**assumes** *contr*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow Stuck \Longrightarrow False$   
**shows**  $\Gamma \vdash \langle c, s \rangle \Rightarrow \notin \{Stuck\}$   
*<proof>*

**lemma** *noStuckE*:  
 $\llbracket \Gamma \vdash \langle c, s \rangle \Rightarrow \notin \{Stuck\}; \Gamma \vdash \langle c, s \rangle \Rightarrow Stuck \rrbracket \Longrightarrow P$   
*<proof>*

**lemma** *noStuck-def'*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow \notin \{Stuck\} = (\neg \Gamma \vdash \langle c, s \rangle \Rightarrow Stuck)$   
*<proof>*

**lemma** *noFaultn-execD*:  $\llbracket \Gamma \vdash \langle c, s \rangle = n \Rightarrow \notin \{Fault\ f\}; \Gamma \vdash \langle c, s \rangle = n \Rightarrow t \rrbracket \Longrightarrow t \neq Fault\ f$   
*<proof>*

**lemma** *noFault-execD*:  $\llbracket \Gamma \vdash \langle c, s \rangle \Rightarrow \notin \{Fault\ f\}; \Gamma \vdash \langle c, s \rangle \Rightarrow t \rrbracket \Longrightarrow t \neq Fault\ f$   
*<proof>*

**lemma** *noFaultn-exec-startD*:  $\llbracket \Gamma \vdash \langle c, s \rangle = n \Rightarrow \notin \{Fault\ f\}; \Gamma \vdash \langle c, s \rangle = n \Rightarrow t \rrbracket \Longrightarrow s \neq Fault\ f$   
*<proof>*

**lemma** *noFault-exec-startD*:  $\llbracket \Gamma \vdash \langle c, s \rangle \Rightarrow \notin \{Fault\ f\}; \Gamma \vdash \langle c, s \rangle \Rightarrow t \rrbracket \Longrightarrow s \neq Fault\ f$   
*<proof>*

**lemma** *noStuckn-execD*:  $\llbracket \Gamma \vdash \langle c, s \rangle = n \Rightarrow \notin \{Stuck\}; \Gamma \vdash \langle c, s \rangle = n \Rightarrow t \rrbracket \Longrightarrow t \neq Stuck$   
*<proof>*

**lemma** *noStuck-execD*:  $\llbracket \Gamma \vdash \langle c, s \rangle \Rightarrow \notin \{Stuck\}; \Gamma \vdash \langle c, s \rangle \Rightarrow t \rrbracket \Longrightarrow t \neq Stuck$   
*<proof>*

**lemma** *noStuckn-exec-startD*:  $\llbracket \Gamma \vdash \langle c, s \rangle = n \Rightarrow \notin \{Stuck\}; \Gamma \vdash \langle c, s \rangle = n \Rightarrow t \rrbracket \Longrightarrow s \neq Stuck$   
*<proof>*

**lemma** *noStuck-exec-startD*:  $\llbracket \Gamma \vdash \langle c, s \rangle \Rightarrow \notin \{Stuck\}; \Gamma \vdash \langle c, s \rangle \Rightarrow t \rrbracket \Longrightarrow s \neq Stuck$   
*<proof>*

**lemma** *noFaultStuckn-execD*:  
 $\llbracket \Gamma \vdash \langle c, s \rangle = n \Rightarrow \notin \{Fault\ True, Fault\ False, Stuck\}; \Gamma \vdash \langle c, s \rangle = n \Rightarrow t \rrbracket \Longrightarrow$   
 $t \notin \{Fault\ True, Fault\ False, Stuck\}$   
*<proof>*

**lemma** *noFaultStuck-execD*:  $\llbracket \Gamma \vdash \langle c, s \rangle \Rightarrow \notin \{Fault\ True, Fault\ False, Stuck\}; \Gamma \vdash \langle c, s \rangle \Rightarrow t \rrbracket$   
 $\Longrightarrow t \notin \{Fault\ True, Fault\ False, Stuck\}$   
*<proof>*

**lemma** *noFaultStuckn-exec-startD*:  
 $\llbracket \Gamma \vdash \langle c, s \rangle = n \Rightarrow \notin \{Fault\ True, Fault\ False, Stuck\}; \Gamma \vdash \langle c, s \rangle = n \Rightarrow t \rrbracket$

$\implies s \notin \{Fault\ True, Fault\ False, Stuck\}$   
 $\langle proof \rangle$

**lemma** *noFaultStuck-exec-startD*:

$\llbracket \Gamma \vdash \langle c, s \rangle \Rightarrow \notin \{Fault\ True, Fault\ False, Stuck\}; \Gamma \vdash \langle c, s \rangle \Rightarrow t \rrbracket$   
 $\implies s \notin \{Fault\ True, Fault\ False, Stuck\}$   
 $\langle proof \rangle$

**lemma** *noStuck-Call*:

**assumes** *noStuck*:  $\Gamma \vdash \langle Call\ p, Normal\ s \rangle \Rightarrow \notin \{Stuck\}$   
**shows**  $p \in dom\ \Gamma$   
 $\langle proof \rangle$

**lemma** *Guard-noFaultStuckD*:

**assumes**  $\Gamma \vdash \langle Guard\ f\ g\ c, Normal\ s \rangle \Rightarrow \notin (\{Stuck\} \cup Fault\ '(-F))$   
**assumes**  $f \notin F$   
**shows**  $s \in g$   
 $\langle proof \rangle$

**lemma** *final-notin-to-finaln*:

**assumes** *notin*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow \notin T$   
**shows**  $\Gamma \vdash \langle c, s \rangle =n \Rightarrow \notin T$   
 $\langle proof \rangle$

**lemma** *noFault-Call-body*:

$\Gamma\ p = Some\ bdy \implies$   
 $\Gamma \vdash \langle Call\ p, Normal\ s \rangle \Rightarrow \notin \{Fault\ f\} =$   
 $\Gamma \vdash \langle the\ (\Gamma\ p), Normal\ s \rangle \Rightarrow \notin \{Fault\ f\}$   
 $\langle proof \rangle$

**lemma** *noStuck-Call-body*:

$\Gamma\ p = Some\ bdy \implies$   
 $\Gamma \vdash \langle Call\ p, Normal\ s \rangle \Rightarrow \notin \{Stuck\} =$   
 $\Gamma \vdash \langle the\ (\Gamma\ p), Normal\ s \rangle \Rightarrow \notin \{Stuck\}$   
 $\langle proof \rangle$

**lemma** *exec-final-notin-to-execn*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow \notin T \implies \Gamma \vdash \langle c, s \rangle =n \Rightarrow \notin T$   
 $\langle proof \rangle$

**lemma** *execn-final-notin-to-exec*:  $\forall n. \Gamma \vdash \langle c, s \rangle =n \Rightarrow \notin T \implies \Gamma \vdash \langle c, s \rangle \Rightarrow \notin T$   
 $\langle proof \rangle$

**lemma** *exec-final-notin-iff-execn*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow \notin T = (\forall n. \Gamma \vdash \langle c, s \rangle =n \Rightarrow \notin T)$   
 $\langle proof \rangle$

**lemma** *Seq-NoFaultStuckD2*:

**assumes** *noabort*:  $\Gamma \vdash \langle Seq\ c1\ c2, s \rangle \Rightarrow \notin (\{Stuck\} \cup Fault\ 'F)$

**shows**  $\forall t. \Gamma \vdash \langle c1, s \rangle \Rightarrow t \longrightarrow t \notin (\{Stuck\} \cup Fault \text{ ' } F) \longrightarrow$   
 $\Gamma \vdash \langle c2, t \rangle \Rightarrow \notin (\{Stuck\} \cup Fault \text{ ' } F)$   
 $\langle proof \rangle$  **lemma** *Seq-NoFaultStuckD1*:  
**assumes** *noabort*:  $\Gamma \vdash \langle Seq\ c1\ c2, s \rangle \Rightarrow \notin (\{Stuck\} \cup Fault \text{ ' } F)$   
**shows**  $\Gamma \vdash \langle c1, s \rangle \Rightarrow \notin (\{Stuck\} \cup Fault \text{ ' } F)$   
 $\langle proof \rangle$

**lemma** *Seq-NoFaultStuckD2'*:  
**assumes** *noabort*:  $\Gamma \vdash \langle Seq\ c1\ c2, s \rangle \Rightarrow \notin (\{Stuck\} \cup Fault \text{ ' } F)$   
**shows**  $\forall t. \Gamma \vdash \langle c1, s \rangle \Rightarrow t \longrightarrow t \notin (\{Stuck\} \cup Fault \text{ ' } F) \longrightarrow$   
 $\Gamma \vdash \langle c2, t \rangle \Rightarrow \notin (\{Stuck\} \cup Fault \text{ ' } F)$   
 $\langle proof \rangle$

### 3.3 Lemmas about *sequence*, *flatten* and *Language.normalize*

**lemma** *execn-sequence-app*:  $\bigwedge s\ s'\ t.$   
 $\llbracket \Gamma \vdash \langle sequence\ Seq\ xs, Normal\ s \rangle = n \Rightarrow s'; \Gamma \vdash \langle sequence\ Seq\ ys, s' \rangle = n \Rightarrow t \rrbracket$   
 $\implies \Gamma \vdash \langle sequence\ Seq\ (xs @ ys), Normal\ s \rangle = n \Rightarrow t$   
 $\langle proof \rangle$

**lemma** *execn-sequence-appD*:  $\bigwedge s\ t. \Gamma \vdash \langle sequence\ Seq\ (xs @ ys), Normal\ s \rangle = n \Rightarrow t$   
 $\implies$   
 $\exists s'. \Gamma \vdash \langle sequence\ Seq\ xs, Normal\ s \rangle = n \Rightarrow s' \wedge \Gamma \vdash \langle sequence\ Seq\ ys, s' \rangle = n \Rightarrow$   
 $t$   
 $\langle proof \rangle$

**lemma** *execn-sequence-appE* [*consumes 1*]:  
 $\llbracket \Gamma \vdash \langle sequence\ Seq\ (xs @ ys), Normal\ s \rangle = n \Rightarrow t;$   
 $\bigwedge s'. \llbracket \Gamma \vdash \langle sequence\ Seq\ xs, Normal\ s \rangle = n \Rightarrow s'; \Gamma \vdash \langle sequence\ Seq\ ys, s' \rangle = n \Rightarrow t \rrbracket \implies$   
 $P$   
 $\rrbracket \implies P$   
 $\langle proof \rangle$

**lemma** *execn-to-execn-sequence-flatten*:  
**assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
**shows**  $\Gamma \vdash \langle sequence\ Seq\ (flatten\ c), s \rangle = n \Rightarrow t$   
 $\langle proof \rangle$

**lemma** *execn-to-execn-normalize*:  
**assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
**shows**  $\Gamma \vdash \langle normalize\ c, s \rangle = n \Rightarrow t$   
 $\langle proof \rangle$

**lemma** *execn-sequence-flatten-to-execn*:  
**shows**  $\bigwedge s\ t. \Gamma \vdash \langle sequence\ Seq\ (flatten\ c), s \rangle = n \Rightarrow t \implies \Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
 $\langle proof \rangle$

**lemma** *execn-normalize-to-execn*:

**shows**  $\bigwedge s t n. \Gamma \vdash \langle \text{normalize } c, s \rangle = n \Rightarrow t \Longrightarrow \Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
*<proof>*

**lemma** *execn-normalize-iff-execn*:

$\Gamma \vdash \langle \text{normalize } c, s \rangle = n \Rightarrow t = \Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
*<proof>*

**lemma** *exec-sequence-app*:

**assumes** *exec-xs*:  $\Gamma \vdash \langle \text{sequence Seq } xs, \text{Normal } s \rangle \Rightarrow s'$   
**assumes** *exec-ys*:  $\Gamma \vdash \langle \text{sequence Seq } ys, s' \rangle \Rightarrow t$   
**shows**  $\Gamma \vdash \langle \text{sequence Seq } (xs @ ys), \text{Normal } s \rangle \Rightarrow t$   
*<proof>*

**lemma** *exec-sequence-appD*:

**assumes** *exec-xs-ys*:  $\Gamma \vdash \langle \text{sequence Seq } (xs @ ys), \text{Normal } s \rangle \Rightarrow t$   
**shows**  $\exists s'. \Gamma \vdash \langle \text{sequence Seq } xs, \text{Normal } s \rangle \Rightarrow s' \wedge \Gamma \vdash \langle \text{sequence Seq } ys, s' \rangle \Rightarrow t$   
*<proof>*

**lemma** *exec-sequence-appE* [consumes 1]:

$\llbracket \Gamma \vdash \langle \text{sequence Seq } (xs @ ys), \text{Normal } s \rangle \Rightarrow t; \bigwedge s'. \llbracket \Gamma \vdash \langle \text{sequence Seq } xs, \text{Normal } s \rangle \Rightarrow s'; \Gamma \vdash \langle \text{sequence Seq } ys, s' \rangle \Rightarrow t \rrbracket \Longrightarrow P \rrbracket \Longrightarrow P$   
*<proof>*

**lemma** *exec-to-exec-sequence-flatten*:

**assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
**shows**  $\Gamma \vdash \langle \text{sequence Seq } (\text{flatten } c), s \rangle \Rightarrow t$   
*<proof>*

**lemma** *exec-sequence-flatten-to-exec*:

**assumes** *exec-seq*:  $\Gamma \vdash \langle \text{sequence Seq } (\text{flatten } c), s \rangle \Rightarrow t$   
**shows**  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
*<proof>*

**lemma** *exec-to-exec-normalize*:

**assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
**shows**  $\Gamma \vdash \langle \text{normalize } c, s \rangle \Rightarrow t$   
*<proof>*

**lemma** *exec-normalize-to-exec*:

**assumes** *exec*:  $\Gamma \vdash \langle \text{normalize } c, s \rangle \Rightarrow t$   
**shows**  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
*<proof>*

**lemma** *exec-normalize-iff-exec*:

$\Gamma \vdash \langle \text{normalize } c, s \rangle \Rightarrow t = \Gamma \vdash \langle c, s \rangle \Rightarrow t$   
*<proof>*

### 3.4 Lemmas about $c_1 \subseteq_g c_2$

**lemma** *execn-to-execn-subseteq-guards*:  $\bigwedge c \ s \ t \ n. \llbracket c \subseteq_g c'; \Gamma \vdash \langle c, s \rangle = n \Rightarrow t \rrbracket$   
 $\implies \exists t'. \Gamma \vdash \langle c', s \rangle = n \Rightarrow t' \wedge$   
 $(\text{isFault } t \longrightarrow \text{isFault } t') \wedge (\neg \text{isFault } t' \longrightarrow t'=t)$   
*<proof>*

**lemma** *exec-to-exec-subseteq-guards*:  
**assumes**  $c-c': c \subseteq_g c'$   
**assumes**  $\text{exec}: \Gamma \vdash \langle c, s \rangle \Rightarrow t$   
**shows**  $\exists t'. \Gamma \vdash \langle c', s \rangle \Rightarrow t' \wedge$   
 $(\text{isFault } t \longrightarrow \text{isFault } t') \wedge (\neg \text{isFault } t' \longrightarrow t'=t)$   
*<proof>*

### 3.5 Lemmas about *merge-guards*

**theorem** *execn-to-execn-merge-guards*:  
**assumes**  $\text{exec-c}: \Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
**shows**  $\Gamma \vdash \langle \text{merge-guards } c, s \rangle = n \Rightarrow t$   
*<proof>*

**lemma** *execn-merge-guards-to-execn-Normal*:  
 $\bigwedge s \ n \ t. \Gamma \vdash \langle \text{merge-guards } c, \text{Normal } s \rangle = n \Rightarrow t \implies \Gamma \vdash \langle c, \text{Normal } s \rangle = n \Rightarrow t$   
*<proof>*

**theorem** *execn-merge-guards-to-execn*:  
 $\Gamma \vdash \langle \text{merge-guards } c, s \rangle = n \Rightarrow t \implies \Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
*<proof>*

**corollary** *execn-iff-execn-merge-guards*:  
 $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t = \Gamma \vdash \langle \text{merge-guards } c, s \rangle = n \Rightarrow t$   
*<proof>*

**theorem** *exec-iff-exec-merge-guards*:  
 $\Gamma \vdash \langle c, s \rangle \Rightarrow t = \Gamma \vdash \langle \text{merge-guards } c, s \rangle \Rightarrow t$   
*<proof>*

**corollary** *exec-to-exec-merge-guards*:  
 $\Gamma \vdash \langle c, s \rangle \Rightarrow t \implies \Gamma \vdash \langle \text{merge-guards } c, s \rangle \Rightarrow t$   
*<proof>*

**corollary** *exec-merge-guards-to-exec*:  
 $\Gamma \vdash \langle \text{merge-guards } c, s \rangle \Rightarrow t \implies \Gamma \vdash \langle c, s \rangle \Rightarrow t$   
*<proof>*

### 3.6 Lemmas about *mark-guards*

**lemma** *execn-to-execn-mark-guards*:  
**assumes**  $\text{exec-c}: \Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
**assumes**  $t\text{-not-Fault}: \neg \text{isFault } t$

**shows**  $\Gamma \vdash \langle \text{mark-guards } f \ c, s \rangle = n \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *execn-to-execn-mark-guards-Fault*:

**assumes** *exec-c*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
**shows**  $\bigwedge f. \llbracket t = \text{Fault } f \rrbracket \Longrightarrow \exists f'. \Gamma \vdash \langle \text{mark-guards } x \ c, s \rangle = n \Rightarrow \text{Fault } f'$   
 $\langle \text{proof} \rangle$

**lemma** *execn-mark-guards-to-execn*:

$\bigwedge s \ n \ t. \Gamma \vdash \langle \text{mark-guards } f \ c, s \rangle = n \Rightarrow t$   
 $\Longrightarrow \exists t'. \Gamma \vdash \langle c, s \rangle = n \Rightarrow t' \wedge$   
 $(\text{isFault } t \longrightarrow \text{isFault } t') \wedge$   
 $(t' = \text{Fault } f \longrightarrow t' = t) \wedge$   
 $(\text{isFault } t' \longrightarrow \text{isFault } t) \wedge$   
 $(\neg \text{isFault } t' \longrightarrow t' = t)$   
 $\langle \text{proof} \rangle$

**lemma** *exec-to-exec-mark-guards*:

**assumes** *exec-c*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
**assumes** *t-not-Fault*:  $\neg \text{isFault } t$   
**shows**  $\Gamma \vdash \langle \text{mark-guards } f \ c, s \rangle \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *exec-to-exec-mark-guards-Fault*:

**assumes** *exec-c*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow \text{Fault } f$   
**shows**  $\exists f'. \Gamma \vdash \langle \text{mark-guards } x \ c, s \rangle \Rightarrow \text{Fault } f'$   
 $\langle \text{proof} \rangle$

**lemma** *exec-mark-guards-to-exec*:

**assumes** *exec-mark*:  $\Gamma \vdash \langle \text{mark-guards } f \ c, s \rangle \Rightarrow t$   
**shows**  $\exists t'. \Gamma \vdash \langle c, s \rangle \Rightarrow t' \wedge$   
 $(\text{isFault } t \longrightarrow \text{isFault } t') \wedge$   
 $(t' = \text{Fault } f \longrightarrow t' = t) \wedge$   
 $(\text{isFault } t' \longrightarrow \text{isFault } t) \wedge$   
 $(\neg \text{isFault } t' \longrightarrow t' = t)$   
 $\langle \text{proof} \rangle$

### 3.7 Lemmas about *strip-guards*

**lemma** *execn-to-execn-strip-guards*:

**assumes** *exec-c*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
**assumes** *t-not-Fault*:  $\neg \text{isFault } t$   
**shows**  $\Gamma \vdash \langle \text{strip-guards } F \ c, s \rangle = n \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *execn-to-execn-strip-guards-Fault*:

**assumes** *exec-c*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$

**shows**  $\bigwedge f. \llbracket t = \text{Fault } f; f \notin F \rrbracket \implies \Gamma \vdash \langle \text{strip-guards } F \ c, s \rangle = n \Rightarrow \text{Fault } f$   
 <proof>

**lemma** *execn-to-execn-strip-guards'*:  
**assumes** *exec-c*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
**assumes** *t-not-Fault*:  $t \notin \text{Fault } ' F$   
**shows**  $\Gamma \vdash \langle \text{strip-guards } F \ c, s \rangle = n \Rightarrow t$   
 <proof>

**lemma** *execn-strip-guards-to-execn*:  
 $\bigwedge s \ n \ t. \Gamma \vdash \langle \text{strip-guards } F \ c, s \rangle = n \Rightarrow t$   
 $\implies \exists t'. \Gamma \vdash \langle c, s \rangle = n \Rightarrow t' \wedge$   
 $(\text{isFault } t \longrightarrow \text{isFault } t') \wedge$   
 $(t' \in \text{Fault } ' (- F) \longrightarrow t' = t) \wedge$   
 $(\neg \text{isFault } t' \longrightarrow t' = t)$   
 <proof>

**lemma** *execn-strip-to-execn*:  
**assumes** *exec-strip*:  $\text{strip } F \ \Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
**shows**  $\exists t'. \Gamma \vdash \langle c, s \rangle = n \Rightarrow t' \wedge$   
 $(\text{isFault } t \longrightarrow \text{isFault } t') \wedge$   
 $(t' \in \text{Fault } ' (- F) \longrightarrow t' = t) \wedge$   
 $(\neg \text{isFault } t' \longrightarrow t' = t)$   
 <proof>

**lemma** *exec-strip-guards-to-exec*:  
**assumes** *exec-strip*:  $\Gamma \vdash \langle \text{strip-guards } F \ c, s \rangle \Rightarrow t$   
**shows**  $\exists t'. \Gamma \vdash \langle c, s \rangle \Rightarrow t' \wedge$   
 $(\text{isFault } t \longrightarrow \text{isFault } t') \wedge$   
 $(t' \in \text{Fault } ' (- F) \longrightarrow t' = t) \wedge$   
 $(\neg \text{isFault } t' \longrightarrow t' = t)$   
 <proof>

**lemma** *exec-strip-to-exec*:  
**assumes** *exec-strip*:  $\text{strip } F \ \Gamma \vdash \langle c, s \rangle \Rightarrow t$   
**shows**  $\exists t'. \Gamma \vdash \langle c, s \rangle \Rightarrow t' \wedge$   
 $(\text{isFault } t \longrightarrow \text{isFault } t') \wedge$   
 $(t' \in \text{Fault } ' (- F) \longrightarrow t' = t) \wedge$   
 $(\neg \text{isFault } t' \longrightarrow t' = t)$   
 <proof>

**lemma** *exec-to-exec-strip-guards*:  
**assumes** *exec-c*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
**assumes** *t-not-Fault*:  $\neg \text{isFault } t$   
**shows**  $\Gamma \vdash \langle \text{strip-guards } F \ c, s \rangle \Rightarrow t$   
 <proof>

**lemma** *exec-to-exec-strip-guards'*:  
**assumes** *exec-c*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
**assumes** *t-not-Fault*:  $t \notin \text{Fault} \text{ ' } F$   
**shows**  $\Gamma \vdash \langle \text{strip-guards } F \ c, s \rangle \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *execn-to-execn-strip*:  
**assumes** *exec-c*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
**assumes** *t-not-Fault*:  $\neg \text{isFault } t$   
**shows**  $\text{strip } F \ \Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *execn-to-execn-strip'*:  
**assumes** *exec-c*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
**assumes** *t-not-Fault*:  $t \notin \text{Fault} \text{ ' } F$   
**shows**  $\text{strip } F \ \Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *exec-to-exec-strip*:  
**assumes** *exec-c*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
**assumes** *t-not-Fault*:  $\neg \text{isFault } t$   
**shows**  $\text{strip } F \ \Gamma \vdash \langle c, s \rangle \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *exec-to-exec-strip'*:  
**assumes** *exec-c*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
**assumes** *t-not-Fault*:  $t \notin \text{Fault} \text{ ' } F$   
**shows**  $\text{strip } F \ \Gamma \vdash \langle c, s \rangle \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *exec-to-exec-strip-guards-Fault*:  
**assumes** *exec-c*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow \text{Fault } f$   
**assumes** *f-notin-F*:  $f \notin F$   
**shows**  $\Gamma \vdash \langle \text{strip-guards } F \ c, s \rangle \Rightarrow \text{Fault } f$   
 $\langle \text{proof} \rangle$

### 3.8 Lemmas about $c_1 \cap_g c_2$

**lemma** *inter-guards-execn-Normal-noFault*:  
 $\bigwedge c \ c_2 \ s \ t \ n. \llbracket (c_1 \cap_g c_2) = \text{Some } c; \Gamma \vdash \langle c, \text{Normal } s \rangle = n \Rightarrow t; \neg \text{isFault } t \rrbracket$   
 $\implies \Gamma \vdash \langle c_1, \text{Normal } s \rangle = n \Rightarrow t \wedge \Gamma \vdash \langle c_2, \text{Normal } s \rangle = n \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *inter-guards-execn-noFault*:  
**assumes** *c*:  $(c_1 \cap_g c_2) = \text{Some } c$   
**assumes** *exec-c*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
**assumes** *noFault*:  $\neg \text{isFault } t$   
**shows**  $\Gamma \vdash \langle c_1, s \rangle = n \Rightarrow t \wedge \Gamma \vdash \langle c_2, s \rangle = n \Rightarrow t$

$\langle proof \rangle$

**lemma** *inter-guards-exec-noFault*:

**assumes**  $c: (c1 \cap_g c2) = \text{Some } c$

**assumes**  $exec\text{-}c: \Gamma \vdash \langle c, s \rangle \Rightarrow t$

**assumes**  $noFault: \neg isFault\ t$

**shows**  $\Gamma \vdash \langle c1, s \rangle \Rightarrow t \wedge \Gamma \vdash \langle c2, s \rangle \Rightarrow t$

$\langle proof \rangle$

**lemma** *inter-guards-execn-Normal-Fault*:

$\bigwedge c\ c2\ s\ n. \llbracket (c1 \cap_g c2) = \text{Some } c; \Gamma \vdash \langle c, Normal\ s \rangle = n \Rightarrow Fault\ f \rrbracket$

$\implies (\Gamma \vdash \langle c1, Normal\ s \rangle = n \Rightarrow Fault\ f \vee \Gamma \vdash \langle c2, Normal\ s \rangle = n \Rightarrow Fault\ f)$

$\langle proof \rangle$

**lemma** *inter-guards-execn-Fault*:

**assumes**  $c: (c1 \cap_g c2) = \text{Some } c$

**assumes**  $exec\text{-}c: \Gamma \vdash \langle c, s \rangle = n \Rightarrow Fault\ f$

**shows**  $\Gamma \vdash \langle c1, s \rangle = n \Rightarrow Fault\ f \vee \Gamma \vdash \langle c2, s \rangle = n \Rightarrow Fault\ f$

$\langle proof \rangle$

**lemma** *inter-guards-exec-Fault*:

**assumes**  $c: (c1 \cap_g c2) = \text{Some } c$

**assumes**  $exec\text{-}c: \Gamma \vdash \langle c, s \rangle \Rightarrow Fault\ f$

**shows**  $\Gamma \vdash \langle c1, s \rangle \Rightarrow Fault\ f \vee \Gamma \vdash \langle c2, s \rangle \Rightarrow Fault\ f$

$\langle proof \rangle$

### 3.9 Restriction of Procedure Environment

**lemma** *restrict-SomeD*:  $(m|_A)\ x = \text{Some } y \implies m\ x = \text{Some } y$

$\langle proof \rangle$

**lemma** *restrict-dom-same* [*simp*]:  $m|_{dom\ m} = m$

$\langle proof \rangle$

**lemma** *restrict-in-dom*:  $x \in A \implies (m|_A)\ x = m\ x$

$\langle proof \rangle$

**lemma** *exec-restrict-to-exec*:

**assumes**  $exec\text{-}restrict: \Gamma|_A \vdash \langle c, s \rangle \Rightarrow t$

**assumes**  $notStuck: t \neq Stuck$

**shows**  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$

$\langle proof \rangle$

**lemma** *execn-restrict-to-execn*:

**assumes**  $exec\text{-}restrict: \Gamma|_A \vdash \langle c, s \rangle = n \Rightarrow t$

**assumes** *notStuck*:  $t \neq \text{Stuck}$   
**shows**  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
 $\langle \text{proof} \rangle$

**lemma** *restrict-NoneD*:  $m\ x = \text{None} \Longrightarrow (m|_A)\ x = \text{None}$   
 $\langle \text{proof} \rangle$

**lemma** *execn-to-execn-restrict*:  
**assumes** *execn*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
**shows**  $\exists t'. \Gamma \vdash \langle c, s \rangle = n \Rightarrow t' \wedge (t = \text{Stuck} \longrightarrow t' = \text{Stuck}) \wedge$   
 $(\forall f. t = \text{Fault } f \longrightarrow t' \in \{\text{Fault } f, \text{Stuck}\}) \wedge (t' \neq \text{Stuck} \longrightarrow t' = t)$   
 $\langle \text{proof} \rangle$

**lemma** *exec-to-exec-restrict*:  
**assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
**shows**  $\exists t'. \Gamma \vdash \langle c, s \rangle \Rightarrow t' \wedge (t = \text{Stuck} \longrightarrow t' = \text{Stuck}) \wedge$   
 $(\forall f. t = \text{Fault } f \longrightarrow t' \in \{\text{Fault } f, \text{Stuck}\}) \wedge (t' \neq \text{Stuck} \longrightarrow t' = t)$   
 $\langle \text{proof} \rangle$

**lemma** *notStuck-GuardD*:  
 $\llbracket \Gamma \vdash \langle \text{Guard } m\ g\ c, \text{Normal } s \rangle \Rightarrow \notin \{\text{Stuck}\}; s \in g \rrbracket \Longrightarrow \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \notin \{\text{Stuck}\}$   
 $\langle \text{proof} \rangle$

**lemma** *notStuck-SeqD1*:  
 $\llbracket \Gamma \vdash \langle \text{Seq } c1\ c2, \text{Normal } s \rangle \Rightarrow \notin \{\text{Stuck}\} \rrbracket \Longrightarrow \Gamma \vdash \langle c1, \text{Normal } s \rangle \Rightarrow \notin \{\text{Stuck}\}$   
 $\langle \text{proof} \rangle$

**lemma** *notStuck-SeqD2*:  
 $\llbracket \Gamma \vdash \langle \text{Seq } c1\ c2, \text{Normal } s \rangle \Rightarrow \notin \{\text{Stuck}\}; \Gamma \vdash \langle c1, \text{Normal } s \rangle \Rightarrow s' \rrbracket \Longrightarrow \Gamma \vdash \langle c2, s' \rangle$   
 $\Rightarrow \notin \{\text{Stuck}\}$   
 $\langle \text{proof} \rangle$

**lemma** *notStuck-SeqD*:  
 $\llbracket \Gamma \vdash \langle \text{Seq } c1\ c2, \text{Normal } s \rangle \Rightarrow \notin \{\text{Stuck}\} \rrbracket \Longrightarrow$   
 $\Gamma \vdash \langle c1, \text{Normal } s \rangle \Rightarrow \notin \{\text{Stuck}\} \wedge (\forall s'. \Gamma \vdash \langle c1, \text{Normal } s \rangle \Rightarrow s' \longrightarrow \Gamma \vdash \langle c2, s' \rangle$   
 $\Rightarrow \notin \{\text{Stuck}\})$   
 $\langle \text{proof} \rangle$

**lemma** *notStuck-CondTrueD*:  
 $\llbracket \Gamma \vdash \langle \text{Cond } b\ c1\ c2, \text{Normal } s \rangle \Rightarrow \notin \{\text{Stuck}\}; s \in b \rrbracket \Longrightarrow \Gamma \vdash \langle c1, \text{Normal } s \rangle \Rightarrow \notin \{\text{Stuck}\}$   
 $\langle \text{proof} \rangle$

**lemma** *notStuck-CondFalseD*:  
 $\llbracket \Gamma \vdash \langle \text{Cond } b\ c1\ c2, \text{Normal } s \rangle \Rightarrow \notin \{\text{Stuck}\}; s \notin b \rrbracket \Longrightarrow \Gamma \vdash \langle c2, \text{Normal } s \rangle \Rightarrow \notin \{\text{Stuck}\}$   
 $\langle \text{proof} \rangle$

**lemma** *notStuck-WhileTrueD1*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle \text{While } b \ c, \text{Normal } s \rangle \Rightarrow \notin \{ \text{Stuck} \}; s \in b \rrbracket \\ & \implies \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \notin \{ \text{Stuck} \} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *notStuck-WhileTrueD2*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle \text{While } b \ c, \text{Normal } s \rangle \Rightarrow \notin \{ \text{Stuck} \}; \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow s'; s \in b \rrbracket \\ & \implies \Gamma \vdash \langle \text{While } b \ c, s' \rangle \Rightarrow \notin \{ \text{Stuck} \} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *notStuck-CallD*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle \text{Call } p \ , \text{Normal } s \rangle \Rightarrow \notin \{ \text{Stuck} \}; \Gamma \ p = \text{Some } \text{bdy} \rrbracket \\ & \implies \Gamma \vdash \langle \text{bdy}, \text{Normal } s \rangle \Rightarrow \notin \{ \text{Stuck} \} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *notStuck-CallDefinedD*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle \Rightarrow \notin \{ \text{Stuck} \} \rrbracket \\ & \implies \Gamma \ p \neq \text{None} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *notStuck-DynComD*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle \text{DynCom } c, \text{Normal } s \rangle \Rightarrow \notin \{ \text{Stuck} \} \rrbracket \\ & \implies \Gamma \vdash \langle (c \ s), \text{Normal } s \rangle \Rightarrow \notin \{ \text{Stuck} \} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *notStuck-CatchD1*:

$$\llbracket \Gamma \vdash \langle \text{Catch } c1 \ c2, \text{Normal } s \rangle \Rightarrow \notin \{ \text{Stuck} \} \rrbracket \implies \Gamma \vdash \langle c1, \text{Normal } s \rangle \Rightarrow \notin \{ \text{Stuck} \}$$
  
 $\langle \text{proof} \rangle$

**lemma** *notStuck-CatchD2*:

$$\begin{aligned} & \llbracket \Gamma \vdash \langle \text{Catch } c1 \ c2, \text{Normal } s \rangle \Rightarrow \notin \{ \text{Stuck} \}; \Gamma \vdash \langle c1, \text{Normal } s \rangle \Rightarrow \text{Abrupt } s' \rrbracket \\ & \implies \Gamma \vdash \langle c2, \text{Normal } s' \rangle \Rightarrow \notin \{ \text{Stuck} \} \\ & \langle \text{proof} \rangle \end{aligned}$$

### 3.10 Miscellaneous

**lemma** *execn-noguards-no-Fault*:

**assumes** *execn*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$   
**assumes** *noguards-c*: *noguards*  $c$   
**assumes** *noguards- $\Gamma$* :  $\forall p \in \text{dom } \Gamma. \text{noguards } (\text{the } (\Gamma \ p))$   
**assumes** *s-no-Fault*:  $\neg \text{isFault } s$   
**shows**  $\neg \text{isFault } t$   
 $\langle \text{proof} \rangle$

**lemma** *exec-noguards-no-Fault*:

**assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
**assumes** *noguards-c*: *noguards*  $c$   
**assumes** *noguards- $\Gamma$* :  $\forall p \in \text{dom } \Gamma. \text{noguards } (\text{the } (\Gamma \ p))$   
**assumes** *s-no-Fault*:  $\neg \text{isFault } s$   
**shows**  $\neg \text{isFault } t$

$\langle proof \rangle$

**lemma** *execn-nothrows-no-Abrupt*:

**assumes** *execn*:  $\Gamma \vdash \langle c, s \rangle = n \Rightarrow t$

**assumes** *nothrows-c*: *nothrows c*

**assumes** *nothrows- $\Gamma$* :  $\forall p \in \text{dom } \Gamma. \text{nothrows } (the (\Gamma p))$

**assumes** *s-no-Abrupt*:  $\neg(isAbr s)$

**shows**  $\neg(isAbr t)$

$\langle proof \rangle$

**lemma** *exec-nothrows-no-Abrupt*:

**assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$

**assumes** *nothrows-c*: *nothrows c*

**assumes** *nothrows- $\Gamma$* :  $\forall p \in \text{dom } \Gamma. \text{nothrows } (the (\Gamma p))$

**assumes** *s-no-Abrupt*:  $\neg(isAbr s)$

**shows**  $\neg(isAbr t)$

$\langle proof \rangle$

**end**

## 4 Hoare Logic for Partial Correctness

**theory** *HoarePartialDef* **imports** *Semantic* **begin**

**type-synonym**  $(\prime s, \prime p)$  *quadruple* =  $(\prime s \text{ assn} \times \prime p \times \prime s \text{ assn} \times \prime s \text{ assn})$

### 4.1 Validity of Hoare Tuples: $\Gamma, \Theta \models_F P \ c \ Q, A$

**definition**

*valid* ::  $[(\prime s, \prime p, \prime f) \text{ body}, \prime f \text{ set}, \prime s \text{ assn}, (\prime s, \prime p, \prime f) \text{ com}, \prime s \text{ assn}, \prime s \text{ assn}] \Rightarrow \text{bool}$   
 $(\langle \cdot, \cdot \rangle \models_F / - - \cdot, \cdot \rightarrow [61, 60, 1000, 20, 1000, 1000] 60)$

**where**

$\Gamma \models_F P \ c \ Q, A \equiv \forall s \ t. \Gamma \vdash \langle c, s \rangle \Rightarrow t \longrightarrow s \in \text{Normal } \prime P \longrightarrow t \notin \text{Fault } \prime F$   
 $\longrightarrow t \in \text{Normal } \prime Q \cup \text{Abrupt } \prime A$

**definition**

*cvalid*:

$[(\prime s, \prime p, \prime f) \text{ body}, (\prime s, \prime p) \text{ quadruple set}, \prime f \text{ set},$   
 $\prime s \text{ assn}, (\prime s, \prime p, \prime f) \text{ com}, \prime s \text{ assn}, \prime s \text{ assn}] \Rightarrow \text{bool}$   
 $(\langle \cdot, \cdot \rangle \models_F / - - \cdot, \cdot \rightarrow [61, 60, 60, 1000, 20, 1000, 1000] 60)$

**where**

$\Gamma, \Theta \models_F P \ c \ Q, A \equiv (\forall (P, p, Q, A) \in \Theta. \Gamma \models_F P \ (Call \ p) \ Q, A) \longrightarrow \Gamma \models_F P \ c \ Q, A$

**definition**

*nvalid* ::  $[(\prime s, \prime p, \prime f) \text{ body}, \text{nat}, \prime f \text{ set},$   
 $\prime s \text{ assn}, (\prime s, \prime p, \prime f) \text{ com}, \prime s \text{ assn}, \prime s \text{ assn}] \Rightarrow \text{bool}$   
 $(\langle \cdot, \cdot \rangle \models_F / - - \cdot, \cdot \rightarrow [61, 60, 60, 1000, 20, 1000, 1000] 60)$

**where**

$$\Gamma \models_n: /_F P \text{ c } Q, A \equiv \forall s t. \Gamma \vdash \langle c, s \rangle =_n \Rightarrow t \longrightarrow s \in \text{Normal} \text{ ' } P \longrightarrow t \notin \text{Fault} \text{ ' } F \\ \longrightarrow t \in \text{Normal} \text{ ' } Q \cup \text{Abrupt} \text{ ' } A$$

**definition**

*cvalid*:

$$[(\text{'s, 'p, 'f} \text{ body, 's, 'p} \text{ quadruple set, nat, 'f set,} \\ \text{'s assn, ('s, 'p, 'f) com, 's assn, 's assn}] \Rightarrow \text{bool} \\ (\langle -, - \rangle \models_n: /_F - - - \rangle [61, 60, 60, 60, 1000, 20, 1000, 1000] 60)$$

**where**

$$\Gamma, \Theta \models_n: /_F P \text{ c } Q, A \equiv (\forall (P, p, Q, A) \in \Theta. \Gamma \models_n: /_F P (\text{Call } p) Q, A) \longrightarrow \Gamma \models_n: /_F P \\ \text{ c } Q, A$$

**notation (ASCII)**

$$\text{valid} (\langle -, - \rangle \models_n: /_F - - - \rangle [61, 60, 1000, 20, 1000, 1000] 60) \text{ and} \\ \text{cvalid} (\langle -, - \rangle \models_n: /_F - - - \rangle [61, 60, 60, 1000, 20, 1000, 1000] 60) \text{ and} \\ \text{nvalid} (\langle -, - \rangle \models_n: /_F - - - \rangle [61, 60, 60, 1000, 20, 1000, 1000] 60) \text{ and} \\ \text{cnvalid} (\langle -, - \rangle \models_n: /_F - - - \rangle [61, 60, 60, 60, 1000, 20, 1000, 1000] 60)$$

## 4.2 Properties of Validity

**lemma valid-iff-nvalid:**  $\Gamma \models_n: /_F P \text{ c } Q, A = (\forall n. \Gamma \models_n: /_F P \text{ c } Q, A)$   
*<proof>*

**lemma cvalid-to-cvalid:**  $(\forall n. \Gamma, \Theta \models_n: /_F P \text{ c } Q, A) \Longrightarrow \Gamma, \Theta \models_n: /_F P \text{ c } Q, A$   
*<proof>*

**lemma nvalidI:**

$$[\bigwedge s t. [\Gamma \vdash \langle c, \text{Normal } s \rangle =_n \Rightarrow t; s \in P; t \notin \text{Fault} \text{ ' } F] \Longrightarrow t \in \text{Normal} \text{ ' } Q \cup \text{Abrupt} \\ \text{ ' } A] \\ \Longrightarrow \Gamma \models_n: /_F P \text{ c } Q, A \\ \langle \text{proof} \rangle$$

**lemma validI:**

$$[\bigwedge s t. [\Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow t; s \in P; t \notin \text{Fault} \text{ ' } F] \Longrightarrow t \in \text{Normal} \text{ ' } Q \cup \text{Abrupt} \text{ ' } \\ A] \\ \Longrightarrow \Gamma \models_n: /_F P \text{ c } Q, A \\ \langle \text{proof} \rangle$$

**lemma cvalidI:**

$$[\bigwedge s t. [\forall (P, p, Q, A) \in \Theta. \Gamma \models_n: /_F P (\text{Call } p) Q, A; \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow t; s \in P; t \notin \text{Fault} \\ \text{ ' } F] \\ \Longrightarrow t \in \text{Normal} \text{ ' } Q \cup \text{Abrupt} \text{ ' } A] \\ \Longrightarrow \Gamma, \Theta \models_n: /_F P \text{ c } Q, A \\ \langle \text{proof} \rangle$$

**lemma** *cvalidD*:

$\llbracket \Gamma, \Theta \models_{/F} P \text{ c } Q, A; \forall (P, p, Q, A) \in \Theta. \Gamma \models_{/F} P \text{ (Call } p) \text{ } Q, A; \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow t; s \in P; t \notin \text{Fault ' } F \rrbracket$   
 $\Rightarrow t \in \text{Normal ' } Q \cup \text{Abrupt ' } A$   
 $\langle \text{proof} \rangle$

**lemma** *cvalidI*:

$\llbracket \Gamma \text{ s } t. \llbracket \forall (P, p, Q, A) \in \Theta. \Gamma \models_n:_{/F} P \text{ (Call } p) \text{ } Q, A; \Gamma \vdash \langle c, \text{Normal } s \rangle =_n \Rightarrow t; s \in P; t \notin \text{Fault ' } F \rrbracket$   
 $\Rightarrow t \in \text{Normal ' } Q \cup \text{Abrupt ' } A \rrbracket$   
 $\Rightarrow \Gamma, \Theta \models_n:_{/F} P \text{ c } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *cnvalidD*:

$\llbracket \Gamma, \Theta \models_n:_{/F} P \text{ c } Q, A; \forall (P, p, Q, A) \in \Theta. \Gamma \models_n:_{/F} P \text{ (Call } p) \text{ } Q, A; \Gamma \vdash \langle c, \text{Normal } s \rangle =_n \Rightarrow t; s \in P; t \notin \text{Fault ' } F \rrbracket$   
 $\Rightarrow t \in \text{Normal ' } Q \cup \text{Abrupt ' } A$   
 $\langle \text{proof} \rangle$

**lemma** *nvalid-augment-Faults*:

**assumes** *validn*:  $\Gamma \models_n:_{/F} P \text{ c } Q, A$   
**assumes**  $F': F \subseteq F'$   
**shows**  $\Gamma \models_n:_{/F'} P \text{ c } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *valid-augment-Faults*:

**assumes** *validn*:  $\Gamma \models_{/F} P \text{ c } Q, A$   
**assumes**  $F': F \subseteq F'$   
**shows**  $\Gamma \models_{/F'} P \text{ c } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *nvalid-to-nvalid-strip*:

**assumes** *validn*:  $\Gamma \models_n:_{/F} P \text{ c } Q, A$   
**assumes**  $F': F' \subseteq -F$   
**shows** *strip*  $F' \Gamma \models_n:_{/F} P \text{ c } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *valid-to-valid-strip*:

**assumes** *valid*:  $\Gamma \models_{/F} P \text{ c } Q, A$   
**assumes**  $F': F' \subseteq -F$   
**shows** *strip*  $F' \Gamma \models_{/F} P \text{ c } Q, A$   
 $\langle \text{proof} \rangle$

### 4.3 The Hoare Rules: $\Gamma, \Theta \vdash_F P \ c \ Q, A$

**lemma** *mono-WeakenContext*:  $A \subseteq B \implies$   
 $(\lambda(P, c, Q, A'). (\Gamma, \Theta, F, P, c, Q, A') \in A) \ x \longrightarrow$   
 $(\lambda(P, c, Q, A'). (\Gamma, \Theta, F, P, c, Q, A') \in B) \ x$   
*<proof>*

**inductive** *hoarep*:: $(('s, 'p, 'f) \text{ body}, ('s, 'p) \text{ quadruple set}, 'f \text{ set},$   
 $'s \text{ assn}, ('s, 'p, 'f) \text{ com}, 's \text{ assn}, 's \text{ assn}] \implies \text{bool}$   
 $(\langle (\exists -, -/\vdash -/-) (-) / -, -/ \rangle [60, 60, 60, 1000, 20, 1000, 1000] 60)$   
**for**  $\Gamma :: ('s, 'p, 'f) \text{ body}$

**where**

*Skip*:  $\Gamma, \Theta \vdash_F Q \ \text{Skip} \ Q, A$

| *Basic*:  $\Gamma, \Theta \vdash_F \{s. f \ s \in Q\} \ (\text{Basic } f) \ Q, A$

| *Spec*:  $\Gamma, \Theta \vdash_F \{s. (\forall t. (s, t) \in r \longrightarrow t \in Q) \wedge (\exists t. (s, t) \in r)\} \ (\text{Spec } r) \ Q, A$

| *Seq*:  $\llbracket \Gamma, \Theta \vdash_F P \ c_1 \ R, A; \Gamma, \Theta \vdash_F R \ c_2 \ Q, A \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash_F P \ (\text{Seq } c_1 \ c_2) \ Q, A$

| *Cond*:  $\llbracket \Gamma, \Theta \vdash_F (P \cap b) \ c_1 \ Q, A; \Gamma, \Theta \vdash_F (P \cap - \ b) \ c_2 \ Q, A \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash_F P \ (\text{Cond } b \ c_1 \ c_2) \ Q, A$

| *While*:  $\Gamma, \Theta \vdash_F (P \cap b) \ c \ P, A$   
 $\implies$   
 $\Gamma, \Theta \vdash_F P \ (\text{While } b \ c) \ (P \cap - \ b), A$

| *Guard*:  $\Gamma, \Theta \vdash_F (g \cap P) \ c \ Q, A$   
 $\implies$   
 $\Gamma, \Theta \vdash_F (g \cap P) \ (\text{Guard } f \ g \ c) \ Q, A$

| *Guarantee*:  $\llbracket f \in F; \Gamma, \Theta \vdash_F (g \cap P) \ c \ Q, A \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash_F P \ (\text{Guard } f \ g \ c) \ Q, A$

| *CallRec*:

$\llbracket (P, p, Q, A) \in \text{Specs};$   
 $\forall (P, p, Q, A) \in \text{Specs}. p \in \text{dom } \Gamma \wedge \Gamma, \Theta \cup \text{Specs} \vdash_F P \ (\text{the } (\Gamma \ p)) \ Q, A \rrbracket$   
 $\implies \Gamma, \Theta \vdash_F P \ (\text{Call } p) \ Q, A$

| *DynCom*:

$\forall s \in P. \Gamma, \Theta \vdash_F P \ (c \ s) \ Q, A$   
 $\implies$   
 $\Gamma, \Theta \vdash_F P \ (\text{DynCom } c) \ Q, A$

| *Throw*:  $\Gamma, \Theta \vdash_{/F} A \text{ Throw } Q, A$

| *Catch*:  $\llbracket \Gamma, \Theta \vdash_{/F} P \ c_1 \ Q, R; \Gamma, \Theta \vdash_{/F} R \ c_2 \ Q, A \rrbracket \implies \Gamma, \Theta \vdash_{/F} P \text{ Catch } c_1 \ c_2 \ Q, A$

| *Conseq*:  $\forall s \in P. \exists P' \ Q' \ A'. \Gamma, \Theta \vdash_{/F} P' \ c \ Q', A' \wedge s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A$   
 $\implies \Gamma, \Theta \vdash_{/F} P \ c \ Q, A$

| *Asm*:  $\llbracket (P, p, Q, A) \in \Theta \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash_{/F} P \ (\text{Call } p) \ Q, A$

| *ExFalso*:  $\llbracket \forall n. \Gamma, \Theta \models_n \vdash_{/F} P \ c \ Q, A; \neg \Gamma \models_{/F} P \ c \ Q, A \rrbracket \implies \Gamma, \Theta \vdash_{/F} P \ c \ Q, A$

— This is a hack rule that enables us to derive completeness for an arbitrary context  $\Theta$ , from completeness for an empty context.

Does not work, because of rule *ExFalso*, the context  $\Theta$  is to blame. A weaker version with empty context can be derived from soundness and completeness later on.

**lemma** *hoare-strip- $\Gamma$* :

**assumes** *deriv*:  $\Gamma, \Theta \vdash_{/F} P \ p \ Q, A$

**shows** *strip*  $(-F) \Gamma, \Theta \vdash_{/F} P \ p \ Q, A$

*<proof>*

**lemma** *hoare-augment-context*:

**assumes** *deriv*:  $\Gamma, \Theta \vdash_{/F} P \ p \ Q, A$

**shows**  $\bigwedge \Theta'. \Theta \subseteq \Theta' \implies \Gamma, \Theta \uparrow_{/F} P \ p \ Q, A$

*<proof>*

#### 4.4 Some Derived Rules

**lemma** *Conseq'*:  $\forall s. s \in P \longrightarrow$

$(\exists P' \ Q' \ A'.$

$(\forall Z. \Gamma, \Theta \vdash_{/F} (P' \ Z) \ c \ (Q' \ Z), (A' \ Z)) \wedge$

$(\exists Z. s \in P' \ Z \wedge (Q' \ Z \subseteq Q) \wedge (A' \ Z \subseteq A)))$

$\implies$

$\Gamma, \Theta \vdash_{/F} P \ c \ Q, A$

*<proof>*

**lemma** *conseq*:  $\llbracket \forall Z. \Gamma, \Theta \vdash_{/F} (P' \ Z) \ c \ (Q' \ Z), (A' \ Z);$

$\forall s. s \in P \longrightarrow (\exists Z. s \in P' \ Z \wedge (Q' \ Z \subseteq Q) \wedge (A' \ Z \subseteq A)) \rrbracket$

$\implies$

$\Gamma, \Theta \vdash_{/F} P \ c \ Q, A$

*<proof>*

**theorem** *conseqPrePost* [*trans*]:

$\Gamma, \Theta \vdash_{/F} P' c Q', A' \implies P \subseteq P' \implies Q' \subseteq Q \implies A' \subseteq A \implies \Gamma, \Theta \vdash_{/F} P c Q, A$   
 ⟨proof⟩

**lemma** *conseqPre* [trans]:  $\Gamma, \Theta \vdash_{/F} P' c Q, A \implies P \subseteq P' \implies \Gamma, \Theta \vdash_{/F} P c Q, A$   
 ⟨proof⟩

**lemma** *conseqPost* [trans]:  $\Gamma, \Theta \vdash_{/F} P c Q', A' \implies Q' \subseteq Q \implies A' \subseteq A$   
 $\implies \Gamma, \Theta \vdash_{/F} P c Q, A$   
 ⟨proof⟩

**lemma** *CallRec'*:

$\llbracket p \in \text{Procs}; \text{Procs} \subseteq \text{dom } \Gamma;$   
 $\forall p \in \text{Procs}.$   
 $\forall Z. \Gamma, \Theta \cup (\bigcup_{p \in \text{Procs}} \bigcup Z. \{((P \ p \ Z), p, Q \ p \ Z, A \ p \ Z)\})$   
 $\vdash_{/F} (P \ p \ Z) (\text{the } (\Gamma \ p)) (Q \ p \ Z), (A \ p \ Z)$   
 $\implies$   
 $\Gamma, \Theta \vdash_{/F} (P \ p \ Z) (\text{Call } p) (Q \ p \ Z), (A \ p \ Z)$   
 ⟨proof⟩

end

## 5 Properties of Partial Correctness Hoare Logic

**theory** *HoarePartialProps* imports *HoarePartialDef* begin

### 5.1 Soundness

**lemma** *hoare-cnvalid*:  
**assumes** *hoare*:  $\Gamma, \Theta \vdash_{/F} P c Q, A$   
**shows**  $\bigwedge n. \Gamma, \Theta \models n:_{/F} P c Q, A$   
 ⟨proof⟩

**theorem** *hoare-sound*:  $\Gamma, \Theta \vdash_{/F} P c Q, A \implies \Gamma, \Theta \models_{/F} P c Q, A$   
 ⟨proof⟩

### 5.2 Completeness

**lemma** *MGT-valid*:  
 $\Gamma \models_{/F} \{s. s=Z \wedge \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \neg(\{\text{Stuck}\} \cup \text{Fault } '(-F))\} c$   
 $\{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\}, \{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$   
 ⟨proof⟩

The consequence rule where the existential  $Z$  is instantiated to  $s$ . Usefull in proof of *MGT-lemma*.

**lemma** *ConseqMGT*:  
**assumes** *modif*:  $\forall Z. \Gamma, \Theta \vdash_{/F} (P' Z) c (Q' Z), (A' Z)$

**assumes impl:**  $\bigwedge s. s \in P \implies s \in P' \wedge (\forall t. t \in Q' \wedge s \longrightarrow t \in Q) \wedge$   
 $(\forall t. t \in A' \wedge s \longrightarrow t \in A)$

**shows**  $\Gamma, \Theta \vdash_{/F} P \text{ c } Q, A$

*<proof>*

**lemma Seq-NoFaultStuckD1:**

**assumes noabort:**  $\Gamma \vdash \langle \text{Seq } c1 \ c2, s \rangle \Rightarrow \notin (\{Stuck\} \cup \text{Fault } ' F)$

**shows**  $\Gamma \vdash \langle c1, s \rangle \Rightarrow \notin (\{Stuck\} \cup \text{Fault } ' F)$

*<proof>*

**lemma Seq-NoFaultStuckD2:**

**assumes noabort:**  $\Gamma \vdash \langle \text{Seq } c1 \ c2, s \rangle \Rightarrow \notin (\{Stuck\} \cup \text{Fault } ' F)$

**shows**  $\forall t. \Gamma \vdash \langle c1, s \rangle \Rightarrow t \longrightarrow t \notin (\{Stuck\} \cup \text{Fault } ' F) \longrightarrow$

$\Gamma \vdash \langle c2, t \rangle \Rightarrow \notin (\{Stuck\} \cup \text{Fault } ' F)$

*<proof>*

**lemma MGT-implies-complete:**

**assumes MGT:**  $\forall Z. \Gamma, \{\} \vdash_{/F} \{s. s=Z \wedge \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \notin (\{Stuck\} \cup \text{Fault } ' (-F))\} \text{ c}$

$\{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$

$\{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$

**assumes valid:**  $\Gamma \models_{/F} P \text{ c } Q, A$

**shows**  $\Gamma, \{\} \vdash_{/F} P \text{ c } Q, A$

*<proof>*

Equipped only with the classic consequence rule  $\llbracket ?\Gamma, ?\Theta \vdash_{/?F} ?P' \ ?c \ ?Q', ?A'; ?P \subseteq ?P'; ?Q' \subseteq ?Q; ?A' \subseteq ?A \rrbracket \implies ?\Gamma, ?\Theta \vdash_{/?F} ?P \ ?c \ ?Q, ?A$  we can only derive this syntactically more involved version of completeness. But semantically it is equivalent to the "real" one (see below)

**lemma MGT-implies-complete':**

**assumes MGT:**  $\forall Z. \Gamma, \{\} \vdash_{/F}$

$\{s. s=Z \wedge \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \notin (\{Stuck\} \cup \text{Fault } ' (-F))\} \text{ c}$

$\{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$

$\{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$

**assumes valid:**  $\Gamma \models_{/F} P \text{ c } Q, A$

**shows**  $\Gamma, \{\} \vdash_{/F} \{s. s=Z \wedge s \in P\} \text{ c } \{t. Z \in P \longrightarrow t \in Q\}, \{t. Z \in P \longrightarrow t \in A\}$

*<proof>*

Semantic equivalence of both kind of formulations

**lemma valid-involved-to-valid:**

**assumes valid:**

$\forall Z. \Gamma \models_{/F} \{s. s=Z \wedge s \in P\} \text{ c } \{t. Z \in P \longrightarrow t \in Q\}, \{t. Z \in P \longrightarrow t \in A\}$

**shows**  $\Gamma \models_{/F} P \text{ c } Q, A$

*<proof>*

The sophisticated consequence rule allow us to do this semantical transformation on the hoare-level, too. The magic is, that it allow us to choose the instance of  $Z$  under the assumption of an state  $s \in P$

**lemma**

**assumes** *deriv*:

$$\forall Z. \Gamma, \{\} \vdash_{/F} \{s. s=Z \wedge s \in P\} c \{t. Z \in P \longrightarrow t \in Q\}, \{t. Z \in P \longrightarrow t \in A\}$$

**shows**  $\Gamma, \{\} \vdash_{/F} P c Q, A$

*<proof>*

**lemma** *valid-to-valid-involved*:

$$\Gamma \models_{/F} P c Q, A \implies$$

$$\Gamma \models_{/F} \{s. s=Z \wedge s \in P\} c \{t. Z \in P \longrightarrow t \in Q\}, \{t. Z \in P \longrightarrow t \in A\}$$

*<proof>*

**lemma**

**assumes** *deriv*:  $\Gamma, \{\} \vdash_{/F} P c Q, A$

**shows**  $\Gamma, \{\} \vdash_{/F} \{s. s=Z \wedge s \in P\} c \{t. Z \in P \longrightarrow t \in Q\}, \{t. Z \in P \longrightarrow t \in A\}$

*<proof>*

**lemma** *conseq-extract-state-indep-prop*:

**assumes** *state-indep-prop*:  $\forall s \in P. R$

**assumes** *to-show*:  $R \implies \Gamma, \Theta \vdash_{/F} P c Q, A$

**shows**  $\Gamma, \Theta \vdash_{/F} P c Q, A$

*<proof>*

**lemma** *MGT-lemma*:

**assumes** *MGT-Calls*:

$$\forall p \in \text{dom } \Gamma. \forall Z. \Gamma, \Theta \vdash_{/F}$$

$$\{s. s=Z \wedge \Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle \Rightarrow \notin (\{\text{Stuck}\} \cup \text{Fault } '(-F))\}$$

$$(\text{Call } p)$$

$$\{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$$

$$\{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$$

**shows**  $\bigwedge Z. \Gamma, \Theta \vdash_{/F} \{s. s=Z \wedge \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \notin (\{\text{Stuck}\} \cup \text{Fault } '(-F))\} c$

$$\{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\}, \{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$$

*<proof>*

**lemma** *MGT-Calls*:

$$\forall p \in \text{dom } \Gamma. \forall Z.$$

$$\Gamma, \{\} \vdash_{/F} \{s. s=Z \wedge \Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle \Rightarrow \notin (\{\text{Stuck}\} \cup \text{Fault } '(-F))\}$$

$$(\text{Call } p)$$

$$\{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$$

$$\{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$$

*<proof>*

**theorem** *hoare-complete*:  $\Gamma \models_{/F} P c Q, A \implies \Gamma, \{\} \vdash_{/F} P c Q, A$

$\langle proof \rangle$

**lemma** *hoare-complete'*:

**assumes** *cvalid*:  $\forall n. \Gamma, \Theta \models n: /_F P \ c \ Q, A$

**shows**  $\Gamma, \Theta \vdash /_F P \ c \ Q, A$

$\langle proof \rangle$

**lemma** *hoare-strip- $\Gamma$* :

**assumes** *deriv*:  $\Gamma, \{\} \vdash /_F P \ p \ Q, A$

**assumes** *F'*:  $F' \subseteq -F$

**shows** *strip*  $F' \ \Gamma, \{\} \vdash /_F P \ p \ Q, A$

$\langle proof \rangle$

## 5.3 And Now: Some Useful Rules

### 5.3.1 Consequence

**lemma** *LiberalConseq-sound*:

**fixes** *F*::'f set

**assumes** *cons*:  $\forall s \in P. \forall (t::('s, 'f) \ xstate). \exists P' \ Q' \ A'. (\forall n. \Gamma, \Theta \models n: /_F P' \ c \ Q', A')$

$\wedge$

$((s \in P' \longrightarrow t \in \text{Normal } ' Q' \cup \text{Abrupt } ' A') \longrightarrow t \in \text{Normal } ' Q \cup \text{Abrupt } ' A)$

**shows**  $\Gamma, \Theta \models n: /_F P \ c \ Q, A$

$\langle proof \rangle$

**lemma** *LiberalConseq*:

**fixes** *F*::'f set

**assumes** *cons*:  $\forall s \in P. \forall (t::('s, 'f) \ xstate). \exists P' \ Q' \ A'. \Gamma, \Theta \vdash /_F P' \ c \ Q', A' \wedge$

$((s \in P' \longrightarrow t \in \text{Normal } ' Q' \cup \text{Abrupt } ' A') \longrightarrow t \in \text{Normal } ' Q \cup \text{Abrupt } ' A)$

**shows**  $\Gamma, \Theta \vdash /_F P \ c \ Q, A$

$\langle proof \rangle$

**lemma**  $\forall s \in P. \exists P' \ Q' \ A'. \Gamma, \Theta \vdash /_F P' \ c \ Q', A' \wedge s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A$

$\implies \Gamma, \Theta \vdash /_F P \ c \ Q, A$

$\langle proof \rangle$

**lemma**

**fixes** *F*::'f set

**assumes** *cons*:  $\forall s \in P. \exists P' \ Q' \ A'. \Gamma, \Theta \vdash /_F P' \ c \ Q', A' \wedge$

$(\forall (t::('s, 'f) \ xstate). (s \in P' \longrightarrow t \in \text{Normal } ' Q' \cup \text{Abrupt } ' A') \longrightarrow t \in \text{Normal } ' Q \cup \text{Abrupt } ' A)$

**shows**  $\Gamma, \Theta \vdash /_F P \ c \ Q, A$

$\langle proof \rangle$

**lemma** *LiberalConseq'*:

**fixes**  $F:: 'f \text{ set}$   
**assumes**  $\text{cons}: \forall s \in P. \exists P' Q' A'. \Gamma, \Theta \vdash /_F P' c Q', A' \wedge$   
 $(\forall (t::('s, 'f) \text{ xstate}). (s \in P' \longrightarrow t \in \text{Normal} \text{ ' } Q' \cup \text{Abrupt} \text{ ' } A') \longrightarrow t \in \text{Normal} \text{ ' } Q \cup \text{Abrupt} \text{ ' } A)$   
**shows**  $\Gamma, \Theta \vdash /_F P c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *LiberalConseq''*:  
**fixes**  $F:: 'f \text{ set}$   
**assumes**  $\text{spec}: \forall Z. \Gamma, \Theta \vdash /_F (P' Z) c (Q' Z), (A' Z)$   
**assumes**  $\text{cons}: \forall s (t::('s, 'f) \text{ xstate}).$   
 $(\forall Z. s \in P' Z \longrightarrow t \in \text{Normal} \text{ ' } Q' Z \cup \text{Abrupt} \text{ ' } A' Z)$   
 $\longrightarrow (s \in P \longrightarrow t \in \text{Normal} \text{ ' } Q \cup \text{Abrupt} \text{ ' } A)$   
**shows**  $\Gamma, \Theta \vdash /_F P c Q, A$   
 $\langle \text{proof} \rangle$

**primrec**  $\text{procs}:: ('s, 'p, 'f) \text{ com} \Rightarrow 'p \text{ set}$   
**where**  
 $\text{procs } \text{Skip} = \{\}$  |  
 $\text{procs } (\text{Basic } f) = \{\}$  |  
 $\text{procs } (\text{Seq } c_1 c_2) = (\text{procs } c_1 \cup \text{procs } c_2)$  |  
 $\text{procs } (\text{Cond } b c_1 c_2) = (\text{procs } c_1 \cup \text{procs } c_2)$  |  
 $\text{procs } (\text{While } b c) = \text{procs } c$  |  
 $\text{procs } (\text{Call } p) = \{p\}$  |  
 $\text{procs } (\text{DynCom } c) = (\bigcup s. \text{procs } (c s))$  |  
 $\text{procs } (\text{Guard } f g c) = \text{procs } c$  |  
 $\text{procs } \text{Throw} = \{\}$  |  
 $\text{procs } (\text{Catch } c_1 c_2) = (\text{procs } c_1 \cup \text{procs } c_2)$

**primrec**  $\text{noSpec}:: ('s, 'p, 'f) \text{ com} \Rightarrow \text{bool}$   
**where**  
 $\text{noSpec } \text{Skip} = \text{True}$  |  
 $\text{noSpec } (\text{Basic } f) = \text{True}$  |  
 $\text{noSpec } (\text{Spec } r) = \text{False}$  |  
 $\text{noSpec } (\text{Seq } c_1 c_2) = (\text{noSpec } c_1 \wedge \text{noSpec } c_2)$  |  
 $\text{noSpec } (\text{Cond } b c_1 c_2) = (\text{noSpec } c_1 \wedge \text{noSpec } c_2)$  |  
 $\text{noSpec } (\text{While } b c) = \text{noSpec } c$  |  
 $\text{noSpec } (\text{Call } p) = \text{True}$  |  
 $\text{noSpec } (\text{DynCom } c) = (\forall s. \text{noSpec } (c s))$  |  
 $\text{noSpec } (\text{Guard } f g c) = \text{noSpec } c$  |  
 $\text{noSpec } \text{Throw} = \text{True}$  |  
 $\text{noSpec } (\text{Catch } c_1 c_2) = (\text{noSpec } c_1 \wedge \text{noSpec } c_2)$

**lemma** *exec-noSpec-no-Stuck*:  
**assumes**  $\text{exec}: \Gamma \vdash \langle c, s \rangle \Rightarrow t$   
**assumes**  $\text{noSpec-c}: \text{noSpec } c$   
**assumes**  $\text{noSpec-}\Gamma: \forall p \in \text{dom } \Gamma. \text{noSpec } (\text{the } (\Gamma p))$   
**assumes**  $\text{procs-subset}: \text{procs } c \subseteq \text{dom } \Gamma$   
**assumes**  $\text{procs-subset-}\Gamma: \forall p \in \text{dom } \Gamma. \text{procs } (\text{the } (\Gamma p)) \subseteq \text{dom } \Gamma$

**assumes**  $s\text{-no-Stuck}$ :  $s \neq \text{Stuck}$   
**shows**  $t \neq \text{Stuck}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{execn-noSpec-no-Stuck}$ :  
**assumes**  $\text{exec}$ :  $\Gamma \vdash \langle c, s \rangle =_n \Rightarrow t$   
**assumes**  $\text{noSpec-c}$ :  $\text{noSpec } c$   
**assumes**  $\text{noSpec-}\Gamma$ :  $\forall p \in \text{dom } \Gamma. \text{noSpec } (\text{the } (\Gamma \ p))$   
**assumes**  $\text{procs-subset}$ :  $\text{procs } c \subseteq \text{dom } \Gamma$   
**assumes**  $\text{procs-subset-}\Gamma$ :  $\forall p \in \text{dom } \Gamma. \text{procs } (\text{the } (\Gamma \ p)) \subseteq \text{dom } \Gamma$   
**assumes**  $s\text{-no-Stuck}$ :  $s \neq \text{Stuck}$   
**shows**  $t \neq \text{Stuck}$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{LiberalConseq-noguards-nothrows-sound}$ :  
**assumes**  $\text{spec}$ :  $\forall Z. \forall n. \Gamma, \Theta \models_n /_F (P' \ Z) \ c \ (Q' \ Z), (A' \ Z)$   
**assumes**  $\text{cons}$ :  $\forall s \ t. (\forall Z. s \in P' \ Z \longrightarrow t \in Q' \ Z)$   
 $\longrightarrow (s \in P \longrightarrow t \in Q)$   
**assumes**  $\text{noguards-c}$ :  $\text{noguards } c$   
**assumes**  $\text{noguards-}\Gamma$ :  $\forall p \in \text{dom } \Gamma. \text{noguards } (\text{the } (\Gamma \ p))$   
**assumes**  $\text{nothrows-c}$ :  $\text{nothrows } c$   
**assumes**  $\text{nothrows-}\Gamma$ :  $\forall p \in \text{dom } \Gamma. \text{nothrows } (\text{the } (\Gamma \ p))$   
**assumes**  $\text{noSpec-c}$ :  $\text{noSpec } c$   
**assumes**  $\text{noSpec-}\Gamma$ :  $\forall p \in \text{dom } \Gamma. \text{noSpec } (\text{the } (\Gamma \ p))$   
**assumes**  $\text{procs-subset}$ :  $\text{procs } c \subseteq \text{dom } \Gamma$   
**assumes**  $\text{procs-subset-}\Gamma$ :  $\forall p \in \text{dom } \Gamma. \text{procs } (\text{the } (\Gamma \ p)) \subseteq \text{dom } \Gamma$   
**shows**  $\Gamma, \Theta \models_n /_F P \ c \ Q, A$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{LiberalConseq-noguards-nothrows}$ :  
**assumes**  $\text{spec}$ :  $\forall Z. \Gamma, \Theta \vdash /_F (P' \ Z) \ c \ (Q' \ Z), (A' \ Z)$   
**assumes**  $\text{cons}$ :  $\forall s \ t. (\forall Z. s \in P' \ Z \longrightarrow t \in Q' \ Z)$   
 $\longrightarrow (s \in P \longrightarrow t \in Q)$   
**assumes**  $\text{noguards-c}$ :  $\text{noguards } c$   
**assumes**  $\text{noguards-}\Gamma$ :  $\forall p \in \text{dom } \Gamma. \text{noguards } (\text{the } (\Gamma \ p))$   
**assumes**  $\text{nothrows-c}$ :  $\text{nothrows } c$   
**assumes**  $\text{nothrows-}\Gamma$ :  $\forall p \in \text{dom } \Gamma. \text{nothrows } (\text{the } (\Gamma \ p))$   
**assumes**  $\text{noSpec-c}$ :  $\text{noSpec } c$   
**assumes**  $\text{noSpec-}\Gamma$ :  $\forall p \in \text{dom } \Gamma. \text{noSpec } (\text{the } (\Gamma \ p))$   
**assumes**  $\text{procs-subset}$ :  $\text{procs } c \subseteq \text{dom } \Gamma$   
**assumes**  $\text{procs-subset-}\Gamma$ :  $\forall p \in \text{dom } \Gamma. \text{procs } (\text{the } (\Gamma \ p)) \subseteq \text{dom } \Gamma$   
**shows**  $\Gamma, \Theta \vdash /_F P \ c \ Q, A$   
 $\langle \text{proof} \rangle$

**lemma**  
**assumes**  $\text{spec}$ :  $\forall Z. \Gamma, \Theta \vdash /_F \{s. s = \text{fst } Z \wedge P \ s \ (\text{snd } Z)\} \ c \ \{t. Q \ (\text{fst } Z) \ (\text{snd } Z)$   
 $t\}, \{\}$

**assumes** *noguards-c*: *noguards c*  
**assumes** *noguards- $\Gamma$* :  $\forall p \in \text{dom } \Gamma. \text{noguards } (\text{the } (\Gamma p))$   
**assumes** *nothrows-c*: *nothrows c*  
**assumes** *nothrows- $\Gamma$* :  $\forall p \in \text{dom } \Gamma. \text{nothrows } (\text{the } (\Gamma p))$   
**assumes** *noSpec-c*: *noSpec c*  
**assumes** *noSpec- $\Gamma$* :  $\forall p \in \text{dom } \Gamma. \text{noSpec } (\text{the } (\Gamma p))$   
**assumes** *procs-subset*: *procs c*  $\subseteq$  *dom*  $\Gamma$   
**assumes** *procs-subset- $\Gamma$* :  $\forall p \in \text{dom } \Gamma. \text{procs } (\text{the } (\Gamma p)) \subseteq \text{dom } \Gamma$   
**shows**  $\forall \sigma. \Gamma, \Theta \vdash_{/F} \{s. s = \sigma\} c \{t. \forall l. P \sigma l \longrightarrow Q \sigma l t\}, \{\}$   
 $\langle \text{proof} \rangle$

### 5.3.2 Modify Return

**lemma** *Proc-exnModifyReturn-sound*:

**assumes** *valid-call*:  $\forall n. \Gamma, \Theta \models_{/F} P \text{ call-exn } \text{init } p \text{ return}' \text{ result-exn } c \ Q, A$   
**assumes** *valid-modif*:  
 $\forall \sigma. \forall n. \Gamma, \Theta \models_{/UNIV} \{\sigma\} \text{ Call } p \ (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**assumes** *ret-modif*:  
 $\forall s \ t. t \in \text{Modif } (\text{init } s)$   
 $\longrightarrow \text{return}' s \ t = \text{return } s \ t$   
**assumes** *ret-modifAbr*:  $\forall s \ t. t \in \text{ModifAbr } (\text{init } s)$   
 $\longrightarrow \text{result-exn } (\text{return}' s \ t) \ t = \text{result-exn } (\text{return } s \ t) \ t$   
**shows**  $\Gamma, \Theta \models_{/F} P \ (\text{call-exn } \text{init } p \ \text{return} \ \text{result-exn } c) \ Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *ProcModifyReturn-sound*:

**assumes** *valid-call*:  $\forall n. \Gamma, \Theta \models_{/F} P \text{ call } \text{init } p \ \text{return}' \ c \ Q, A$   
**assumes** *valid-modif*:  
 $\forall \sigma. \forall n. \Gamma, \Theta \models_{/UNIV} \{\sigma\} \text{ Call } p \ (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**assumes** *ret-modif*:  
 $\forall s \ t. t \in \text{Modif } (\text{init } s)$   
 $\longrightarrow \text{return}' s \ t = \text{return } s \ t$   
**assumes** *ret-modifAbr*:  $\forall s \ t. t \in \text{ModifAbr } (\text{init } s)$   
 $\longrightarrow \text{return}' s \ t = \text{return } s \ t$   
**shows**  $\Gamma, \Theta \models_{/F} P \ (\text{call } \text{init } p \ \text{return} \ c) \ Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *Proc-exnModifyReturn*:

**assumes** *spec*:  $\Gamma, \Theta \vdash_{/F} P \ (\text{call-exn } \text{init } p \ \text{return}' \ \text{result-exn } c) \ Q, A$   
**assumes** *result-conform*:  
 $\forall s \ t. t \in \text{Modif } (\text{init } s) \longrightarrow (\text{return}' s \ t) = (\text{return } s \ t)$   
**assumes** *return-conform*:  
 $\forall s \ t. t \in \text{ModifAbr } (\text{init } s)$   
 $\longrightarrow (\text{result-exn } (\text{return}' s \ t) \ t) = (\text{result-exn } (\text{return } s \ t) \ t)$   
**assumes** *modifies-spec*:  
 $\forall \sigma. \Gamma, \Theta \vdash_{/UNIV} \{\sigma\} \text{ Call } p \ (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\Gamma, \Theta \vdash_{/F} P \ (\text{call-exn } \text{init } p \ \text{return} \ \text{result-exn } c) \ Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *ProcModifyReturn*:  
**assumes** *spec*:  $\Gamma, \Theta \vdash_{/F} P \text{ (call init p return' c) } Q, A$   
**assumes** *result-conform*:  
 $\forall s t. t \in \text{Modif (init s)} \longrightarrow (\text{return' s t}) = (\text{return s t})$   
**assumes** *return-conform*:  
 $\forall s t. t \in \text{ModifAbr (init s)}$   
 $\longrightarrow (\text{return' s t}) = (\text{return s t})$   
**assumes** *modifies-spec*:  
 $\forall \sigma. \Gamma, \Theta \vdash_{/UNIV} \{\sigma\} \text{ Call p (Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\Gamma, \Theta \vdash_{/F} P \text{ (call init p return c) } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *Proc-exnModifyReturnSameFaults-sound*:  
**assumes** *valid-call*:  $\forall n. \Gamma, \Theta \models_{/F} P \text{ call-exn init p return' result-exn c } Q, A$   
**assumes** *valid-modif*:  
 $\forall \sigma. \forall n. \Gamma, \Theta \models_{/F} \{\sigma\} \text{ Call p (Modif } \sigma), (\text{ModifAbr } \sigma)$   
**assumes** *ret-modif*:  
 $\forall s t. t \in \text{Modif (init s)}$   
 $\longrightarrow \text{return' s t} = \text{return s t}$   
**assumes** *ret-modifAbr*:  $\forall s t. t \in \text{ModifAbr (init s)}$   
 $\longrightarrow \text{result-exn (return' s t) t} = \text{result-exn (return s t) t}$   
**shows**  $\Gamma, \Theta \models_{/F} P \text{ (call-exn init p return result-exn c) } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *ProcModifyReturnSameFaults-sound*:  
**assumes** *valid-call*:  $\forall n. \Gamma, \Theta \models_{/F} P \text{ call init p return' c } Q, A$   
**assumes** *valid-modif*:  
 $\forall \sigma. \forall n. \Gamma, \Theta \models_{/F} \{\sigma\} \text{ Call p (Modif } \sigma), (\text{ModifAbr } \sigma)$   
**assumes** *ret-modif*:  
 $\forall s t. t \in \text{Modif (init s)}$   
 $\longrightarrow \text{return' s t} = \text{return s t}$   
**assumes** *ret-modifAbr*:  $\forall s t. t \in \text{ModifAbr (init s)}$   
 $\longrightarrow \text{return' s t} = \text{return s t}$   
**shows**  $\Gamma, \Theta \models_{/F} P \text{ (call init p return c) } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *Proc-exnModifyReturnSameFaults*:  
**assumes** *spec*:  $\Gamma, \Theta \vdash_{/F} P \text{ (call-exn init p return' result-exn c) } Q, A$   
**assumes** *result-conform*:  
 $\forall s t. t \in \text{Modif (init s)} \longrightarrow (\text{return' s t}) = (\text{return s t})$   
**assumes** *return-conform*:  
 $\forall s t. t \in \text{ModifAbr (init s)} \longrightarrow (\text{result-exn (return' s t) t}) = (\text{result-exn (return s t) t})$   
**assumes** *modifies-spec*:  
 $\forall \sigma. \Gamma, \Theta \vdash_{/F} \{\sigma\} \text{ Call p (Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\Gamma, \Theta \vdash_{/F} P \text{ (call-exn init p return result-exn c) } Q, A$

*<proof>*

**lemma** *ProcModifyReturnSameFaults:*

**assumes** *spec:*  $\Gamma, \Theta \vdash_{/F} P \text{ (call init p return' c) } Q, A$

**assumes** *result-conform:*

$\forall s t. t \in \text{Modif (init s)} \longrightarrow (\text{return' s t}) = (\text{return s t})$

**assumes** *return-conform:*

$\forall s t. t \in \text{ModifAbr (init s)} \longrightarrow (\text{return' s t}) = (\text{return s t})$

**assumes** *modifies-spec:*

$\forall \sigma. \Gamma, \Theta \vdash_{/F} \{\sigma\} \text{ Call p (Modif } \sigma), (\text{ModifAbr } \sigma)$

**shows**  $\Gamma, \Theta \vdash_{/F} P \text{ (call init p return c) } Q, A$

*<proof>*

### 5.3.3 DynCall

**lemma** *dynProc-exnModifyReturn-sound:*

**assumes** *valid-call:*  $\bigwedge n. \Gamma, \Theta \models_{/F} P \text{ dynCall-exn f g init p return' result-exn c } Q, A$

**assumes** *valid-modif:*

$\forall s \in P. \forall \sigma. \forall n.$

$\Gamma, \Theta \models_{/UNIV} \{\sigma\} \text{ Call (p s) (Modif } \sigma), (\text{ModifAbr } \sigma)$

**assumes** *ret-modif:*

$\forall s t. t \in \text{Modif (init s)}$

$\longrightarrow \text{return' s t} = \text{return s t}$

**assumes** *ret-modifAbr:*  $\forall s t. t \in \text{ModifAbr (init s)}$

$\longrightarrow \text{result-exn (return' s t) t} = \text{result-exn (return s t) t}$

**shows**  $\Gamma, \Theta \models_{/F} P \text{ (dynCall-exn f g init p return result-exn c) } Q, A$

*<proof>*

**lemma** *dynProcModifyReturn-sound:*

**assumes** *valid-call:*  $\bigwedge n. \Gamma, \Theta \models_{/F} P \text{ dynCall init p return' c } Q, A$

**assumes** *valid-modif:*

$\forall s \in P. \forall \sigma. \forall n.$

$\Gamma, \Theta \models_{/UNIV} \{\sigma\} \text{ Call (p s) (Modif } \sigma), (\text{ModifAbr } \sigma)$

**assumes** *ret-modif:*

$\forall s t. t \in \text{Modif (init s)}$

$\longrightarrow \text{return' s t} = \text{return s t}$

**assumes** *ret-modifAbr:*  $\forall s t. t \in \text{ModifAbr (init s)}$

$\longrightarrow \text{return' s t} = \text{return s t}$

**shows**  $\Gamma, \Theta \models_{/F} P \text{ (dynCall init p return c) } Q, A$

*<proof>*

**lemma** *dynProc-exnModifyReturn:*

**assumes** *dyn-call:*  $\Gamma, \Theta \vdash_{/F} P \text{ dynCall-exn f g init p return' result-exn c } Q, A$

**assumes** *ret-modif:*

$\forall s t. t \in \text{Modif (init s)}$

$\longrightarrow \text{return}' s t = \text{return} s t$   
**assumes** *ret-modifAbr*:  $\forall s t. t \in \text{ModifAbr} (\text{init } s)$   
 $\longrightarrow \text{result-exn} (\text{return}' s t) t = \text{result-exn} (\text{return} s t) t$   
**assumes** *modif*:  
 $\forall s \in P. \forall \sigma.$   
 $\Gamma, \Theta \vdash_{/UNIV} \{\sigma\} \text{Call} (p s) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\Gamma, \Theta \vdash_{/F} P (\text{dynCall-exn } f g \text{init } p \text{return result-exn } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *dynProcModifyReturn*:  
**assumes** *dyn-call*:  $\Gamma, \Theta \vdash_{/F} P \text{dynCall init } p \text{return}' c Q, A$   
**assumes** *ret-modif*:  
 $\forall s t. t \in \text{Modif} (\text{init } s)$   
 $\longrightarrow \text{return}' s t = \text{return} s t$   
**assumes** *ret-modifAbr*:  $\forall s t. t \in \text{ModifAbr} (\text{init } s)$   
 $\longrightarrow \text{return}' s t = \text{return} s t$   
**assumes** *modif*:  
 $\forall s \in P. \forall \sigma.$   
 $\Gamma, \Theta \vdash_{/UNIV} \{\sigma\} \text{Call} (p s) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\Gamma, \Theta \vdash_{/F} P (\text{dynCall init } p \text{return } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *dynProc-exnModifyReturnSameFaults-sound*:  
**assumes** *valid-call*:  $\bigwedge n. \Gamma, \Theta \models n:_{/F} P \text{dynCall-exn } f g \text{init } p \text{return}' \text{result-exn } c$   
 $Q, A$   
**assumes** *valid-modif*:  
 $\forall s \in P. \forall \sigma. \forall n.$   
 $\Gamma, \Theta \models n:_{/F} \{\sigma\} \text{Call} (p s) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**assumes** *ret-modif*:  
 $\forall s t. t \in \text{Modif} (\text{init } s) \longrightarrow \text{return}' s t = \text{return} s t$   
**assumes** *ret-modifAbr*:  $\forall s t. t \in \text{ModifAbr} (\text{init } s) \longrightarrow \text{result-exn} (\text{return}' s t) t$   
 $= \text{result-exn} (\text{return} s t) t$   
**shows**  $\Gamma, \Theta \models n:_{/F} P (\text{dynCall-exn } f g \text{init } p \text{return result-exn } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *dynProcModifyReturnSameFaults-sound*:  
**assumes** *valid-call*:  $\bigwedge n. \Gamma, \Theta \models n:_{/F} P \text{dynCall init } p \text{return}' c Q, A$   
**assumes** *valid-modif*:  
 $\forall s \in P. \forall \sigma. \forall n.$   
 $\Gamma, \Theta \models n:_{/F} \{\sigma\} \text{Call} (p s) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**assumes** *ret-modif*:  
 $\forall s t. t \in \text{Modif} (\text{init } s) \longrightarrow \text{return}' s t = \text{return} s t$   
**assumes** *ret-modifAbr*:  $\forall s t. t \in \text{ModifAbr} (\text{init } s) \longrightarrow \text{return}' s t = \text{return} s t$   
**shows**  $\Gamma, \Theta \models n:_{/F} P (\text{dynCall init } p \text{return } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *dynProc-exnModifyReturnSameFaults*:

**assumes** *dyn-call*:  $\Gamma, \Theta \vdash /_F P \text{ dynCall-exn } f \ g \ \text{init } p \ \text{return}' \ \text{result-exn } c \ Q, A$   
**assumes** *ret-modif*:  
 $\forall s \ t. t \in \text{Modif } (\text{init } s)$   
 $\longrightarrow \text{return}' \ s \ t = \text{return } s \ t$   
**assumes** *ret-modifAbr*:  $\forall s \ t. t \in \text{ModifAbr } (\text{init } s)$   
 $\longrightarrow \text{result-exn } (\text{return}' \ s \ t) \ t = \text{result-exn } (\text{return } s \ t) \ t$   
**assumes** *modif*:  
 $\forall s \in P. \forall \sigma. \Gamma, \Theta \vdash /_F \{\sigma\} \ \text{Call } (p \ s) \ (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\Gamma, \Theta \vdash /_F P \ (\text{dynCall-exn } f \ g \ \text{init } p \ \text{return} \ \text{result-exn } c) \ Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *dynProcModifyReturnSameFaults*:  
**assumes** *dyn-call*:  $\Gamma, \Theta \vdash /_F P \ \text{dynCall } \text{init } p \ \text{return}' \ c \ Q, A$   
**assumes** *ret-modif*:  
 $\forall s \ t. t \in \text{Modif } (\text{init } s)$   
 $\longrightarrow \text{return}' \ s \ t = \text{return } s \ t$   
**assumes** *ret-modifAbr*:  $\forall s \ t. t \in \text{ModifAbr } (\text{init } s)$   
 $\longrightarrow \text{return}' \ s \ t = \text{return } s \ t$   
**assumes** *modif*:  
 $\forall s \in P. \forall \sigma. \Gamma, \Theta \vdash /_F \{\sigma\} \ \text{Call } (p \ s) \ (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\Gamma, \Theta \vdash /_F P \ (\text{dynCall } \text{init } p \ \text{return } c) \ Q, A$   
 $\langle \text{proof} \rangle$

### 5.3.4 Conjunction of Postcondition

**lemma** *PostConjI-sound*:  
**assumes** *valid-Q*:  $\forall n. \Gamma, \Theta \models n: /_F P \ c \ Q, A$   
**assumes** *valid-R*:  $\forall n. \Gamma, \Theta \models n: /_F P \ c \ R, B$   
**shows**  $\Gamma, \Theta \models n: /_F P \ c \ (Q \cap R), (A \cap B)$   
 $\langle \text{proof} \rangle$

**lemma** *PostConjI*:  
**assumes** *deriv-Q*:  $\Gamma, \Theta \vdash /_F P \ c \ Q, A$   
**assumes** *deriv-R*:  $\Gamma, \Theta \vdash /_F P \ c \ R, B$   
**shows**  $\Gamma, \Theta \vdash /_F P \ c \ (Q \cap R), (A \cap B)$   
 $\langle \text{proof} \rangle$

**lemma** *Merge-PostConj-sound*:  
**assumes** *validF*:  $\forall n. \Gamma, \Theta \models n: /_F P \ c \ Q, A$   
**assumes** *validG*:  $\forall n. \Gamma, \Theta \models n: /_G P' \ c \ R, X$   
**assumes** *F-G*:  $F \subseteq G$   
**assumes** *P-P'*:  $P \subseteq P'$   
**shows**  $\Gamma, \Theta \models n: /_F P \ c \ (Q \cap R), (A \cap X)$   
 $\langle \text{proof} \rangle$

**lemma** *Merge-PostConj*:  
**assumes** *validF*:  $\Gamma, \Theta \vdash /_F P \ c \ Q, A$

**assumes** *validG*:  $\Gamma, \Theta \vdash /_G P' c R, X$   
**assumes** *F-G*:  $F \subseteq G$   
**assumes** *P-P'*:  $P \subseteq P'$   
**shows**  $\Gamma, \Theta \vdash /_F P c (Q \cap R), (A \cap X)$   
*<proof>*

### 5.3.5 Weaken Context

**lemma** *WeakenContext-sound*:  
**assumes** *valid-c*:  $\forall n. \Gamma, \Theta' \models n: /_F P c Q, A$   
**assumes** *valid-ctxt*:  $\forall (P, p, Q, A) \in \Theta'. \Gamma, \Theta \models n: /_F P (Call\ p)\ Q, A$   
**shows**  $\Gamma, \Theta \models n: /_F P c Q, A$   
*<proof>*

**lemma** *WeakenContext*:  
**assumes** *deriv-c*:  $\Gamma, \Theta \vdash /_F P c Q, A$   
**assumes** *deriv-ctxt*:  $\forall (P, p, Q, A) \in \Theta'. \Gamma, \Theta \vdash /_F P (Call\ p)\ Q, A$   
**shows**  $\Gamma, \Theta \vdash /_F P c Q, A$   
*<proof>*

### 5.3.6 Guards and Guarantees

**lemma** *SplitGuards-sound*:  
**assumes** *valid-c1*:  $\forall n. \Gamma, \Theta \models n: /_F P c_1 Q, A$   
**assumes** *valid-c2*:  $\forall n. \Gamma, \Theta \models n: /_F P c_2 UNIV, UNIV$   
**assumes** *c*:  $(c_1 \cap_g c_2) = Some\ c$   
**shows**  $\Gamma, \Theta \models n: /_F P c Q, A$   
*<proof>*

**lemma** *SplitGuards*:  
**assumes** *c*:  $(c_1 \cap_g c_2) = Some\ c$   
**assumes** *deriv-c1*:  $\Gamma, \Theta \vdash /_F P c_1 Q, A$   
**assumes** *deriv-c2*:  $\Gamma, \Theta \vdash /_F P c_2 UNIV, UNIV$   
**shows**  $\Gamma, \Theta \vdash /_F P c Q, A$   
*<proof>*

**lemma** *CombineStrip-sound*:  
**assumes** *valid*:  $\forall n. \Gamma, \Theta \models n: /_F P c Q, A$   
**assumes** *valid-strip*:  $\forall n. \Gamma, \Theta \models n: /_{\{\}} P (strip-guards\ (-F)\ c)\ UNIV, UNIV$   
**shows**  $\Gamma, \Theta \models n: /_{\{\}} P c Q, A$   
*<proof>*

**lemma** *CombineStrip*:  
**assumes** *deriv*:  $\Gamma, \Theta \vdash /_F P c Q, A$   
**assumes** *deriv-strip*:  $\Gamma, \Theta \vdash /_{\{\}} P (strip-guards\ (-F)\ c)\ UNIV, UNIV$   
**shows**  $\Gamma, \Theta \vdash /_{\{\}} P c Q, A$

$\langle proof \rangle$

**lemma** *GuardsFlip-sound*:

**assumes** *valid*:  $\forall n. \Gamma, \Theta \models n: /_F P c Q, A$

**assumes** *validFlip*:  $\forall n. \Gamma, \Theta \models n: /_{-F} P c UNIV, UNIV$

**shows**  $\Gamma, \Theta \models n: /\{\} P c Q, A$

$\langle proof \rangle$

**lemma** *GuardsFlip*:

**assumes** *deriv*:  $\Gamma, \Theta \vdash /_F P c Q, A$

**assumes** *derivFlip*:  $\Gamma, \Theta \vdash /_{-F} P c UNIV, UNIV$

**shows**  $\Gamma, \Theta \vdash /\{\} P c Q, A$

$\langle proof \rangle$

**lemma** *MarkGuardsI-sound*:

**assumes** *valid*:  $\forall n. \Gamma, \Theta \models n: /\{\} P c Q, A$

**shows**  $\Gamma, \Theta \models n: /\{\} P \text{ mark-guards } f c Q, A$

$\langle proof \rangle$

**lemma** *MarkGuardsI*:

**assumes** *deriv*:  $\Gamma, \Theta \vdash /\{\} P c Q, A$

**shows**  $\Gamma, \Theta \vdash /\{\} P \text{ mark-guards } f c Q, A$

$\langle proof \rangle$

**lemma** *MarkGuardsD-sound*:

**assumes** *valid*:  $\forall n. \Gamma, \Theta \models n: /\{\} P \text{ mark-guards } f c Q, A$

**shows**  $\Gamma, \Theta \models n: /\{\} P c Q, A$

$\langle proof \rangle$

**lemma** *MarkGuardsD*:

**assumes** *deriv*:  $\Gamma, \Theta \vdash /\{\} P \text{ mark-guards } f c Q, A$

**shows**  $\Gamma, \Theta \vdash /\{\} P c Q, A$

$\langle proof \rangle$

**lemma** *MergeGuardsI-sound*:

**assumes** *valid*:  $\forall n. \Gamma, \Theta \models n: /_F P c Q, A$

**shows**  $\Gamma, \Theta \models n: /_F P \text{ merge-guards } c Q, A$

$\langle proof \rangle$

**lemma** *MergeGuardsI*:

**assumes** *deriv*:  $\Gamma, \Theta \vdash /_F P c Q, A$

**shows**  $\Gamma, \Theta \vdash /_F P \text{ merge-guards } c Q, A$

$\langle proof \rangle$

**lemma** *MergeGuardsD-sound*:

**assumes** *valid*:  $\forall n. \Gamma, \Theta \models n: /_F P \text{ merge-guards } c Q, A$

**shows**  $\Gamma, \Theta \models n: /_F P c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *MergeGuardsD*:  
**assumes**  $\text{deriv}: \Gamma, \Theta \vdash /_F P \text{ merge-guards } c Q, A$   
**shows**  $\Gamma, \Theta \vdash /_F P c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *SubsetGuards-sound*:  
**assumes**  $c-c': c \subseteq_g c'$   
**assumes**  $\text{valid}: \forall n. \Gamma, \Theta \models n: / \{ \} P c' Q, A$   
**shows**  $\Gamma, \Theta \models n: / \{ \} P c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *SubsetGuards*:  
**assumes**  $c-c': c \subseteq_g c'$   
**assumes**  $\text{deriv}: \Gamma, \Theta \vdash / \{ \} P c' Q, A$   
**shows**  $\Gamma, \Theta \vdash / \{ \} P c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *NormalizeD-sound*:  
**assumes**  $\text{valid}: \forall n. \Gamma, \Theta \models n: /_F P (\text{normalize } c) Q, A$   
**shows**  $\Gamma, \Theta \models n: /_F P c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *NormalizeD*:  
**assumes**  $\text{deriv}: \Gamma, \Theta \vdash /_F P (\text{normalize } c) Q, A$   
**shows**  $\Gamma, \Theta \vdash /_F P c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *NormalizeI-sound*:  
**assumes**  $\text{valid}: \forall n. \Gamma, \Theta \models n: /_F P c Q, A$   
**shows**  $\Gamma, \Theta \models n: /_F P (\text{normalize } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *NormalizeI*:  
**assumes**  $\text{deriv}: \Gamma, \Theta \vdash /_F P c Q, A$   
**shows**  $\Gamma, \Theta \vdash /_F P (\text{normalize } c) Q, A$   
 $\langle \text{proof} \rangle$

### 5.3.7 Restricting the Procedure Environment

**lemma** *nvalid-restrict-to-nvalid*:  
**assumes**  $\text{valid-c}: \Gamma |_M \models n: /_F P c Q, A$   
**shows**  $\Gamma \models n: /_F P c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *valid-restrict-to-valid*:  
**assumes** *valid-c*:  $\Gamma \mid_M \Vdash_F P \text{ c } Q, A$   
**shows**  $\Gamma \Vdash_F P \text{ c } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *augment-procs*:  
**assumes** *deriv-c*:  $\Gamma \mid_M, \{\} \vdash_F P \text{ c } Q, A$   
**shows**  $\Gamma, \{\} \vdash_F P \text{ c } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *augment-Faults*:  
**assumes** *deriv-c*:  $\Gamma, \{\} \vdash_F P \text{ c } Q, A$   
**assumes** *F*:  $F \subseteq F'$   
**shows**  $\Gamma, \{\} \vdash_{F'} P \text{ c } Q, A$   
 $\langle \text{proof} \rangle$

**end**

## 6 Derived Hoare Rules for Partial Correctness

**theory** *HoarePartial* **imports** *HoarePartialProps* **begin**

**lemma** *conseq-no-aux*:  
 $\llbracket \Gamma, \Theta \vdash_F P' \text{ c } Q', A';$   
 $\forall s. s \in P \longrightarrow (s \in P' \wedge (Q' \subseteq Q) \wedge (A' \subseteq A)) \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash_F P \text{ c } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *conseq-exploit-pre*:  
 $\llbracket \forall s \in P. \Gamma, \Theta \vdash_F (\{s\} \cap P) \text{ c } Q, A \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash_F P \text{ c } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *conseq*:  $\llbracket \forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{ c } (Q' Z), (A' Z);$   
 $\forall s. s \in P \longrightarrow (\exists Z. s \in P' Z \wedge (Q' Z \subseteq Q) \wedge (A' Z \subseteq A)) \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash_F P \text{ c } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *LEM*:  $\llbracket \forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{ c } (Q' Z), (A' Z);$   
 $P \subseteq \{s. \exists Z. s \in P' Z \wedge (Q' Z \subseteq Q) \wedge (A' Z \subseteq A)\} \rrbracket$   
 $\implies$

$\Gamma, \Theta \vdash_F P \text{ (lem } x \text{ c) } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *LemAnno*:

**assumes** *conseq*:  $P \subseteq \{s. \exists Z. s \in P' Z \wedge (\forall t. t \in Q' Z \longrightarrow t \in Q) \wedge (\forall t. t \in A' Z \longrightarrow t \in A)\}$

**assumes** *lem*:  $\forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{ c } (Q' Z), (A' Z)$

**shows**  $\Gamma, \Theta \vdash_F P \text{ (lem } x \text{ c) } Q, A$

$\langle \text{proof} \rangle$

**lemma** *LemAnnoNoAbrupt*:

**assumes** *conseq*:  $P \subseteq \{s. \exists Z. s \in P' Z \wedge (\forall t. t \in Q' Z \longrightarrow t \in Q)\}$

**assumes** *lem*:  $\forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{ c } (Q' Z), \{\}$

**shows**  $\Gamma, \Theta \vdash_F P \text{ (lem } x \text{ c) } Q, \{\}$

$\langle \text{proof} \rangle$

**lemma** *TrivPost*:  $\forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{ c } (Q' Z), (A' Z)$

$\implies$   
 $\forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{ c } UNIV, UNIV$

$\langle \text{proof} \rangle$

**lemma** *TrivPostNoAbr*:  $\forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{ c } (Q' Z), \{\}$

$\implies$   
 $\forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{ c } UNIV, \{\}$

$\langle \text{proof} \rangle$

**lemma** *conseq-under-new-pre*:  $[\Gamma, \Theta \vdash_F P' \text{ c } Q', A';$

$\forall s \in P. s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A]$

$\implies \Gamma, \Theta \vdash_F P \text{ c } Q, A$

$\langle \text{proof} \rangle$

**lemma** *conseq-Kleymann*:  $[\forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{ c } (Q' Z), (A' Z);$

$\forall s \in P. (\exists Z. s \in P' Z \wedge (Q' Z \subseteq Q) \wedge (A' Z \subseteq A))]$

$\implies$   
 $\Gamma, \Theta \vdash_F P \text{ c } Q, A$

$\langle \text{proof} \rangle$

**lemma** *DynComConseq*:

**assumes**  $P \subseteq \{s. \exists P' Q' A'. \Gamma, \Theta \vdash_F P' \text{ (c } s) Q', A' \wedge P \subseteq P' \wedge Q' \subseteq Q \wedge A' \subseteq A\}$

**shows**  $\Gamma, \Theta \vdash_F P \text{ DynCom } \text{c } Q, A$

$\langle \text{proof} \rangle$

**lemma** *SpecAnno*:

**assumes** *consequence*:  $P \subseteq \{s. (\exists Z. s \in P' Z \wedge (Q' Z \subseteq Q) \wedge (A' Z \subseteq A))\}$

**assumes** *spec*:  $\forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{ (c } Z) (Q' Z), (A' Z)$

**assumes** *bdy-constant*:  $\forall Z. c \ Z = c \text{ undefined}$

**shows**  $\Gamma, \Theta \vdash_F P (\text{specAnno } P' c Q' A') Q, A$   
 ⟨proof⟩

**lemma** *SpecAnno'*:

$\llbracket P \subseteq \{s. \exists Z. s \in P' Z \wedge$   
 $(\forall t. t \in Q' Z \longrightarrow t \in Q) \wedge (\forall t. t \in A' Z \longrightarrow t \in A)\};$   
 $\forall Z. \Gamma, \Theta \vdash_F (P' Z) (c Z) (Q' Z), (A' Z);$   
 $\forall Z. c Z = c \text{ undefined}$   
 $\rrbracket \Longrightarrow$   
 $\Gamma, \Theta \vdash_F P (\text{specAnno } P' c Q' A') Q, A$   
 ⟨proof⟩

**lemma** *SpecAnnoNoAbrupt*:

$\llbracket P \subseteq \{s. \exists Z. s \in P' Z \wedge$   
 $(\forall t. t \in Q' Z \longrightarrow t \in Q)\};$   
 $\forall Z. \Gamma, \Theta \vdash_F (P' Z) (c Z) (Q' Z), \{\};$   
 $\forall Z. c Z = c \text{ undefined}$   
 $\rrbracket \Longrightarrow$   
 $\Gamma, \Theta \vdash_F P (\text{specAnno } P' c Q' (\lambda s. \{\})) Q, A$   
 ⟨proof⟩

**lemma** *Skip*:  $P \subseteq Q \Longrightarrow \Gamma, \Theta \vdash_F P \text{ Skip } Q, A$   
 ⟨proof⟩

**lemma** *Basic*:  $P \subseteq \{s. (f s) \in Q\} \Longrightarrow \Gamma, \Theta \vdash_F P (\text{Basic } f) Q, A$   
 ⟨proof⟩

**lemma** *BasicCond*:

$\llbracket P \subseteq \{s. (b s \longrightarrow f s \in Q) \wedge (\neg b s \longrightarrow g s \in Q)\} \rrbracket \Longrightarrow$   
 $\Gamma, \Theta \vdash_F P \text{ Basic } (\lambda s. \text{if } b s \text{ then } f s \text{ else } g s) Q, A$   
 ⟨proof⟩

**lemma** *Spec*:  $P \subseteq \{s. (\forall t. (s, t) \in r \longrightarrow t \in Q) \wedge (\exists t. (s, t) \in r)\}$   
 $\Longrightarrow \Gamma, \Theta \vdash_F P (\text{Spec } r) Q, A$   
 ⟨proof⟩

**lemma** *SpecIf*:

$\llbracket P \subseteq \{s. (b s \longrightarrow f s \in Q) \wedge (\neg b s \longrightarrow g s \in Q \wedge h s \in Q)\} \rrbracket \Longrightarrow$   
 $\Gamma, \Theta \vdash_F P \text{ Spec } (\text{if-rel } b f g h) Q, A$   
 ⟨proof⟩

**lemma** *Seq* [*trans, intro?*]:

$\llbracket \Gamma, \Theta \vdash_F P c_1 R, A; \Gamma, \Theta \vdash_F R c_2 Q, A \rrbracket \Longrightarrow \Gamma, \Theta \vdash_F P (\text{Seq } c_1 c_2) Q, A$   
 ⟨proof⟩

**lemma** *SeqSame*:

$$\llbracket \Gamma, \Theta \vdash /_F P \ c_1 \ Q, A; \Gamma, \Theta \vdash /_F Q \ c_2 \ Q, A \rrbracket \Longrightarrow \Gamma, \Theta \vdash /_F P \ (\text{Seq } c_1 \ c_2) \ Q, A$$

*<proof>*

**lemma** *SeqSwap*:

$$\llbracket \Gamma, \Theta \vdash /_F R \ c_2 \ Q, A; \Gamma, \Theta \vdash /_F P \ c_1 \ R, A \rrbracket \Longrightarrow \Gamma, \Theta \vdash /_F P \ (\text{Seq } c_1 \ c_2) \ Q, A$$

*<proof>*

**lemma** *BSeq*:

$$\llbracket \Gamma, \Theta \vdash /_F P \ c_1 \ R, A; \Gamma, \Theta \vdash /_F R \ c_2 \ Q, A \rrbracket \Longrightarrow \Gamma, \Theta \vdash /_F P \ (\text{bseq } c_1 \ c_2) \ Q, A$$

*<proof>*

**lemma** *BSeqSame*:

$$\llbracket \Gamma, \Theta \vdash /_F P \ c_1 \ Q, A; \Gamma, \Theta \vdash /_F Q \ c_2 \ Q, A \rrbracket \Longrightarrow \Gamma, \Theta \vdash /_F P \ (\text{bseq } c_1 \ c_2) \ Q, A$$

*<proof>*

**lemma** *Cond*:

$$\text{assumes } wp: P \subseteq \{s. (s \in b \longrightarrow s \in P_1) \wedge (s \notin b \longrightarrow s \in P_2)\}$$

$$\text{assumes } deriv\text{-}c1: \Gamma, \Theta \vdash /_F P_1 \ c_1 \ Q, A$$

$$\text{assumes } deriv\text{-}c2: \Gamma, \Theta \vdash /_F P_2 \ c_2 \ Q, A$$

$$\text{shows } \Gamma, \Theta \vdash /_F P \ (\text{Cond } b \ c_1 \ c_2) \ Q, A$$

*<proof>*

**lemma** *CondSwap*:

$$\llbracket \Gamma, \Theta \vdash /_F P_1 \ c_1 \ Q, A; \Gamma, \Theta \vdash /_F P_2 \ c_2 \ Q, A; P \subseteq \{s. (s \in b \longrightarrow s \in P_1) \wedge (s \notin b \longrightarrow s \in P_2)\} \rrbracket$$

$$\Longrightarrow$$

$$\Gamma, \Theta \vdash /_F P \ (\text{Cond } b \ c_1 \ c_2) \ Q, A$$

*<proof>*

**lemma** *Cond'*:

$$\llbracket P \subseteq \{s. (b \subseteq P_1) \wedge (- b \subseteq P_2)\}; \Gamma, \Theta \vdash /_F P_1 \ c_1 \ Q, A; \Gamma, \Theta \vdash /_F P_2 \ c_2 \ Q, A \rrbracket$$

$$\Longrightarrow$$

$$\Gamma, \Theta \vdash /_F P \ (\text{Cond } b \ c_1 \ c_2) \ Q, A$$

*<proof>*

**lemma** *CondInv*:

$$\text{assumes } wp: P \subseteq Q$$

$$\text{assumes } inv: Q \subseteq \{s. (s \in b \longrightarrow s \in P_1) \wedge (s \notin b \longrightarrow s \in P_2)\}$$

$$\text{assumes } deriv\text{-}c1: \Gamma, \Theta \vdash /_F P_1 \ c_1 \ Q, A$$

$$\text{assumes } deriv\text{-}c2: \Gamma, \Theta \vdash /_F P_2 \ c_2 \ Q, A$$

$$\text{shows } \Gamma, \Theta \vdash /_F P \ (\text{Cond } b \ c_1 \ c_2) \ Q, A$$

*<proof>*

**lemma** *CondInv'*:

**assumes**  $wp: P \subseteq I$   
**assumes**  $inv: I \subseteq \{s. (s \in b \longrightarrow s \in P_1) \wedge (s \notin b \longrightarrow s \in P_2)\}$   
**assumes**  $wp': I \subseteq Q$   
**assumes**  $deriv-c1: \Gamma, \Theta \vdash_{/F} P_1 \ c_1 \ I, A$   
**assumes**  $deriv-c2: \Gamma, \Theta \vdash_{/F} P_2 \ c_2 \ I, A$   
**shows**  $\Gamma, \Theta \vdash_{/F} P \ (Cond \ b \ c_1 \ c_2) \ Q, A$   
 $\langle proof \rangle$

**lemma** *switchNil*:  
 $P \subseteq Q \implies \Gamma, \Theta \vdash_{/F} P \ (switch \ v \ []) \ Q, A$   
 $\langle proof \rangle$

**lemma** *switchCons*:  
 $\llbracket P \subseteq \{s. (v \ s \in V \longrightarrow s \in P_1) \wedge (v \ s \notin V \longrightarrow s \in P_2)\};$   
 $\Gamma, \Theta \vdash_{/F} P_1 \ c \ Q, A;$   
 $\Gamma, \Theta \vdash_{/F} P_2 \ (switch \ v \ vs) \ Q, A \rrbracket$   
 $\implies \Gamma, \Theta \vdash_{/F} P \ (switch \ v \ ((V, c) \# vs)) \ Q, A$   
 $\langle proof \rangle$

**lemma** *Guard*:  
 $\llbracket P \subseteq g \cap R; \Gamma, \Theta \vdash_{/F} R \ c \ Q, A \rrbracket$   
 $\implies \Gamma, \Theta \vdash_{/F} P \ (Guard \ f \ g \ c) \ Q, A$   
 $\langle proof \rangle$

**lemma** *GuardSwap*:  
 $\llbracket \Gamma, \Theta \vdash_{/F} R \ c \ Q, A; P \subseteq g \cap R \rrbracket$   
 $\implies \Gamma, \Theta \vdash_{/F} P \ (Guard \ f \ g \ c) \ Q, A$   
 $\langle proof \rangle$

**lemma** *Guarantee*:  
 $\llbracket P \subseteq \{s. s \in g \longrightarrow s \in R\}; \Gamma, \Theta \vdash_{/F} R \ c \ Q, A; f \in F \rrbracket$   
 $\implies \Gamma, \Theta \vdash_{/F} P \ (Guard \ f \ g \ c) \ Q, A$   
 $\langle proof \rangle$

**lemma** *GuaranteeSwap*:  
 $\llbracket \Gamma, \Theta \vdash_{/F} R \ c \ Q, A; P \subseteq \{s. s \in g \longrightarrow s \in R\}; f \in F \rrbracket$   
 $\implies \Gamma, \Theta \vdash_{/F} P \ (Guard \ f \ g \ c) \ Q, A$   
 $\langle proof \rangle$

**lemma** *GuardStrip*:  
 $\llbracket P \subseteq R; \Gamma, \Theta \vdash_{/F} R \ c \ Q, A; f \in F \rrbracket$   
 $\implies \Gamma, \Theta \vdash_{/F} P \ (Guard \ f \ g \ c) \ Q, A$   
 $\langle proof \rangle$

**lemma** *GuardStripSame*:

$$\begin{aligned} & \llbracket \Gamma, \Theta \vdash /_F P \ c \ Q, A; f \in F \rrbracket \\ & \implies \Gamma, \Theta \vdash /_F P \ (\text{Guard } f \ g \ c) \ Q, A \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *GuardStripSwap*:

$$\begin{aligned} & \llbracket \Gamma, \Theta \vdash /_F R \ c \ Q, A; P \subseteq R; f \in F \rrbracket \\ & \implies \Gamma, \Theta \vdash /_F P \ (\text{Guard } f \ g \ c) \ Q, A \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *GuaranteeStrip*:

$$\begin{aligned} & \llbracket P \subseteq R; \Gamma, \Theta \vdash /_F R \ c \ Q, A; f \in F \rrbracket \\ & \implies \Gamma, \Theta \vdash /_F P \ (\text{guaranteeStrip } f \ g \ c) \ Q, A \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *GuaranteeStripSwap*:

$$\begin{aligned} & \llbracket \Gamma, \Theta \vdash /_F R \ c \ Q, A; P \subseteq R; f \in F \rrbracket \\ & \implies \Gamma, \Theta \vdash /_F P \ (\text{guaranteeStrip } f \ g \ c) \ Q, A \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *GuaranteeAsGuard*:

$$\begin{aligned} & \llbracket P \subseteq g \cap R; \Gamma, \Theta \vdash /_F R \ c \ Q, A \rrbracket \\ & \implies \Gamma, \Theta \vdash /_F P \ (\text{guaranteeStrip } f \ g \ c) \ Q, A \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *GuaranteeAsGuardSwap*:

$$\begin{aligned} & \llbracket \Gamma, \Theta \vdash /_F R \ c \ Q, A; P \subseteq g \cap R \rrbracket \\ & \implies \Gamma, \Theta \vdash /_F P \ (\text{guaranteeStrip } f \ g \ c) \ Q, A \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *GuardsNil*:

$$\begin{aligned} & \Gamma, \Theta \vdash /_F P \ c \ Q, A \implies \\ & \Gamma, \Theta \vdash /_F P \ (\text{guards } [] \ c) \ Q, A \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *GuardsCons*:

$$\begin{aligned} & \Gamma, \Theta \vdash /_F P \ \text{Guard } f \ g \ (\text{guards } g \ c) \ Q, A \implies \\ & \Gamma, \Theta \vdash /_F P \ (\text{guards } ((f, g) \# g \ c) \ c) \ Q, A \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *GuardsConsGuaranteeStrip*:

$$\begin{aligned} & \Gamma, \Theta \vdash /_F P \ \text{guaranteeStrip } f \ g \ (\text{guards } g \ c) \ Q, A \implies \\ & \Gamma, \Theta \vdash /_F P \ (\text{guards } (\text{guaranteeStripPair } f \ g \ # g \ c) \ c) \ Q, A \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *While*:

**assumes**  $P-I: P \subseteq I$   
**assumes**  $deriv-body: \Gamma, \Theta \vdash_{/F} (I \cap b) c I, A$   
**assumes**  $I-Q: I \cap -b \subseteq Q$   
**shows**  $\Gamma, \Theta \vdash_{/F} P (whileAnno\ b\ I\ V\ c)\ Q, A$   
 $\langle proof \rangle$

$J$  will be instantiated by tactic with  $gs' \cap I$  for those guards that are not stripped.

**lemma** *WhileAnnoG*:  
 $\Gamma, \Theta \vdash_{/F} P (guards\ gs$   
 $\quad (whileAnno\ b\ J\ V\ (Seq\ c\ (guards\ gs\ Skip))))\ Q, A$   
 $\implies$   
 $\Gamma, \Theta \vdash_{/F} P (whileAnnoG\ gs\ b\ I\ V\ c)\ Q, A$   
 $\langle proof \rangle$

This form stems from *strip-guards*  $F (whileAnnoG\ gs\ b\ I\ V\ c)$

**lemma** *WhileNoGuard'*:  
**assumes**  $P-I: P \subseteq I$   
**assumes**  $deriv-body: \Gamma, \Theta \vdash_{/F} (I \cap b) c I, A$   
**assumes**  $I-Q: I \cap -b \subseteq Q$   
**shows**  $\Gamma, \Theta \vdash_{/F} P (whileAnno\ b\ I\ V\ (Seq\ c\ Skip))\ Q, A$   
 $\langle proof \rangle$

**lemma** *WhileAnnoFix*:  
**assumes**  $consequence: P \subseteq \{s. (\exists Z. s \in I\ Z \wedge (I\ Z \cap -b \subseteq Q))\}$   
**assumes**  $bdy: \forall Z. \Gamma, \Theta \vdash_{/F} (I\ Z \cap b) (c\ Z) (I\ Z), A$   
**assumes**  $bdy-constant: \forall Z. c\ Z = c\ undefined$   
**shows**  $\Gamma, \Theta \vdash_{/F} P (whileAnnoFix\ b\ I\ V\ c)\ Q, A$   
 $\langle proof \rangle$

**lemma** *WhileAnnoFix'*:  
**assumes**  $consequence: P \subseteq \{s. (\exists Z. s \in I\ Z \wedge$   
 $\quad (\forall t. t \in I\ Z \cap -b \longrightarrow t \in Q))\}$   
**assumes**  $bdy: \forall Z. \Gamma, \Theta \vdash_{/F} (I\ Z \cap b) (c\ Z) (I\ Z), A$   
**assumes**  $bdy-constant: \forall Z. c\ Z = c\ undefined$   
**shows**  $\Gamma, \Theta \vdash_{/F} P (whileAnnoFix\ b\ I\ V\ c)\ Q, A$   
 $\langle proof \rangle$

**lemma** *WhileAnnoGFix*:  
**assumes** *whileAnnoFix*:  
 $\Gamma, \Theta \vdash_{/F} P (guards\ gs$   
 $\quad (whileAnnoFix\ b\ J\ V\ (\lambda Z. (Seq\ (c\ Z)\ (guards\ gs\ Skip))))\ Q, A$   
**shows**  $\Gamma, \Theta \vdash_{/F} P (whileAnnoGFix\ gs\ b\ I\ V\ c)\ Q, A$   
 $\langle proof \rangle$

**lemma** *Bind*:  
**assumes**  $adapt: P \subseteq \{s. s \in P'\ s\}$

**assumes**  $c: \forall s. \Gamma, \Theta \vdash_F (P' s) (c (e s)) Q, A$   
**shows**  $\Gamma, \Theta \vdash_F P (bind\ e\ c) Q, A$   
 $\langle proof \rangle$

**lemma** *Block-exn*:

**assumes** *adapt*:  $P \subseteq \{s. init\ s \in P' s\}$   
**assumes** *bdy*:  $\forall s. \Gamma, \Theta \vdash_F (P' s) bdy \{t. return\ s\ t \in R\ s\ t\}, \{t. result\text{-}exn\ (return\ s\ t)\ t \in A\}$   
**assumes**  $c: \forall s\ t. \Gamma, \Theta \vdash_F (R\ s\ t) (c\ s\ t) Q, A$   
**shows**  $\Gamma, \Theta \vdash_F P (block\text{-}exn\ init\ bdy\ return\ result\text{-}exn\ c) Q, A$   
 $\langle proof \rangle$

**lemma** *Block*:

**assumes** *adapt*:  $P \subseteq \{s. init\ s \in P' s\}$   
**assumes** *bdy*:  $\forall s. \Gamma, \Theta \vdash_F (P' s) bdy \{t. return\ s\ t \in R\ s\ t\}, \{t. return\ s\ t \in A\}$   
**assumes**  $c: \forall s\ t. \Gamma, \Theta \vdash_F (R\ s\ t) (c\ s\ t) Q, A$   
**shows**  $\Gamma, \Theta \vdash_F P (block\ init\ bdy\ return\ c) Q, A$   
 $\langle proof \rangle$

**lemma** *BlockSwap*:

**assumes**  $c: \forall s\ t. \Gamma, \Theta \vdash_F (R\ s\ t) (c\ s\ t) Q, A$   
**assumes** *bdy*:  $\forall s. \Gamma, \Theta \vdash_F (P' s) bdy \{t. return\ s\ t \in R\ s\ t\}, \{t. return\ s\ t \in A\}$   
**assumes** *adapt*:  $P \subseteq \{s. init\ s \in P' s\}$   
**shows**  $\Gamma, \Theta \vdash_F P (block\ init\ bdy\ return\ c) Q, A$   
 $\langle proof \rangle$

**lemma** *Block-exnSpec*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. init\ s \in P' Z \wedge$   
 $(\forall t. t \in Q' Z \longrightarrow return\ s\ t \in R\ s\ t) \wedge$   
 $(\forall t. t \in A' Z \longrightarrow (result\text{-}exn\ (return\ s\ t)\ t) \in A)\}$   
**assumes**  $c: \forall s\ t. \Gamma, \Theta \vdash_F (R\ s\ t) (c\ s\ t) Q, A$   
**assumes** *bdy*:  $\forall Z. \Gamma, \Theta \vdash_F (P' Z) bdy (Q' Z), (A' Z)$   
**shows**  $\Gamma, \Theta \vdash_F P (block\text{-}exn\ init\ bdy\ return\ result\text{-}exn\ c) Q, A$   
 $\langle proof \rangle$

**lemma** *BlockSpec*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. init\ s \in P' Z \wedge$   
 $(\forall t. t \in Q' Z \longrightarrow return\ s\ t \in R\ s\ t) \wedge$   
 $(\forall t. t \in A' Z \longrightarrow return\ s\ t \in A)\}$   
**assumes**  $c: \forall s\ t. \Gamma, \Theta \vdash_F (R\ s\ t) (c\ s\ t) Q, A$   
**assumes** *bdy*:  $\forall Z. \Gamma, \Theta \vdash_F (P' Z) bdy (Q' Z), (A' Z)$   
**shows**  $\Gamma, \Theta \vdash_F P (block\ init\ bdy\ return\ c) Q, A$   
 $\langle proof \rangle$

**lemma** *Throw*:  $P \subseteq A \implies \Gamma, \Theta \vdash_F P \text{ Throw } Q, A$   
 ⟨proof⟩

**lemmas** *Catch = hoarep.Catch*

**lemma** *CatchSwap*:  $\llbracket \Gamma, \Theta \vdash_F R \ c_2 \ Q, A; \Gamma, \Theta \vdash_F P \ c_1 \ Q, R \rrbracket \implies \Gamma, \Theta \vdash_F P \text{ Catch}$   
 $c_1 \ c_2 \ Q, A$   
 ⟨proof⟩

**lemma** *CatchSame*:  $\llbracket \Gamma, \Theta \vdash_F P \ c_1 \ Q, A; \Gamma, \Theta \vdash_F A \ c_2 \ Q, A \rrbracket \implies \Gamma, \Theta \vdash_F P \text{ Catch}$   
 $c_1 \ c_2 \ Q, A$   
 ⟨proof⟩

**lemma** *raise*:  $P \subseteq \{s. f \ s \in A\} \implies \Gamma, \Theta \vdash_F P \text{ raise } f \ Q, A$   
 ⟨proof⟩

**lemma** *condCatch*:  $\llbracket \Gamma, \Theta \vdash_F P \ c_1 \ Q, ((b \cap R) \cup (-b \cap A)); \Gamma, \Theta \vdash_F R \ c_2 \ Q, A \rrbracket$   
 $\implies \Gamma, \Theta \vdash_F P \text{ condCatch } c_1 \ b \ c_2 \ Q, A$   
 ⟨proof⟩

**lemma** *condCatchSwap*:  $\llbracket \Gamma, \Theta \vdash_F R \ c_2 \ Q, A; \Gamma, \Theta \vdash_F P \ c_1 \ Q, ((b \cap R) \cup (-b \cap A)) \rrbracket$   
 $\implies \Gamma, \Theta \vdash_F P \text{ condCatch } c_1 \ b \ c_2 \ Q, A$   
 ⟨proof⟩

**lemma** *condCatchSame*:

**assumes** *c1*:  $\Gamma, \Theta \vdash_F P \ c_1 \ Q, A$

**assumes** *c2*:  $\Gamma, \Theta \vdash_F A \ c_2 \ Q, A$

**shows**  $\Gamma, \Theta \vdash_F P \text{ condCatch } c_1 \ b \ c_2 \ Q, A$

⟨proof⟩

**lemma** *ProcSpec*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' \ Z \wedge$   
 $(\forall t. t \in Q' \ Z \longrightarrow \text{return } s \ t \in R \ s \ t) \wedge$   
 $(\forall t. t \in A' \ Z \longrightarrow \text{return } s \ t \in A)\}$

**assumes** *c*:  $\forall s \ t. \Gamma, \Theta \vdash_F (R \ s \ t) \ (c \ s \ t) \ Q, A$

**assumes** *p*:  $\forall Z. \Gamma, \Theta \vdash_F (P' \ Z) \ \text{Call } p \ (Q' \ Z), (A' \ Z)$

**shows**  $\Gamma, \Theta \vdash_F P \ (\text{call } \text{init } p \ \text{return } c) \ Q, A$

⟨proof⟩

**lemma** *Proc-exnSpec*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' \ Z \wedge$   
 $(\forall t. t \in Q' \ Z \longrightarrow \text{return } s \ t \in R \ s \ t) \wedge$   
 $(\forall t. t \in A' \ Z \longrightarrow \text{result-exn } (\text{return } s \ t) \ t \in A)\}$

**assumes** *c*:  $\forall s \ t. \Gamma, \Theta \vdash_F (R \ s \ t) \ (c \ s \ t) \ Q, A$

**assumes** *p*:  $\forall Z. \Gamma, \Theta \vdash_F (P' \ Z) \ \text{Call } p \ (Q' \ Z), (A' \ Z)$

**shows**  $\Gamma, \Theta \vdash_F P \ (\text{call-exn } \text{init } p \ \text{return } \text{result-exn } c) \ Q, A$

$\langle \text{proof} \rangle$

**lemma** *ProcSpec'*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall t \in Q' Z. \text{return } s t \in R s t) \wedge$   
 $(\forall t \in A' Z. \text{return } s t \in A)\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{/F} (R s t) (c s t) Q, A$

**assumes** *p*:  $\forall Z. \Gamma, \Theta \vdash_{/F} (P' Z) \text{ Call } p (Q' Z), (A' Z)$

**shows**  $\Gamma, \Theta \vdash_{/F} P (\text{call init } p \text{ return } c) Q, A$

$\langle \text{proof} \rangle$

**lemma** *Proc-exnSpecNoAbrupt*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall t. t \in Q' Z \longrightarrow \text{return } s t \in R s t)\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{/F} (R s t) (c s t) Q, A$

**assumes** *p*:  $\forall Z. \Gamma, \Theta \vdash_{/F} (P' Z) \text{ Call } p (Q' Z), \{\}$

**shows**  $\Gamma, \Theta \vdash_{/F} P (\text{call-exn init } p \text{ return result-exn } c) Q, A$

$\langle \text{proof} \rangle$

**lemma** *ProcSpecNoAbrupt*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall t. t \in Q' Z \longrightarrow \text{return } s t \in R s t)\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{/F} (R s t) (c s t) Q, A$

**assumes** *p*:  $\forall Z. \Gamma, \Theta \vdash_{/F} (P' Z) \text{ Call } p (Q' Z), \{\}$

**shows**  $\Gamma, \Theta \vdash_{/F} P (\text{call init } p \text{ return } c) Q, A$

$\langle \text{proof} \rangle$

**lemma** *FCall*:

$\Gamma, \Theta \vdash_{/F} P (\text{call init } p \text{ return } (\lambda s t. c (\text{result } t))) Q, A$

$\implies \Gamma, \Theta \vdash_{/F} P (\text{fcall init } p \text{ return result } c) Q, A$

$\langle \text{proof} \rangle$

**lemma** *ProcRec*:

**assumes** *deriv-bodies*:

$\forall p \in \text{Procs.}$

$\forall Z. \Gamma, \Theta \cup (\bigcup p \in \text{Procs.} \bigcup Z. \{(P p Z, p, Q p Z, A p Z)\})$   
 $\vdash_{/F} (P p Z) (\text{the } (\Gamma p)) (Q p Z), (A p Z)$

**assumes** *Procs-defined*:  $\text{Procs} \subseteq \text{dom } \Gamma$

**shows**  $\forall p \in \text{Procs.} \forall Z. \Gamma, \Theta \vdash_{/F} (P p Z) \text{ Call } p (Q p Z), (A p Z)$

$\langle \text{proof} \rangle$

**lemma** *ProcRec'*:

**assumes** *ctxt*:  $\Theta' = \Theta \cup (\bigcup p \in \text{Procs.} \bigcup Z. \{(P p Z, p, Q p Z, A p Z)\})$

**assumes** *deriv-bodies*:

$\forall p \in \text{Procs.} \forall Z. \Gamma, \Theta' \vdash_{/F} (P p Z) (\text{the } (\Gamma p)) (Q p Z), (A p Z)$

**assumes** *Procs-defined*:  $\text{Procs} \subseteq \text{dom } \Gamma$

**shows**  $\forall p \in \text{Procs}. \forall Z. \Gamma, \Theta \vdash_{/F} (P \ p \ Z) \text{ Call } p \ (Q \ p \ Z), (A \ p \ Z)$   
 $\langle \text{proof} \rangle$

**lemma ProcRecList:**

**assumes deriv-bodies:**

$\forall p \in \text{set Procs}.$

$\forall Z. \Gamma, \Theta \cup (\bigcup p \in \text{set Procs}. \bigcup Z. \{(P \ p \ Z, p, Q \ p \ Z, A \ p \ Z)\})$   
 $\vdash_{/F} (P \ p \ Z) \text{ (the } (\Gamma \ p)) \ (Q \ p \ Z), (A \ p \ Z)$

**assumes dist: distinct Procs**

**assumes Procs-defined: set Procs  $\subseteq$  dom  $\Gamma$**

**shows**  $\forall p \in \text{set Procs}. \forall Z. \Gamma, \Theta \vdash_{/F} (P \ p \ Z) \text{ Call } p \ (Q \ p \ Z), (A \ p \ Z)$   
 $\langle \text{proof} \rangle$

**lemma ProcRecSpecs:**

$\llbracket \forall (P, p, Q, A) \in \text{Specs}. \Gamma, \Theta \cup \text{Specs} \vdash_{/F} P \text{ (the } (\Gamma \ p)) \ Q, A;$

$\forall (P, p, Q, A) \in \text{Specs}. p \in \text{dom } \Gamma \rrbracket$

$\implies \forall (P, p, Q, A) \in \text{Specs}. \Gamma, \Theta \vdash_{/F} P \text{ (Call } p) \ Q, A$

$\langle \text{proof} \rangle$

**lemma ProcRec1:**

**assumes deriv-body:**

$\forall Z. \Gamma, \Theta \cup (\bigcup Z. \{(P \ Z, p, Q \ Z, A \ Z)\}) \vdash_{/F} (P \ Z) \text{ (the } (\Gamma \ p)) \ (Q \ Z), (A \ Z)$

**assumes p-defined:  $p \in \text{dom } \Gamma$**

**shows**  $\forall Z. \Gamma, \Theta \vdash_{/F} (P \ Z) \text{ Call } p \ (Q \ Z), (A \ Z)$

$\langle \text{proof} \rangle$

**lemma ProcNoRec1:**

**assumes deriv-body:**

$\forall Z. \Gamma, \Theta \vdash_{/F} (P \ Z) \text{ (the } (\Gamma \ p)) \ (Q \ Z), (A \ Z)$

**assumes p-def:  $p \in \text{dom } \Gamma$**

**shows**  $\forall Z. \Gamma, \Theta \vdash_{/F} (P \ Z) \text{ Call } p \ (Q \ Z), (A \ Z)$

$\langle \text{proof} \rangle$

**lemma ProcBody:**

**assumes WP:  $P \subseteq P'$**

**assumes deriv-body:  $\Gamma, \Theta \vdash_{/F} P' \text{ body } Q, A$**

**assumes body:  $\Gamma \ p = \text{Some body}$**

**shows**  $\Gamma, \Theta \vdash_{/F} P \text{ Call } p \ Q, A$

$\langle \text{proof} \rangle$

**lemma CallBody:**

**assumes adapt:  $P \subseteq \{s. \text{init } s \in P' \ s\}$**

**assumes bdy:  $\forall s. \Gamma, \Theta \vdash_{/F} (P' \ s) \text{ body } \{t. \text{return } s \ t \in R \ s \ t\}, \{t. \text{return } s \ t \in A\}$**

**assumes c:  $\forall s \ t. \Gamma, \Theta \vdash_{/F} (R \ s \ t) \ (c \ s \ t) \ Q, A$**

**assumes body:  $\Gamma \ p = \text{Some body}$**

**shows**  $\Gamma, \Theta \vdash_F P \text{ (call init p return c) } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *Call-exnBody*:

**assumes** *adapt*:  $P \subseteq \{s. \text{init } s \in P' s\}$

**assumes** *bdy*:  $\forall s. \Gamma, \Theta \vdash_F (P' s) \text{ body } \{t. \text{return } s t \in R s t\}, \{t. \text{result-exn (return } s t) t \in A\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_F (R s t) (c s t) Q, A$

**assumes** *body*:  $\Gamma p = \text{Some body}$

**shows**  $\Gamma, \Theta \vdash_F P \text{ (call-exn init p return result-exn c) } Q, A$

$\langle \text{proof} \rangle$

**lemmas** *ProcModifyReturn* = *HoarePartialProps.ProcModifyReturn*

**lemmas** *ProcModifyReturnSameFaults* = *HoarePartialProps.ProcModifyReturnSameFaults*

**lemmas** *Proc-exnModifyReturn* = *HoarePartialProps.Proc-exnModifyReturn*

**lemmas** *Proc-exnModifyReturnSameFaults* = *HoarePartialProps.Proc-exnModifyReturnSameFaults*

**lemma** *Proc-exnModifyReturnNoAbr*:

**assumes** *spec*:  $\Gamma, \Theta \vdash_F P \text{ (call-exn init p return' result-exn c) } Q, A$

**assumes** *result-conform*:

$\forall s t. t \in \text{Modif (init s)} \longrightarrow (\text{return' s t}) = (\text{return s t})$

**assumes** *modifies-spec*:

$\forall \sigma. \Gamma, \Theta \vdash_{UNIV} \{\sigma\} \text{ Call } p \text{ (Modif } \sigma), \{\}$

**shows**  $\Gamma, \Theta \vdash_F P \text{ (call-exn init p return result-exn c) } Q, A$

$\langle \text{proof} \rangle$

**lemma** *ProcModifyReturnNoAbr*:

**assumes** *spec*:  $\Gamma, \Theta \vdash_F P \text{ (call init p return' c) } Q, A$

**assumes** *result-conform*:

$\forall s t. t \in \text{Modif (init s)} \longrightarrow (\text{return' s t}) = (\text{return s t})$

**assumes** *modifies-spec*:

$\forall \sigma. \Gamma, \Theta \vdash_{UNIV} \{\sigma\} \text{ Call } p \text{ (Modif } \sigma), \{\}$

**shows**  $\Gamma, \Theta \vdash_F P \text{ (call init p return c) } Q, A$

$\langle \text{proof} \rangle$

**lemma** *Proc-exnModifyReturnNoAbrSameFaults*:

**assumes** *spec*:  $\Gamma, \Theta \vdash_F P \text{ (call-exn init p return' result-exn c) } Q, A$

**assumes** *result-conform*:

$\forall s t. t \in \text{Modif (init s)} \longrightarrow (\text{return' s t}) = (\text{return s t})$

**assumes** *modifies-spec*:

$\forall \sigma. \Gamma, \Theta \vdash_F \{\sigma\} \text{ Call } p \text{ (Modif } \sigma), \{\}$

**shows**  $\Gamma, \Theta \vdash_F P \text{ (call-exn init p return result-exn c) } Q, A$

$\langle \text{proof} \rangle$

**lemma** *ProcModifyReturnNoAbrSameFaults*:

**assumes** *spec*:  $\Gamma, \Theta \vdash_F P \text{ (call init p return' c) } Q, A$

**assumes** *result-conform*:

$\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow (\text{return}' s t) = (\text{return } s t)$   
**assumes** *modifies-spec*:  
 $\forall \sigma. \Gamma, \Theta \vdash_{/F} \{\sigma\} \text{ Call } p (\text{Modif } \sigma), \{\}$   
**shows**  $\Gamma, \Theta \vdash_{/F} P (\text{call init } p \text{ return } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProc-exn*:  
**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' s Z \wedge$   
 $(\forall t. t \in Q' s Z \longrightarrow \text{return } s t \in R s t) \wedge$   
 $(\forall t. t \in A' s Z \longrightarrow \text{result-exn } (\text{return } s t) t \in A)\}$   
**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{/F} (R s t) (c s t) Q, A$   
**assumes** *p*:  $\forall s \in P. \forall Z. \Gamma, \Theta \vdash_{/F} (P' s Z) \text{ Call } (p s) (Q' s Z), (A' s Z)$   
**shows**  $\Gamma, \Theta \vdash_{/F} P \text{ dynCall-exn } f \text{ UNIV } \text{init } p \text{ return } \text{result-exn } c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProc-exn-guards-cons*:  
**assumes** *p*:  $\Gamma, \Theta \vdash_{/F} P \text{ dynCall-exn } f \text{ UNIV } \text{init } p \text{ return } \text{result-exn } c Q, A$   
**shows**  $\Gamma, \Theta \vdash_{/F} (g \cap P) \text{ dynCall-exn } f g \text{init } p \text{ return } \text{result-exn } c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProc*:  
**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' s Z \wedge$   
 $(\forall t. t \in Q' s Z \longrightarrow \text{return } s t \in R s t) \wedge$   
 $(\forall t. t \in A' s Z \longrightarrow \text{return } s t \in A)\}$   
**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{/F} (R s t) (c s t) Q, A$   
**assumes** *p*:  $\forall s \in P. \forall Z. \Gamma, \Theta \vdash_{/F} (P' s Z) \text{ Call } (p s) (Q' s Z), (A' s Z)$   
**shows**  $\Gamma, \Theta \vdash_{/F} P \text{ dynCall } \text{init } p \text{ return } c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProc-exn'*:  
**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' s Z \wedge$   
 $(\forall t \in Q' s Z. \text{return } s t \in R s t) \wedge$   
 $(\forall t \in A' s Z. \text{result-exn } (\text{return } s t) t \in A)\}$   
**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{/F} (R s t) (c s t) Q, A$   
**assumes** *p*:  $\forall s \in P. \forall Z. \Gamma, \Theta \vdash_{/F} (P' s Z) \text{ Call } (p s) (Q' s Z), (A' s Z)$   
**shows**  $\Gamma, \Theta \vdash_{/F} P \text{ dynCall-exn } f \text{ UNIV } \text{init } p \text{ return } \text{result-exn } c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProc'*:  
**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' s Z \wedge$   
 $(\forall t \in Q' s Z. \text{return } s t \in R s t) \wedge$   
 $(\forall t \in A' s Z. \text{return } s t \in A)\}$   
**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{/F} (R s t) (c s t) Q, A$   
**assumes** *p*:  $\forall s \in P. \forall Z. \Gamma, \Theta \vdash_{/F} (P' s Z) \text{ Call } (p s) (Q' s Z), (A' s Z)$   
**shows**  $\Gamma, \Theta \vdash_{/F} P \text{ dynCall } \text{init } p \text{ return } c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProc-exnStaticSpec*:

**assumes** *adapt*:  $P \subseteq \{s. s \in S \wedge (\exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall \tau. \tau \in Q' Z \longrightarrow \text{return } s \tau \in R s \tau) \wedge$   
 $(\forall \tau. \tau \in A' Z \longrightarrow \text{result-exn } (\text{return } s \tau) \tau \in A))\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_F (R s t) (c s t) Q, A$

**assumes** *spec*:  $\forall s \in S. \forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{Call } (p s) (Q' Z), (A' Z)$

**shows**  $\Gamma, \Theta \vdash_F P (\text{dynCall-exn } f \text{ UNIV } \text{init } p \text{ return } \text{result-exn } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProcStaticSpec*:

**assumes** *adapt*:  $P \subseteq \{s. s \in S \wedge (\exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall \tau. \tau \in Q' Z \longrightarrow \text{return } s \tau \in R s \tau) \wedge$   
 $(\forall \tau. \tau \in A' Z \longrightarrow \text{return } s \tau \in A))\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_F (R s t) (c s t) Q, A$

**assumes** *spec*:  $\forall s \in S. \forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{Call } (p s) (Q' Z), (A' Z)$

**shows**  $\Gamma, \Theta \vdash_F P (\text{dynCall } \text{init } p \text{ return } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProc-exnProcPar*:

**assumes** *adapt*:  $P \subseteq \{s. p s = q \wedge (\exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall \tau. \tau \in Q' Z \longrightarrow \text{return } s \tau \in R s \tau) \wedge$   
 $(\forall \tau. \tau \in A' Z \longrightarrow \text{result-exn } (\text{return } s \tau) \tau \in A))\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_F (R s t) (c s t) Q, A$

**assumes** *spec*:  $\forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{Call } q (Q' Z), (A' Z)$

**shows**  $\Gamma, \Theta \vdash_F P (\text{dynCall-exn } f \text{ UNIV } \text{init } p \text{ return } \text{result-exn } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProcProcPar*:

**assumes** *adapt*:  $P \subseteq \{s. p s = q \wedge (\exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall \tau. \tau \in Q' Z \longrightarrow \text{return } s \tau \in R s \tau) \wedge$   
 $(\forall \tau. \tau \in A' Z \longrightarrow \text{return } s \tau \in A))\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_F (R s t) (c s t) Q, A$

**assumes** *spec*:  $\forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{Call } q (Q' Z), (A' Z)$

**shows**  $\Gamma, \Theta \vdash_F P (\text{dynCall } \text{init } p \text{ return } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProc-exnProcParNoAbrupt*:

**assumes** *adapt*:  $P \subseteq \{s. p s = q \wedge (\exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall \tau. \tau \in Q' Z \longrightarrow \text{return } s \tau \in R s \tau))\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_F (R s t) (c s t) Q, A$

**assumes** *spec*:  $\forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{Call } q (Q' Z), \{\}$

**shows**  $\Gamma, \Theta \vdash_F P (\text{dynCall-exn } f \text{ UNIV } \text{init } p \text{ return } \text{result-exn } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProcProcParNoAbrupt*:

**assumes** *adapt*:  $P \subseteq \{s. p s = q \wedge (\exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall \tau. \tau \in Q' Z \longrightarrow \text{return } s \tau \in R s \tau))\}$

**assumes**  $c: \forall s t. \Gamma, \Theta \vdash /_F (R s t) (c s t) Q, A$   
**assumes**  $spec: \forall Z. \Gamma, \Theta \vdash /_F (P' Z) Call q (Q' Z), \{\}$   
**shows**  $\Gamma, \Theta \vdash /_F P (dynCall init p return c) Q, A$   
 $\langle proof \rangle$

**lemma** *DynProc-exnModifyReturnNoAbr*:  
**assumes** *to-prove*:  $\Gamma, \Theta \vdash /_F P (dynCall-exn f g init p return' result-exn c) Q, A$   
**assumes** *ret-nrm-modif*:  $\forall s t. t \in (Modif (init s))$   
 $\longrightarrow return' s t = return s t$   
**assumes** *modif-clause*:  
 $\forall s \in P. \forall \sigma. \Gamma, \Theta \vdash /_{UNIV} \{\sigma\} Call (p s) (Modif \sigma), \{\}$   
**shows**  $\Gamma, \Theta \vdash /_F P (dynCall-exn f g init p return result-exn c) Q, A$   
 $\langle proof \rangle$

**lemma** *DynProcModifyReturnNoAbr*:  
**assumes** *to-prove*:  $\Gamma, \Theta \vdash /_F P (dynCall init p return' c) Q, A$   
**assumes** *ret-nrm-modif*:  $\forall s t. t \in (Modif (init s))$   
 $\longrightarrow return' s t = return s t$   
**assumes** *modif-clause*:  
 $\forall s \in P. \forall \sigma. \Gamma, \Theta \vdash /_{UNIV} \{\sigma\} Call (p s) (Modif \sigma), \{\}$   
**shows**  $\Gamma, \Theta \vdash /_F P (dynCall init p return c) Q, A$   
 $\langle proof \rangle$

**lemma** *ProcDyn-exnModifyReturnNoAbrSameFaults*:  
**assumes** *to-prove*:  $\Gamma, \Theta \vdash /_F P (dynCall-exn f g init p return' result-exn c) Q, A$   
**assumes** *ret-nrm-modif*:  $\forall s t. t \in (Modif (init s))$   
 $\longrightarrow return' s t = return s t$   
**assumes** *modif-clause*:  
 $\forall s \in P. \forall \sigma. \Gamma, \Theta \vdash /_F \{\sigma\} (Call (p s)) (Modif \sigma), \{\}$   
**shows**  $\Gamma, \Theta \vdash /_F P (dynCall-exn f g init p return result-exn c) Q, A$   
 $\langle proof \rangle$

**lemma** *ProcDynModifyReturnNoAbrSameFaults*:  
**assumes** *to-prove*:  $\Gamma, \Theta \vdash /_F P (dynCall init p return' c) Q, A$   
**assumes** *ret-nrm-modif*:  $\forall s t. t \in (Modif (init s))$   
 $\longrightarrow return' s t = return s t$   
**assumes** *modif-clause*:  
 $\forall s \in P. \forall \sigma. \Gamma, \Theta \vdash /_F \{\sigma\} (Call (p s)) (Modif \sigma), \{\}$   
**shows**  $\Gamma, \Theta \vdash /_F P (dynCall init p return c) Q, A$   
 $\langle proof \rangle$

**lemma** *Proc-exnProcParModifyReturn*:  
**assumes**  $q: P \subseteq \{s. p s = q\} \cap P'$   
— *DynProcProcPar* introduces the same constraint as first conjunction in  $P'$ , so the vcg can simplify it.  
**assumes** *to-prove*:  $\Gamma, \Theta \vdash /_F P' (dynCall-exn f g init p return' result-exn c) Q, A$

**assumes** *ret-nrm-modif*:  $\forall s t. t \in (\text{Modif } (\text{init } s))$   
 $\longrightarrow \text{return}' s t = \text{return } s t$   
**assumes** *ret-abr-modif*:  $\forall s t. t \in (\text{ModifAbr } (\text{init } s))$   
 $\longrightarrow \text{result-exn } (\text{return}' s t) t = \text{result-exn } (\text{return } s t) t$   
**assumes** *modif-clause*:  
 $\forall \sigma. \Gamma, \Theta \vdash_{UNIV} \{\sigma\} (\text{Call } q) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\Gamma, \Theta \vdash_F P (\text{dynCall-exn } f g \text{ init } p \text{ return } \text{result-exn } c) Q, A$   
*<proof>*

**lemma** *ProcProcParModifyReturn*:

**assumes**  $q: P \subseteq \{s. p s = q\} \cap P'$   
— *DynProcProcPar* introduces the same constraint as first conjunction in  $P'$ , so the vcg can simplify it.  
**assumes** *to-prove*:  $\Gamma, \Theta \vdash_F P' (\text{dynCall } \text{init } p \text{ return}' c) Q, A$   
**assumes** *ret-nrm-modif*:  $\forall s t. t \in (\text{Modif } (\text{init } s))$   
 $\longrightarrow \text{return}' s t = \text{return } s t$   
**assumes** *ret-abr-modif*:  $\forall s t. t \in (\text{ModifAbr } (\text{init } s))$   
 $\longrightarrow \text{return}' s t = \text{return } s t$   
**assumes** *modif-clause*:  
 $\forall \sigma. \Gamma, \Theta \vdash_{UNIV} \{\sigma\} (\text{Call } q) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\Gamma, \Theta \vdash_F P (\text{dynCall } \text{init } p \text{ return } c) Q, A$   
*<proof>*

**lemma** *Proc-exnProcParModifyReturnSameFaults*:

**assumes**  $q: P \subseteq \{s. p s = q\} \cap P'$   
— *DynProcProcPar* introduces the same constraint as first conjunction in  $P'$ , so the vcg can simplify it.  
**assumes** *to-prove*:  $\Gamma, \Theta \vdash_F P' (\text{dynCall-exn } f g \text{ init } p \text{ return}' \text{result-exn } c) Q, A$   
**assumes** *ret-nrm-modif*:  $\forall s t. t \in (\text{Modif } (\text{init } s))$   
 $\longrightarrow \text{return}' s t = \text{return } s t$   
**assumes** *ret-abr-modif*:  $\forall s t. t \in (\text{ModifAbr } (\text{init } s))$   
 $\longrightarrow \text{result-exn } (\text{return}' s t) t = \text{result-exn } (\text{return } s t) t$   
**assumes** *modif-clause*:  
 $\forall \sigma. \Gamma, \Theta \vdash_F \{\sigma\} \text{Call } q (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\Gamma, \Theta \vdash_F P (\text{dynCall-exn } f g \text{ init } p \text{ return } \text{result-exn } c) Q, A$   
*<proof>*

**lemma** *ProcProcParModifyReturnSameFaults*:

**assumes**  $q: P \subseteq \{s. p s = q\} \cap P'$   
— *DynProcProcPar* introduces the same constraint as first conjunction in  $P'$ , so the vcg can simplify it.  
**assumes** *to-prove*:  $\Gamma, \Theta \vdash_F P' (\text{dynCall } \text{init } p \text{ return}' c) Q, A$   
**assumes** *ret-nrm-modif*:  $\forall s t. t \in (\text{Modif } (\text{init } s))$   
 $\longrightarrow \text{return}' s t = \text{return } s t$   
**assumes** *ret-abr-modif*:  $\forall s t. t \in (\text{ModifAbr } (\text{init } s))$   
 $\longrightarrow \text{return}' s t = \text{return } s t$   
**assumes** *modif-clause*:

$\forall \sigma. \Gamma, \Theta \vdash /_F \{ \sigma \} \text{ Call } q \text{ (Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\Gamma, \Theta \vdash /_F P \text{ (dynCall init } p \text{ return } c) Q, A$   
 (proof)

**lemma** *Proc-exnProcParModifyReturnNoAbr:*

**assumes**  $q: P \subseteq \{s. p \ s = q\} \cap P'$   
 — *DynProcProcParNoAbrupt* introduces the same constraint as first conjunction in  $P'$ , so the vcg can simplify it.  
**assumes** *to-prove:*  $\Gamma, \Theta \vdash /_F P' \text{ (dynCall-exn } f \ g \ \text{init } p \ \text{return}' \ \text{result-exn } c) Q, A$   
**assumes** *ret-nrm-modif:*  $\forall s \ t. t \in (\text{Modif } (\text{init } s)) \rightarrow \text{return}' \ s \ t = \text{return } s \ t$   
**assumes** *modif-clause:*  
 $\forall \sigma. \Gamma, \Theta \vdash /_{UNIV} \{ \sigma \} (\text{Call } q) (\text{Modif } \sigma), \{ \}$   
**shows**  $\Gamma, \Theta \vdash /_F P \text{ (dynCall-exn } f \ g \ \text{init } p \ \text{return } \text{result-exn } c) Q, A$   
 (proof)

**lemma** *ProcProcParModifyReturnNoAbr:*

**assumes**  $q: P \subseteq \{s. p \ s = q\} \cap P'$   
 — *DynProcProcParNoAbrupt* introduces the same constraint as first conjunction in  $P'$ , so the vcg can simplify it.  
**assumes** *to-prove:*  $\Gamma, \Theta \vdash /_F P' \text{ (dynCall init } p \ \text{return}' \ c) Q, A$   
**assumes** *ret-nrm-modif:*  $\forall s \ t. t \in (\text{Modif } (\text{init } s)) \rightarrow \text{return}' \ s \ t = \text{return } s \ t$   
**assumes** *modif-clause:*  
 $\forall \sigma. \Gamma, \Theta \vdash /_{UNIV} \{ \sigma \} (\text{Call } q) (\text{Modif } \sigma), \{ \}$   
**shows**  $\Gamma, \Theta \vdash /_F P \text{ (dynCall init } p \ \text{return } c) Q, A$   
 (proof)

**lemma** *Proc-exnProcParModifyReturnNoAbrSameFaults:*

**assumes**  $q: P \subseteq \{s. p \ s = q\} \cap P'$   
 — *DynProcProcParNoAbrupt* introduces the same constraint as first conjunction in  $P'$ , so the vcg can simplify it.  
**assumes** *to-prove:*  $\Gamma, \Theta \vdash /_F P' \text{ (dynCall-exn } f \ g \ \text{init } p \ \text{return}' \ \text{result-exn } c) Q, A$   
**assumes** *ret-nrm-modif:*  $\forall s \ t. t \in (\text{Modif } (\text{init } s)) \rightarrow \text{return}' \ s \ t = \text{return } s \ t$   
**assumes** *modif-clause:*  
 $\forall \sigma. \Gamma, \Theta \vdash /_F \{ \sigma \} (\text{Call } q) (\text{Modif } \sigma), \{ \}$   
**shows**  $\Gamma, \Theta \vdash /_F P \text{ (dynCall-exn } f \ g \ \text{init } p \ \text{return } \text{result-exn } c) Q, A$   
 (proof)

**lemma** *ProcProcParModifyReturnNoAbrSameFaults:*

**assumes**  $q: P \subseteq \{s. p \ s = q\} \cap P'$   
 — *DynProcProcParNoAbrupt* introduces the same constraint as first conjunction in  $P'$ , so the vcg can simplify it.  
**assumes** *to-prove:*  $\Gamma, \Theta \vdash /_F P' \text{ (dynCall init } p \ \text{return}' \ c) Q, A$   
**assumes** *ret-nrm-modif:*  $\forall s \ t. t \in (\text{Modif } (\text{init } s)) \rightarrow \text{return}' \ s \ t = \text{return } s \ t$

**assumes** *modif-clause*:

$\forall \sigma. \Gamma, \Theta \vdash_{/F} \{\sigma\} \text{ (Call } q) \text{ (Modif } \sigma), \{\}$

**shows**  $\Gamma, \Theta \vdash_{/F} P \text{ (dynCall init } p \text{ return } c) Q, A$

*<proof>*

**lemma** *MergeGuards-iff*:  $\Gamma, \Theta \vdash_{/F} P \text{ merge-guards } c Q, A = \Gamma, \Theta \vdash_{/F} P c Q, A$

*<proof>*

**lemma** *CombineStrip'*:

**assumes** *deriv*:  $\Gamma, \Theta \vdash_{/F} P c' Q, A$

**assumes** *deriv-strip-triv*:  $\Gamma, \{\} \vdash_{/\{\}} P c'' UNIV, UNIV$

**assumes** *c''*:  $c'' = \text{mark-guards False (strip-guards } (-F) c')$

**assumes** *c*:  $\text{merge-guards } c = \text{merge-guards (mark-guards False } c')$

**shows**  $\Gamma, \Theta \vdash_{/\{\}} P c Q, A$

*<proof>*

**lemma** *CombineStrip''*:

**assumes** *deriv*:  $\Gamma, \Theta \vdash_{/\{\text{True}\}} P c' Q, A$

**assumes** *deriv-strip-triv*:  $\Gamma, \{\} \vdash_{/\{\}} P c'' UNIV, UNIV$

**assumes** *c''*:  $c'' = \text{mark-guards False (strip-guards } (\{\text{False}\}) c')$

**assumes** *c*:  $\text{merge-guards } c = \text{merge-guards (mark-guards False } c')$

**shows**  $\Gamma, \Theta \vdash_{/\{\}} P c Q, A$

*<proof>*

**lemma** *AsmUN*:

$(\bigcup Z. \{(P Z, p, Q Z, A Z)\}) \subseteq \Theta$

$\implies$

$\forall Z. \Gamma, \Theta \vdash_{/F} (P Z) \text{ (Call } p) \text{ (Q Z), (A Z)}$

*<proof>*

**lemma** *augment-context'*:

$\llbracket \Theta \subseteq \Theta'; \forall Z. \Gamma, \Theta \vdash_{/F} (P Z) p (Q Z), (A Z) \rrbracket$

$\implies \forall Z. \Gamma, \Theta \vdash_{/F} (P Z) p (Q Z), (A Z)$

*<proof>*

**lemma** *hoarep-strip*:

$\llbracket \forall Z. \Gamma, \{\} \vdash_{/F} (P Z) p (Q Z), (A Z); F' \subseteq -F \rrbracket \implies$

$\forall Z. \text{strip } F' \Gamma, \{\} \vdash_{/F} (P Z) p (Q Z), (A Z)$

*<proof>*

**lemma** *augment-emptyFaults*:

$\llbracket \forall Z. \Gamma, \{\} \vdash_{/\{\}} (P Z) p (Q Z), (A Z) \rrbracket \implies$

$\forall Z. \Gamma, \{\} \vdash_{/F} (P Z) p (Q Z), (A Z)$

*<proof>*

**lemma** *augment-FaultsUNIV*:

$$\begin{aligned} & \llbracket \forall Z. \Gamma, \{\} \vdash_{/F} (P Z) p (Q Z), (A Z) \rrbracket \implies \\ & \quad \forall Z. \Gamma, \{\} \vdash_{/UNIV} (P Z) p (Q Z), (A Z) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *PostConjI* [*trans*]:

$$\begin{aligned} & \llbracket \Gamma, \Theta \vdash_{/F} P c Q, A; \Gamma, \Theta \vdash_{/F} P c R, B \rrbracket \implies \Gamma, \Theta \vdash_{/F} P c (Q \cap R), (A \cap B) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *PostConjI'* :

$$\begin{aligned} & \llbracket \Gamma, \Theta \vdash_{/F} P c Q, A; \Gamma, \Theta \vdash_{/F} P c Q, A \rrbracket \implies \Gamma, \Theta \vdash_{/F} P c R, B \\ & \implies \Gamma, \Theta \vdash_{/F} P c (Q \cap R), (A \cap B) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *PostConjE* [*consumes 1*]:

$$\begin{aligned} & \text{assumes } \textit{conj}: \Gamma, \Theta \vdash_{/F} P c (Q \cap R), (A \cap B) \\ & \text{assumes } E: \llbracket \Gamma, \Theta \vdash_{/F} P c Q, A; \Gamma, \Theta \vdash_{/F} P c R, B \rrbracket \implies S \\ & \text{shows } S \\ & \langle \text{proof} \rangle \end{aligned}$$

## 6.1 Rules for Single-Step Proof

We are now ready to introduce a set of Hoare rules to be used in single-step structured proofs in Isabelle/Isar.

Assertions of Hoare Logic may be manipulated in calculational proofs, with the inclusion expressed in terms of sets or predicates. Reversed order is supported as well.

**lemma** *annotateI* [*trans*]:

$$\begin{aligned} & \llbracket \Gamma, \Theta \vdash_{/F} P \textit{ anno } Q, A; c = \textit{ anno} \rrbracket \implies \Gamma, \Theta \vdash_{/F} P c Q, A \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *annotate-normI*:

$$\begin{aligned} & \text{assumes } \textit{deriv-anno}: \Gamma, \Theta \vdash_{/F} P \textit{ anno } Q, A \\ & \text{assumes } \textit{norm-eq}: \textit{normalize } c = \textit{normalize anno} \\ & \text{shows } \Gamma, \Theta \vdash_{/F} P c Q, A \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *annotateWhile*:

$$\begin{aligned} & \llbracket \Gamma, \Theta \vdash_{/F} P (\textit{whileAnnoG } gs \ b \ I \ V \ c) \ Q, A \rrbracket \implies \Gamma, \Theta \vdash_{/F} P (\textit{while } gs \ b \ c) \ Q, A \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *reannotateWhile*:

$$\begin{aligned} & \llbracket \Gamma, \Theta \vdash_{/F} P (\textit{whileAnnoG } gs \ b \ I \ V \ c) \ Q, A \rrbracket \implies \Gamma, \Theta \vdash_{/F} P (\textit{whileAnnoG } gs \ b \ J \ V \\ & \ c) \ Q, A \end{aligned}$$

$\langle proof \rangle$

**lemma** *reannotateWhileNoGuard*:

$\llbracket \Gamma, \Theta \vdash /_F P \text{ (whileAnno } b \ I \ V \ c) \ Q, A \rrbracket \implies \Gamma, \Theta \vdash /_F P \text{ (whileAnno } b \ J \ V \ c) \ Q, A$   
 $\langle proof \rangle$

**lemma** *[trans]*:  $P' \subseteq P \implies \Gamma, \Theta \vdash /_F P \ c \ Q, A \implies \Gamma, \Theta \vdash /_F P' \ c \ Q, A$   
 $\langle proof \rangle$

**lemma** *[trans]*:  $Q \subseteq Q' \implies \Gamma, \Theta \vdash /_F P \ c \ Q, A \implies \Gamma, \Theta \vdash /_F P \ c \ Q', A$   
 $\langle proof \rangle$

**lemma** *[trans]*:

$\Gamma, \Theta \vdash /_F \{s. P \ s\} \ c \ Q, A \implies (\bigwedge s. P' \ s \longrightarrow P \ s) \implies \Gamma, \Theta \vdash /_F \{s. P' \ s\} \ c \ Q, A$   
 $\langle proof \rangle$

**lemma** *[trans]*:

$(\bigwedge s. P' \ s \longrightarrow P \ s) \implies \Gamma, \Theta \vdash /_F \{s. P \ s\} \ c \ Q, A \implies \Gamma, \Theta \vdash /_F \{s. P' \ s\} \ c \ Q, A$   
 $\langle proof \rangle$

**lemma** *[trans]*:

$\Gamma, \Theta \vdash /_F P \ c \ \{s. Q \ s\}, A \implies (\bigwedge s. Q \ s \longrightarrow Q' \ s) \implies \Gamma, \Theta \vdash /_F P \ c \ \{s. Q' \ s\}, A$   
 $\langle proof \rangle$

**lemma** *[trans]*:

$(\bigwedge s. Q \ s \longrightarrow Q' \ s) \implies \Gamma, \Theta \vdash /_F P \ c \ \{s. Q \ s\}, A \implies \Gamma, \Theta \vdash /_F P \ c \ \{s. Q' \ s\}, A$   
 $\langle proof \rangle$

**lemma** *[intro?]*:  $\Gamma, \Theta \vdash /_F P \text{ Skip } P, A$

$\langle proof \rangle$

**lemma** *CondInt* *[trans, intro?]*:

$\llbracket \Gamma, \Theta \vdash /_F (P \cap b) \ c1 \ Q, A; \Gamma, \Theta \vdash /_F (P \cap \neg b) \ c2 \ Q, A \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash /_F P \text{ (Cond } b \ c1 \ c2) \ Q, A$   
 $\langle proof \rangle$

**lemma** *CondConj* *[trans, intro?]*:

$\llbracket \Gamma, \Theta \vdash /_F \{s. P \ s \wedge b \ s\} \ c1 \ Q, A; \Gamma, \Theta \vdash /_F \{s. P \ s \wedge \neg b \ s\} \ c2 \ Q, A \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash /_F \{s. P \ s\} \text{ (Cond } \{s. b \ s\} \ c1 \ c2) \ Q, A$   
 $\langle proof \rangle$

**lemma** *WhileInvInt* *[intro?]*:

$\Gamma, \Theta \vdash /_F (P \cap b) \ c \ P, A \implies \Gamma, \Theta \vdash /_F P \text{ (whileAnno } b \ P \ V \ c) \ (P \cap \neg b), A$   
 $\langle proof \rangle$

**lemma** *WhileInt* [*intro?*]:  
 $\Gamma, \Theta \vdash /_F (P \cap b) \ c \ P, A$   
 $\implies$   
 $\Gamma, \Theta \vdash /_F P \ (whileAnno \ b \ \{s. \ undefined\} \ V \ c) \ (P \cap \neg b), A$   
*<proof>*

**lemma** *WhileInvConj* [*intro?*]:  
 $\Gamma, \Theta \vdash /_F \{s. P \ s \wedge b \ s\} \ c \ \{s. P \ s\}, A$   
 $\implies \Gamma, \Theta \vdash /_F \{s. P \ s\} \ (whileAnno \ \{s. b \ s\} \ \{s. P \ s\} \ V \ c) \ \{s. P \ s \wedge \neg b \ s\}, A$   
*<proof>*

**lemma** *WhileConj* [*intro?*]:  
 $\Gamma, \Theta \vdash /_F \{s. P \ s \wedge b \ s\} \ c \ \{s. P \ s\}, A$   
 $\implies$   
 $\Gamma, \Theta \vdash /_F \{s. P \ s\} \ (whileAnno \ \{s. b \ s\} \ \{s. \ undefined\} \ V \ c) \ \{s. P \ s \wedge \neg b \ s\}, A$   
*<proof>*

**end**

## 7 Terminating Programs

**theory** *Termination* **imports** *Semantic* **begin**

### 7.1 Inductive Characterisation: $\Gamma \vdash c \downarrow s$

**inductive** *terminates*::('s,'p,'f) *body*  $\Rightarrow$  ('s,'p,'f) *com*  $\Rightarrow$  ('s,'f) *xstate*  $\Rightarrow$  *bool*  
 (*<+- ↓ ->* [60,20,60] 89)

**for**  $\Gamma$ ::('s,'p,'f) *body*

**where**

*Skip*:  $\Gamma \vdash Skip \downarrow (Normal \ s)$

| *Basic*:  $\Gamma \vdash Basic \ f \downarrow (Normal \ s)$

| *Spec*:  $\Gamma \vdash Spec \ r \downarrow (Normal \ s)$

| *Guard*:  $\llbracket s \in g; \Gamma \vdash c \downarrow (Normal \ s) \rrbracket$

$\implies$

$\Gamma \vdash Guard \ f \ g \ c \downarrow (Normal \ s)$

| *GuardFault*:  $s \notin g$

$\implies$

$\Gamma \vdash Guard \ f \ g \ c \downarrow (Normal \ s)$

| *Fault* [*intro,simp*]:  $\Gamma \vdash c \downarrow Fault \ f$

- | *Seq*:  $\llbracket \Gamma \vdash c_1 \downarrow \text{Normal } s; \forall s'. \Gamma \vdash \langle c_1, \text{Normal } s \rangle \Rightarrow s' \longrightarrow \Gamma \vdash c_2 \downarrow s' \rrbracket$   
 $\implies$   
 $\Gamma \vdash \text{Seq } c_1 \ c_2 \downarrow (\text{Normal } s)$
- | *CondTrue*:  $\llbracket s \in b; \Gamma \vdash c_1 \downarrow (\text{Normal } s) \rrbracket$   
 $\implies$   
 $\Gamma \vdash \text{Cond } b \ c_1 \ c_2 \downarrow (\text{Normal } s)$
- | *CondFalse*:  $\llbracket s \notin b; \Gamma \vdash c_2 \downarrow (\text{Normal } s) \rrbracket$   
 $\implies$   
 $\Gamma \vdash \text{Cond } b \ c_1 \ c_2 \downarrow (\text{Normal } s)$
- | *WhileTrue*:  $\llbracket s \in b; \Gamma \vdash c \downarrow (\text{Normal } s);$   
 $\forall s'. \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow s' \longrightarrow \Gamma \vdash \text{While } b \ c \downarrow s' \rrbracket$   
 $\implies$   
 $\Gamma \vdash \text{While } b \ c \downarrow (\text{Normal } s)$
- | *WhileFalse*:  $\llbracket s \notin b \rrbracket$   
 $\implies$   
 $\Gamma \vdash \text{While } b \ c \downarrow (\text{Normal } s)$
- | *Call*:  $\llbracket \Gamma \ p = \text{Some } \text{bdy}; \Gamma \vdash \text{bdy} \downarrow (\text{Normal } s) \rrbracket$   
 $\implies$   
 $\Gamma \vdash \text{Call } p \downarrow (\text{Normal } s)$
- | *CallUndefined*:  $\llbracket \Gamma \ p = \text{None} \rrbracket$   
 $\implies$   
 $\Gamma \vdash \text{Call } p \downarrow (\text{Normal } s)$
- | *Stuck* [*intro, simp*]:  $\Gamma \vdash c \downarrow \text{Stuck}$
- | *DynCom*:  $\llbracket \Gamma \vdash (c \ s) \downarrow (\text{Normal } s) \rrbracket$   
 $\implies$   
 $\Gamma \vdash \text{DynCom } c \downarrow (\text{Normal } s)$
- | *Throw*:  $\Gamma \vdash \text{Throw} \downarrow (\text{Normal } s)$
- | *Abrupt* [*intro, simp*]:  $\Gamma \vdash c \downarrow \text{Abrupt } s$
- | *Catch*:  $\llbracket \Gamma \vdash c_1 \downarrow \text{Normal } s;$   
 $\forall s'. \Gamma \vdash \langle c_1, \text{Normal } s \rangle \Rightarrow \text{Abrupt } s' \longrightarrow \Gamma \vdash c_2 \downarrow \text{Normal } s' \rrbracket$   
 $\implies$   
 $\Gamma \vdash \text{Catch } c_1 \ c_2 \downarrow \text{Normal } s$

**inductive-cases** *terminates-elim-cases* [*cases set*]:

$\Gamma \vdash \text{Skip} \downarrow s$   
 $\Gamma \vdash \text{Guard } f \ g \ c \downarrow s$   
 $\Gamma \vdash \text{Basic } f \downarrow s$   
 $\Gamma \vdash \text{Spec } r \downarrow s$   
 $\Gamma \vdash \text{Seq } c1 \ c2 \downarrow s$   
 $\Gamma \vdash \text{Cond } b \ c1 \ c2 \downarrow s$   
 $\Gamma \vdash \text{While } b \ c \downarrow s$   
 $\Gamma \vdash \text{Call } p \downarrow s$   
 $\Gamma \vdash \text{DynCom } c \downarrow s$   
 $\Gamma \vdash \text{Throw} \downarrow s$   
 $\Gamma \vdash \text{Catch } c1 \ c2 \downarrow s$

**inductive-cases** *terminates-Normal-elim-cases* [*cases set*]:

$\Gamma \vdash \text{Skip} \downarrow \text{Normal } s$   
 $\Gamma \vdash \text{Guard } f \ g \ c \downarrow \text{Normal } s$   
 $\Gamma \vdash \text{Basic } f \downarrow \text{Normal } s$   
 $\Gamma \vdash \text{Spec } r \downarrow \text{Normal } s$   
 $\Gamma \vdash \text{Seq } c1 \ c2 \downarrow \text{Normal } s$   
 $\Gamma \vdash \text{Cond } b \ c1 \ c2 \downarrow \text{Normal } s$   
 $\Gamma \vdash \text{While } b \ c \downarrow \text{Normal } s$   
 $\Gamma \vdash \text{Call } p \downarrow \text{Normal } s$   
 $\Gamma \vdash \text{DynCom } c \downarrow \text{Normal } s$   
 $\Gamma \vdash \text{Throw} \downarrow \text{Normal } s$   
 $\Gamma \vdash \text{Catch } c1 \ c2 \downarrow \text{Normal } s$

**lemma** *terminates-Skip'*:  $\Gamma \vdash \text{Skip} \downarrow s$   
 ⟨*proof*⟩

**lemma** *terminates-Call-body*:  
 $\Gamma \ p = \text{Some } \text{bdy} \implies \Gamma \vdash \text{Call } p \downarrow s = \Gamma \vdash (\text{the } (\Gamma \ p)) \downarrow s$   
 ⟨*proof*⟩

**lemma** *terminates-Normal-Call-body*:  
 $p \in \text{dom } \Gamma \implies$   
 $\Gamma \vdash \text{Call } p \downarrow \text{Normal } s = \Gamma \vdash (\text{the } (\Gamma \ p)) \downarrow \text{Normal } s$   
 ⟨*proof*⟩

**lemma** *terminates-implies-exec*:  
**assumes** *terminates*:  $\Gamma \vdash c \downarrow s$   
**shows**  $\exists t. \Gamma \vdash \langle c, s \rangle \Rightarrow t$   
 ⟨*proof*⟩

**lemma** *terminates-block-exn*:  
 $\llbracket \Gamma \vdash \text{bdy} \downarrow \text{Normal } (\text{init } s);$   
 $\forall t. \Gamma \vdash \langle \text{bdy}, \text{Normal } (\text{init } s) \rangle \Rightarrow \text{Normal } t \longrightarrow \Gamma \vdash c \ s \ t \downarrow \text{Normal } (\text{return } s \ t) \rrbracket$   
 $\implies \Gamma \vdash \text{block-exn } \text{init } \text{bdy } \text{return } \text{result-exn } c \downarrow \text{Normal } s$   
 ⟨*proof*⟩

**lemma** *terminates-block*:

$$\begin{aligned} & \llbracket \Gamma \vdash \text{bdy} \downarrow \text{Normal} (\text{init } s); \\ & \quad \forall t. \Gamma \vdash \langle \text{bdy}, \text{Normal} (\text{init } s) \rangle \Rightarrow \text{Normal } t \longrightarrow \Gamma \vdash c \text{ s } t \downarrow \text{Normal} (\text{return } s) t \rrbracket \\ & \implies \Gamma \vdash \text{block init bdy return } c \downarrow \text{Normal } s \\ & \quad \langle \text{proof} \rangle \end{aligned}$$

**lemma** *terminates-block-exn-elim* [cases set, consumes 1]:

**assumes** *termi*:  $\Gamma \vdash \text{block-exn init bdy return result-exn } c \downarrow \text{Normal } s$

**assumes** *e*:  $\llbracket \Gamma \vdash \text{bdy} \downarrow \text{Normal} (\text{init } s);$

$\quad \forall t. \Gamma \vdash \langle \text{bdy}, \text{Normal} (\text{init } s) \rangle \Rightarrow \text{Normal } t \longrightarrow \Gamma \vdash c \text{ s } t \downarrow \text{Normal} (\text{return } s$   
 $t)$

$\rrbracket \implies P$

**shows** *P*

$\langle \text{proof} \rangle$

**lemma** *terminates-block-elim* [cases set, consumes 1]:

**assumes** *termi*:  $\Gamma \vdash \text{block init bdy return } c \downarrow \text{Normal } s$

**assumes** *e*:  $\llbracket \Gamma \vdash \text{bdy} \downarrow \text{Normal} (\text{init } s);$

$\quad \forall t. \Gamma \vdash \langle \text{bdy}, \text{Normal} (\text{init } s) \rangle \Rightarrow \text{Normal } t \longrightarrow \Gamma \vdash c \text{ s } t \downarrow \text{Normal} (\text{return } s$   
 $t)$

$\rrbracket \implies P$

**shows** *P*

$\langle \text{proof} \rangle$

**lemma** *terminates-call*:

$\llbracket \Gamma \text{ p} = \text{Some bdy}; \Gamma \vdash \text{bdy} \downarrow \text{Normal} (\text{init } s);$

$\quad \forall t. \Gamma \vdash \langle \text{bdy}, \text{Normal} (\text{init } s) \rangle \Rightarrow \text{Normal } t \longrightarrow \Gamma \vdash c \text{ s } t \downarrow \text{Normal} (\text{return } s) t \rrbracket$

$\implies \Gamma \vdash \text{call init p return } c \downarrow \text{Normal } s$

$\langle \text{proof} \rangle$

**lemma** *terminates-call-exn*:

$\llbracket \Gamma \text{ p} = \text{Some bdy}; \Gamma \vdash \text{bdy} \downarrow \text{Normal} (\text{init } s);$

$\quad \forall t. \Gamma \vdash \langle \text{bdy}, \text{Normal} (\text{init } s) \rangle \Rightarrow \text{Normal } t \longrightarrow \Gamma \vdash c \text{ s } t \downarrow \text{Normal} (\text{return } s) t \rrbracket$

$\implies \Gamma \vdash \text{call-exn init p return result-exn } c \downarrow \text{Normal } s$

$\langle \text{proof} \rangle$

**lemma** *terminates-callUndefined*:

$\llbracket \Gamma \text{ p} = \text{None} \rrbracket$

$\implies \Gamma \vdash \text{call init p return result} \downarrow \text{Normal } s$

$\langle \text{proof} \rangle$

**lemma** *terminates-call-exnUndefined*:

$\llbracket \Gamma \text{ p} = \text{None} \rrbracket$

$\implies \Gamma \vdash \text{call-exn init p return result-exn result} \downarrow \text{Normal } s$

$\langle \text{proof} \rangle$

**lemma** *terminates-call-exn-elim* [cases set, consumes 1]:

**assumes** *termi*:  $\Gamma \vdash \text{call-exn init p return result-exn } c \downarrow \text{Normal } s$

**assumes** *bdy*:  $\bigwedge \text{bdy}. \llbracket \Gamma \text{ p} = \text{Some bdy}; \Gamma \vdash \text{bdy} \downarrow \text{Normal} (\text{init } s);$

$\forall t. \Gamma \vdash \langle bdy, Normal (init\ s) \rangle \Rightarrow Normal\ t \longrightarrow \Gamma \vdash c\ s\ t \downarrow Normal (return\ s\ t)$   
 $\implies P$   
**assumes** *undef*:  $\llbracket \Gamma\ p = None \rrbracket \implies P$   
**shows** *P*  
 $\langle proof \rangle$

**lemma** *terminates-call-elim* [*cases set, consumes 1*]:  
**assumes** *termi*:  $\Gamma \vdash call\ init\ p\ return\ c \downarrow Normal\ s$   
**assumes** *bdy*:  $\bigwedge bdy. \llbracket \Gamma\ p = Some\ bdy; \Gamma \vdash bdy \downarrow Normal (init\ s);$   
 $\forall t. \Gamma \vdash \langle bdy, Normal (init\ s) \rangle \Rightarrow Normal\ t \longrightarrow \Gamma \vdash c\ s\ t \downarrow Normal (return\ s\ t)$   
 $\implies P$   
**assumes** *undef*:  $\llbracket \Gamma\ p = None \rrbracket \implies P$   
**shows** *P*  
 $\langle proof \rangle$

**lemma** *terminates-dynCall*:  
 $\llbracket \Gamma \vdash call\ init\ (p\ s)\ return\ c \downarrow Normal\ s \rrbracket$   
 $\implies \Gamma \vdash dynCall\ init\ p\ return\ c \downarrow Normal\ s$   
 $\langle proof \rangle$

**lemma** *terminates-guards*:  $\Gamma \vdash c \downarrow Normal\ s \implies \Gamma \vdash guards\ gs\ c \downarrow Normal\ s$   
 $\langle proof \rangle$

**lemma** *terminates-guards-Fault*:  $find\ (\lambda(f, g). s \notin g)\ gs = Some\ (f, g) \implies \Gamma \vdash guards\ gs\ c \downarrow Normal\ s$   
 $\langle proof \rangle$

**lemma** *terminates-maybe-guard-Fault*:  $s \notin g \implies \Gamma \vdash maybe-guard\ f\ g\ c \downarrow Normal\ s$   
 $\langle proof \rangle$

**lemma** *terminates-guards-DynCom*:  $\Gamma \vdash (c\ s) \downarrow Normal\ s \implies \Gamma \vdash guards\ gs\ (DynCom\ c) \downarrow Normal\ s$   
 $\langle proof \rangle$

**lemma** *terminates-maybe-guard-DynCom*:  $\Gamma \vdash (c\ s) \downarrow Normal\ s \implies \Gamma \vdash maybe-guard\ f\ g\ (DynCom\ c) \downarrow Normal\ s$   
 $\langle proof \rangle$

**lemma** *terminates-dynCall-exn*:  
 $\llbracket \Gamma \vdash call-exn\ init\ (p\ s)\ return\ result-exn\ c \downarrow Normal\ s \rrbracket$   
 $\implies \Gamma \vdash dynCall-exn\ f\ g\ init\ p\ return\ result-exn\ c \downarrow Normal\ s$   
 $\langle proof \rangle$

**lemma** *terminates-dynCall-elim* [*cases set, consumes 1*]:  
**assumes** *termi*:  $\Gamma \vdash dynCall\ init\ p\ return\ c \downarrow Normal\ s$   
**assumes**  $\llbracket \Gamma \vdash call\ init\ (p\ s)\ return\ c \downarrow Normal\ s \rrbracket \implies P$   
**shows** *P*

$\langle proof \rangle$

**lemma** *terminates-guards-elim* [*cases set, consumes 1, case-names noFault someFault*]:

**assumes** *termi*:  $\Gamma \vdash \text{guards } gs \ c \downarrow \text{Normal } s$   
**assumes** *noFault*:  $\llbracket \forall f \ g. (f, g) \in \text{set } gs \longrightarrow s \in g; \Gamma \vdash c \downarrow \text{Normal } s \rrbracket \Longrightarrow P$   
**assumes** *someFault*:  $\bigwedge f \ g. \text{find } (\lambda(f, g). s \notin g) \ gs = \text{Some } (f, g) \Longrightarrow P$   
**shows**  $P$   
 $\langle proof \rangle$

**lemma** *terminates-maybe-guard-elim* [*cases set, consumes 1, case-names noFault someFault*]:

**assumes** *termi*:  $\Gamma \vdash \text{maybe-guard } f \ g \ c \downarrow \text{Normal } s$   
**assumes** *noFault*:  $\llbracket s \in g; \Gamma \vdash c \downarrow \text{Normal } s \rrbracket \Longrightarrow P$   
**assumes** *someFault*:  $s \notin g \Longrightarrow P$   
**shows**  $P$   
 $\langle proof \rangle$

**lemma** *terminates-dynCall-exn-elim* [*cases set, consumes 1, case-names noFault someFault*]:

**assumes** *termi*:  $\Gamma \vdash \text{dynCall-exn } f \ g \ \text{init } p \ \text{return } \text{result-exn } c \downarrow \text{Normal } s$   
**assumes** *noFault*:  $\llbracket s \in g; \Gamma \vdash \text{call-exn } \text{init } (p \ s) \ \text{return } \text{result-exn } c \downarrow \text{Normal } s \rrbracket \Longrightarrow P$   
**assumes** *someFault*:  $s \notin g \Longrightarrow P$   
**shows**  $P$   
 $\langle proof \rangle$

## 7.2 Lemmas about *sequence, flatten and Language.normalize*

**lemma** *terminates-sequence-app*:

$\bigwedge s. \llbracket \Gamma \vdash \text{sequence } \text{Seq } xs \downarrow \text{Normal } s; \forall s'. \Gamma \vdash \langle \text{sequence } \text{Seq } xs, \text{Normal } s \rangle \Rightarrow s' \longrightarrow \Gamma \vdash \text{sequence } \text{Seq } ys \downarrow s' \rrbracket$   
 $\Longrightarrow \Gamma \vdash \text{sequence } \text{Seq } (xs \ @ \ ys) \downarrow \text{Normal } s$   
 $\langle proof \rangle$

**lemma** *terminates-sequence-appD*:

$\bigwedge s. \Gamma \vdash \text{sequence } \text{Seq } (xs \ @ \ ys) \downarrow \text{Normal } s$   
 $\Longrightarrow \Gamma \vdash \text{sequence } \text{Seq } xs \downarrow \text{Normal } s \wedge$   
 $(\forall s'. \Gamma \vdash \langle \text{sequence } \text{Seq } xs, \text{Normal } s \rangle \Rightarrow s' \longrightarrow \Gamma \vdash \text{sequence } \text{Seq } ys \downarrow s')$   
 $\langle proof \rangle$

**lemma** *terminates-sequence-appE* [*consumes 1*]:

$\llbracket \Gamma \vdash \text{sequence } \text{Seq } (xs \ @ \ ys) \downarrow \text{Normal } s; \llbracket \Gamma \vdash \text{sequence } \text{Seq } xs \downarrow \text{Normal } s; \forall s'. \Gamma \vdash \langle \text{sequence } \text{Seq } xs, \text{Normal } s \rangle \Rightarrow s' \longrightarrow \Gamma \vdash \text{sequence } \text{Seq } ys \downarrow s' \rrbracket \Longrightarrow P$   
 $\Longrightarrow P$   
 $\langle proof \rangle$

**lemma** *terminates-to-terminates-sequence-flatten*:

**assumes** *termi*:  $\Gamma \vdash c \downarrow s$   
**shows**  $\Gamma \vdash \text{sequence Seq (flatten } c) \downarrow s$   
 $\langle \text{proof} \rangle$

**lemma** *terminates-to-terminates-normalize*:

**assumes** *termi*:  $\Gamma \vdash c \downarrow s$   
**shows**  $\Gamma \vdash \text{normalize } c \downarrow s$   
 $\langle \text{proof} \rangle$

**lemma** *terminates-sequence-flatten-to-terminates*:

**shows**  $\bigwedge s. \Gamma \vdash \text{sequence Seq (flatten } c) \downarrow s \implies \Gamma \vdash c \downarrow s$   
 $\langle \text{proof} \rangle$

**lemma** *terminates-normalize-to-terminates*:

**shows**  $\bigwedge s. \Gamma \vdash \text{normalize } c \downarrow s \implies \Gamma \vdash c \downarrow s$   
 $\langle \text{proof} \rangle$

**lemma** *terminates-iff-terminates-normalize*:

$\Gamma \vdash \text{normalize } c \downarrow s = \Gamma \vdash c \downarrow s$   
 $\langle \text{proof} \rangle$

### 7.3 Lemmas about *strip-guards*

**lemma** *terminates-strip-guards-to-terminates*:  $\bigwedge s. \Gamma \vdash \text{strip-guards } F \downarrow c \downarrow s \implies \Gamma \vdash c \downarrow s$

$\langle \text{proof} \rangle$

**lemma** *terminates-strip-to-terminates*:

**assumes** *termi-strip*:  $\text{strip } F \downarrow c \downarrow s$   
**shows**  $\Gamma \vdash c \downarrow s$   
 $\langle \text{proof} \rangle$

### 7.4 Lemmas about $c_1 \cap_g c_2$

**lemma** *inter-guards-terminates*:

$\bigwedge c \ c_2 \ s. \llbracket (c_1 \cap_g c_2) = \text{Some } c; \Gamma \vdash c_1 \downarrow s \rrbracket$   
 $\implies \Gamma \vdash c \downarrow s$   
 $\langle \text{proof} \rangle$

**lemma** *inter-guards-terminates'*:

**assumes**  $c: (c_1 \cap_g c_2) = \text{Some } c$   
**assumes** *termi-c2*:  $\Gamma \vdash c_2 \downarrow s$   
**shows**  $\Gamma \vdash c \downarrow s$   
 $\langle \text{proof} \rangle$

### 7.5 Lemmas about *mark-guards*

**lemma** *terminates-to-terminates-mark-guards*:

**assumes** *termi*:  $\Gamma \vdash c \downarrow s$   
**shows**  $\Gamma \vdash \text{mark-guards } f \downarrow c \downarrow s$   
 $\langle \text{proof} \rangle$

**lemma** *terminates-mark-guards-to-terminates-Normal*:  
 $\bigwedge s. \Gamma \vdash \text{mark-guards } f \ c \downarrow \text{Normal } s \implies \Gamma \vdash c \downarrow \text{Normal } s$   
 ⟨proof⟩

**lemma** *terminates-mark-guards-to-terminates*:  
 $\Gamma \vdash \text{mark-guards } f \ c \downarrow s \implies \Gamma \vdash c \downarrow s$   
 ⟨proof⟩

## 7.6 Lemmas about *merge-guards*

**lemma** *terminates-to-terminates-merge-guards*:  
 assumes *termi*:  $\Gamma \vdash c \downarrow s$   
 shows  $\Gamma \vdash \text{merge-guards } c \downarrow s$   
 ⟨proof⟩

**lemma** *terminates-merge-guards-to-terminates-Normal*:  
 shows  $\bigwedge s. \Gamma \vdash \text{merge-guards } c \downarrow \text{Normal } s \implies \Gamma \vdash c \downarrow \text{Normal } s$   
 ⟨proof⟩

**lemma** *terminates-merge-guards-to-terminates*:  
 $\Gamma \vdash \text{merge-guards } c \downarrow s \implies \Gamma \vdash c \downarrow s$   
 ⟨proof⟩

**theorem** *terminates-iff-terminates-merge-guards*:  
 $\Gamma \vdash c \downarrow s = \Gamma \vdash \text{merge-guards } c \downarrow s$   
 ⟨proof⟩

## 7.7 Lemmas about $c_1 \subseteq_g c_2$

**lemma** *terminates-fewer-guards-Normal*:  
 shows  $\bigwedge c \ s. \llbracket \Gamma \vdash c' \downarrow \text{Normal } s; c \subseteq_g c'; \Gamma \vdash \langle c', \text{Normal } s \rangle \Rightarrow \notin \text{Fault ' UNIV} \rrbracket$   
 $\implies \Gamma \vdash c \downarrow \text{Normal } s$   
 ⟨proof⟩

**theorem** *terminates-fewer-guards*:  
 shows  $\llbracket \Gamma \vdash c' \downarrow s; c \subseteq_g c'; \Gamma \vdash \langle c', s \rangle \Rightarrow \notin \text{Fault ' UNIV} \rrbracket$   
 $\implies \Gamma \vdash c \downarrow s$   
 ⟨proof⟩

**lemma** *terminates-noFault-strip-guards*:  
 assumes *termi*:  $\Gamma \vdash c \downarrow \text{Normal } s$   
 shows  $\llbracket \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \notin \text{Fault ' F} \rrbracket \implies \Gamma \vdash \text{strip-guards } F \ c \downarrow \text{Normal } s$   
 ⟨proof⟩

## 7.8 Lemmas about *strip-guards*

**lemma** *terminates-noFault-strip*:  
 assumes *termi*:  $\Gamma \vdash c \downarrow \text{Normal } s$   
 shows  $\llbracket \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \notin \text{Fault ' F} \rrbracket \implies \text{strip } F \ \Gamma \vdash c \downarrow \text{Normal } s$

$\langle proof \rangle$

## 7.9 Miscellaneous

**lemma** *terminates-while-lemma*:

**assumes** *termi*:  $\Gamma \vdash w \downarrow fk$

**shows**  $\bigwedge k b c. \llbracket fk = Normal (f k); w = While b c; \forall i. \Gamma \vdash \langle c, Normal (f i) \rangle \Rightarrow Normal (f (Suc i)) \rrbracket$   
 $\implies \exists i. f i \notin b$

$\langle proof \rangle$

**lemma** *terminates-while*:

$\llbracket \Gamma \vdash (While b c) \downarrow Normal (f k); \forall i. \Gamma \vdash \langle c, Normal (f i) \rangle \Rightarrow Normal (f (Suc i)) \rrbracket$   
 $\implies \exists i. f i \notin b$

$\langle proof \rangle$

**lemma** *wf-terminates-while*:

*wf*  $\{(t, s). \Gamma \vdash (While b c) \downarrow Normal s \wedge s \in b \wedge \Gamma \vdash \langle c, Normal s \rangle \Rightarrow Normal t\}$

$\langle proof \rangle$

**lemma** *terminates-restrict-to-terminates*:

**assumes** *terminates-res*:  $\Gamma \upharpoonright_M \vdash c \downarrow s$

**assumes** *not-Stuck*:  $\Gamma \upharpoonright_M \vdash \langle c, s \rangle \Rightarrow \notin \{Stuck\}$

**shows**  $\Gamma \vdash c \downarrow s$

$\langle proof \rangle$

**end**

## 8 Small-Step Semantics and Infinite Computations

**theory** *SmallStep* **imports** *Termination*

**begin**

The redex of a statement is the substatement, which is actually altered by the next step in the small-step semantics.

**primrec** *redex*::  $(\prime s, \prime p, \prime f) com \Rightarrow (\prime s, \prime p, \prime f) com$

**where**

*redex* *Skip* = *Skip* |  
*redex* (*Basic* *f*) = (*Basic* *f*) |  
*redex* (*Spec* *r*) = (*Spec* *r*) |  
*redex* (*Seq* *c*<sub>1</sub> *c*<sub>2</sub>) = *redex* *c*<sub>1</sub> |  
*redex* (*Cond* *b* *c*<sub>1</sub> *c*<sub>2</sub>) = (*Cond* *b* *c*<sub>1</sub> *c*<sub>2</sub>) |  
*redex* (*While* *b* *c*) = (*While* *b* *c*) |  
*redex* (*Call* *p*) = (*Call* *p*) |  
*redex* (*DynCom* *d*) = (*DynCom* *d*) |  
*redex* (*Guard* *f* *b* *c*) = (*Guard* *f* *b* *c*) |  
*redex* (*Throw*) = *Throw* |

$redex (Catch\ c_1\ c_2) = redex\ c_1$

## 8.1 Small-Step Computation: $\Gamma \vdash (c, s) \rightarrow (c', s')$

**type-synonym**  $(s, p, f)\ config = (s, p, f)\ com \times (s, f)\ xstate$

**inductive**  $step::[(s, p, f)\ body, (s, p, f)\ config, (s, p, f)\ config] \Rightarrow bool$   
 $(\langle \vdash (- \rightarrow / -) \rangle [81, 81, 81] 100)$

**for**  $\Gamma::(s, p, f)\ body$

**where**

*Basic*:  $\Gamma \vdash (Basic\ f, Normal\ s) \rightarrow (Skip, Normal\ (f\ s))$

| *Spec*:  $(s, t) \in r \Longrightarrow \Gamma \vdash (Spec\ r, Normal\ s) \rightarrow (Skip, Normal\ t)$

| *SpecStuck*:  $\forall t. (s, t) \notin r \Longrightarrow \Gamma \vdash (Spec\ r, Normal\ s) \rightarrow (Skip, Stuck)$

| *Guard*:  $s \in g \Longrightarrow \Gamma \vdash (Guard\ f\ g\ c, Normal\ s) \rightarrow (c, Normal\ s)$

| *GuardFault*:  $s \notin g \Longrightarrow \Gamma \vdash (Guard\ f\ g\ c, Normal\ s) \rightarrow (Skip, Fault\ f)$

| *Seq*:  $\Gamma \vdash (c_1, s) \rightarrow (c_1', s')$

$\Longrightarrow$

$\Gamma \vdash (Seq\ c_1\ c_2, s) \rightarrow (Seq\ c_1'\ c_2, s')$

| *SeqSkip*:  $\Gamma \vdash (Seq\ Skip\ c_2, s) \rightarrow (c_2, s)$

| *SeqThrow*:  $\Gamma \vdash (Seq\ Throw\ c_2, Normal\ s) \rightarrow (Throw, Normal\ s)$

| *CondTrue*:  $s \in b \Longrightarrow \Gamma \vdash (Cond\ b\ c_1\ c_2, Normal\ s) \rightarrow (c_1, Normal\ s)$

| *CondFalse*:  $s \notin b \Longrightarrow \Gamma \vdash (Cond\ b\ c_1\ c_2, Normal\ s) \rightarrow (c_2, Normal\ s)$

| *WhileTrue*:  $\llbracket s \in b \rrbracket$

$\Longrightarrow$

$\Gamma \vdash (While\ b\ c, Normal\ s) \rightarrow (Seq\ c\ (While\ b\ c), Normal\ s)$

| *WhileFalse*:  $\llbracket s \notin b \rrbracket$

$\Longrightarrow$

$\Gamma \vdash (While\ b\ c, Normal\ s) \rightarrow (Skip, Normal\ s)$

| *Call*:  $\Gamma\ p = Some\ bdy \Longrightarrow$

$\Gamma \vdash (Call\ p, Normal\ s) \rightarrow (bdy, Normal\ s)$

| *CallUndefined*:  $\Gamma\ p = None \Longrightarrow$

$\Gamma \vdash (Call\ p, Normal\ s) \rightarrow (Skip, Stuck)$

| *DynCom*:  $\Gamma \vdash (DynCom\ c, Normal\ s) \rightarrow (c\ s, Normal\ s)$

| *Catch*:  $\llbracket \Gamma \vdash (c_1, s) \rightarrow (c_1', s') \rrbracket$

$\Longrightarrow$

$\Gamma \vdash (Catch\ c_1\ c_2, s) \rightarrow (Catch\ c_1'\ c_2, s')$

$\mid$  *CatchThrow*:  $\Gamma \vdash (\text{Catch Throw } c_2, \text{Normal } s) \rightarrow (c_2, \text{Normal } s)$   
 $\mid$  *CatchSkip*:  $\Gamma \vdash (\text{Catch Skip } c_2, s) \rightarrow (\text{Skip}, s)$   
 $\mid$  *FaultProp*:  $\llbracket c \neq \text{Skip}; \text{redex } c = c \rrbracket \implies \Gamma \vdash (c, \text{Fault } f) \rightarrow (\text{Skip}, \text{Fault } f)$   
 $\mid$  *StuckProp*:  $\llbracket c \neq \text{Skip}; \text{redex } c = c \rrbracket \implies \Gamma \vdash (c, \text{Stuck}) \rightarrow (\text{Skip}, \text{Stuck})$   
 $\mid$  *AbruptProp*:  $\llbracket c \neq \text{Skip}; \text{redex } c = c \rrbracket \implies \Gamma \vdash (c, \text{Abrupt } f) \rightarrow (\text{Skip}, \text{Abrupt } f)$

**lemmas** *step-induct* = *step.induct* [*of* - (*c, s*) (*c', s'*), *split-format* (*complete*), *case-names*  
*Basic Spec SpecStuck Guard GuardFault Seq SeqSkip SeqThrow CondTrue Cond-*  
*False*  
*WhileTrue WhileFalse Call CallUndefined DynCom Catch CatchThrow CatchSkip*  
*FaultProp StuckProp AbruptProp, induct set]*

**inductive-cases** *step-elim-cases* [*cases set*]:

$\Gamma \vdash (\text{Skip}, s) \rightarrow u$   
 $\Gamma \vdash (\text{Guard } f g c, s) \rightarrow u$   
 $\Gamma \vdash (\text{Basic } f, s) \rightarrow u$   
 $\Gamma \vdash (\text{Spec } r, s) \rightarrow u$   
 $\Gamma \vdash (\text{Seq } c1 c2, s) \rightarrow u$   
 $\Gamma \vdash (\text{Cond } b c1 c2, s) \rightarrow u$   
 $\Gamma \vdash (\text{While } b c, s) \rightarrow u$   
 $\Gamma \vdash (\text{Call } p, s) \rightarrow u$   
 $\Gamma \vdash (\text{DynCom } c, s) \rightarrow u$   
 $\Gamma \vdash (\text{Throw}, s) \rightarrow u$   
 $\Gamma \vdash (\text{Catch } c1 c2, s) \rightarrow u$

**inductive-cases** *step-Normal-elim-cases* [*cases set*]:

$\Gamma \vdash (\text{Skip}, \text{Normal } s) \rightarrow u$   
 $\Gamma \vdash (\text{Guard } f g c, \text{Normal } s) \rightarrow u$   
 $\Gamma \vdash (\text{Basic } f, \text{Normal } s) \rightarrow u$   
 $\Gamma \vdash (\text{Spec } r, \text{Normal } s) \rightarrow u$   
 $\Gamma \vdash (\text{Seq } c1 c2, \text{Normal } s) \rightarrow u$   
 $\Gamma \vdash (\text{Cond } b c1 c2, \text{Normal } s) \rightarrow u$   
 $\Gamma \vdash (\text{While } b c, \text{Normal } s) \rightarrow u$   
 $\Gamma \vdash (\text{Call } p, \text{Normal } s) \rightarrow u$   
 $\Gamma \vdash (\text{DynCom } c, \text{Normal } s) \rightarrow u$   
 $\Gamma \vdash (\text{Throw}, \text{Normal } s) \rightarrow u$   
 $\Gamma \vdash (\text{Catch } c1 c2, \text{Normal } s) \rightarrow u$

The final configuration is either of the form (*Skip*, -) for normal termination, or (*Throw*, *Normal s*) in case the program was started in a *Normal* state and terminated abruptly. The *Abrupt* state is not used to model abrupt termination, in contrast to the big-step semantics. Only if the program starts in an *Abrupt* states it ends in the same *Abrupt* state.

**definition** *final*:: (*'s, 'p, 'f*) *config*  $\Rightarrow$  *bool* **where**  
*final cfg* = (*fst cfg* = *Skip*  $\vee$  (*fst cfg* = *Throw*  $\wedge$  ( $\exists s. \text{snd } \text{cfg} = \text{Normal } s$ )))

**abbreviation**

$$\text{step-rtrancl} :: [(\text{'s','p','f'}) \text{ body}, (\text{'s','p','f'}) \text{ config}, (\text{'s','p','f'}) \text{ config}] \Rightarrow \text{bool}$$

$$\langle \text{!+} (- \rightarrow^* / -) \rangle [81,81,81] 100$$
**where**

$$\Gamma \vdash \text{cf0} \rightarrow^* \text{cf1} \equiv (\text{CONST step } \Gamma)^{**} \text{cf0 cf1}$$
**abbreviation**

$$\text{step-trancl} :: [(\text{'s','p','f'}) \text{ body}, (\text{'s','p','f'}) \text{ config}, (\text{'s','p','f'}) \text{ config}] \Rightarrow \text{bool}$$

$$\langle \text{!+} (- \rightarrow^+ / -) \rangle [81,81,81] 100$$
**where**

$$\Gamma \vdash \text{cf0} \rightarrow^+ \text{cf1} \equiv (\text{CONST step } \Gamma)^{++} \text{cf0 cf1}$$

## 8.2 Structural Properties of Small Step Computations

**lemma** *redex-not-Seq*:  $\text{redex } c = \text{Seq } c1 \ c2 \Longrightarrow P$

*<proof>*

**lemma** *no-step-final*:

**assumes** *step*:  $\Gamma \vdash (c, s) \rightarrow (c', s')$

**shows** *final*  $(c, s) \Longrightarrow P$

*<proof>*

**lemma** *no-step-final'*:

**assumes** *step*:  $\Gamma \vdash \text{cfg} \rightarrow \text{cfg}'$

**shows** *final*  $\text{cfg} \Longrightarrow P$

*<proof>*

**lemma** *step-Abrupt*:

**assumes** *step*:  $\Gamma \vdash (c, s) \rightarrow (c', s')$

**shows**  $\bigwedge x. s = \text{Abrupt } x \Longrightarrow s' = \text{Abrupt } x$

*<proof>*

**lemma** *step-Fault*:

**assumes** *step*:  $\Gamma \vdash (c, s) \rightarrow (c', s')$

**shows**  $\bigwedge f. s = \text{Fault } f \Longrightarrow s' = \text{Fault } f$

*<proof>*

**lemma** *step-Stuck*:

**assumes** *step*:  $\Gamma \vdash (c, s) \rightarrow (c', s')$

**shows**  $\bigwedge f. s = \text{Stuck} \Longrightarrow s' = \text{Stuck}$

*<proof>*

**lemma** *SeqSteps*:

**assumes** *steps*:  $\Gamma \vdash \text{cfg}_1 \rightarrow^* \text{cfg}_2$

**shows**  $\bigwedge c_1 \ s \ c_1' \ s'. \llbracket \text{cfg}_1 = (c_1, s); \text{cfg}_2 = (c_1', s') \rrbracket$   
 $\Longrightarrow \Gamma \vdash (\text{Seq } c_1 \ c_2, s) \rightarrow^* (\text{Seq } c_1' \ c_2, s')$

*<proof>*

**lemma** *CatchSteps*:

**assumes** *steps*:  $\Gamma \vdash \text{cfg}_1 \rightarrow^* \text{cfg}_2$

**shows**  $\bigwedge c_1 s c_1' s'. \llbracket \text{cfg}_1 = (c_1, s); \text{cfg}_2 = (c_1', s') \rrbracket$   
 $\implies \Gamma \vdash (\text{Catch } c_1 c_2, s) \rightarrow^* (\text{Catch } c_1' c_2, s')$

*<proof>*

**lemma** *steps-Fault*:  $\Gamma \vdash (c, \text{Fault } f) \rightarrow^* (\text{Skip}, \text{Fault } f)$

*<proof>*

**lemma** *steps-Stuck*:  $\Gamma \vdash (c, \text{Stuck}) \rightarrow^* (\text{Skip}, \text{Stuck})$

*<proof>*

**lemma** *steps-Abrupt*:  $\Gamma \vdash (c, \text{Abrupt } s) \rightarrow^* (\text{Skip}, \text{Abrupt } s)$

*<proof>*

**lemma** *step-Fault-prop*:

**assumes** *step*:  $\Gamma \vdash (c, s) \rightarrow (c', s')$

**shows**  $\bigwedge f. s = \text{Fault } f \implies s' = \text{Fault } f$

*<proof>*

**lemma** *step-Abrupt-prop*:

**assumes** *step*:  $\Gamma \vdash (c, s) \rightarrow (c', s')$

**shows**  $\bigwedge x. s = \text{Abrupt } x \implies s' = \text{Abrupt } x$

*<proof>*

**lemma** *step-Stuck-prop*:

**assumes** *step*:  $\Gamma \vdash (c, s) \rightarrow (c', s')$

**shows**  $s = \text{Stuck} \implies s' = \text{Stuck}$

*<proof>*

**lemma** *steps-Fault-prop*:

**assumes** *step*:  $\Gamma \vdash (c, s) \rightarrow^* (c', s')$

**shows**  $s = \text{Fault } f \implies s' = \text{Fault } f$

*<proof>*

**lemma** *steps-Abrupt-prop*:

**assumes** *step*:  $\Gamma \vdash (c, s) \rightarrow^* (c', s')$

**shows**  $s = \text{Abrupt } t \implies s' = \text{Abrupt } t$

*<proof>*

**lemma** *steps-Stuck-prop*:

**assumes** *step*:  $\Gamma \vdash (c, s) \rightarrow^* (c', s')$

**shows**  $s = \text{Stuck} \implies s' = \text{Stuck}$

*<proof>*

### 8.3 Equivalence between Small-Step and Big-Step Semantics

**theorem** *exec-impl-steps*:

**assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$

**shows**  $\exists c' t'. \Gamma \vdash (c, s) \rightarrow^* (c', t') \wedge$   
*(case t of*  
*Abrupt x  $\Rightarrow$  if s=t then c'=Skip  $\wedge$  t'=t else c'=Throw  $\wedge$  t'=Normal x*  
*| -  $\Rightarrow$  c'=Skip  $\wedge$  t'=t)*

*\langle proof \rangle*

**corollary** *exec-impl-steps-Normal:*

**assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow \text{Normal } t$

**shows**  $\Gamma \vdash (c, s) \rightarrow^* (\text{Skip}, \text{Normal } t)$

*\langle proof \rangle*

**corollary** *exec-impl-steps-Normal-Abrupt:*

**assumes** *exec*:  $\Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \text{Abrupt } t$

**shows**  $\Gamma \vdash (c, \text{Normal } s) \rightarrow^* (\text{Throw}, \text{Normal } t)$

*\langle proof \rangle*

**corollary** *exec-impl-steps-Abrupt-Abrupt:*

**assumes** *exec*:  $\Gamma \vdash \langle c, \text{Abrupt } t \rangle \Rightarrow \text{Abrupt } t$

**shows**  $\Gamma \vdash (c, \text{Abrupt } t) \rightarrow^* (\text{Skip}, \text{Abrupt } t)$

*\langle proof \rangle*

**corollary** *exec-impl-steps-Fault:*

**assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow \text{Fault } f$

**shows**  $\Gamma \vdash (c, s) \rightarrow^* (\text{Skip}, \text{Fault } f)$

*\langle proof \rangle*

**corollary** *exec-impl-steps-Stuck:*

**assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow \text{Stuck}$

**shows**  $\Gamma \vdash (c, s) \rightarrow^* (\text{Skip}, \text{Stuck})$

*\langle proof \rangle*

**lemma** *step-Abrupt-end:*

**assumes** *step*:  $\Gamma \vdash (c_1, s) \rightarrow (c_1', s')$

**shows**  $s' = \text{Abrupt } x \implies s = \text{Abrupt } x$

*\langle proof \rangle*

**lemma** *step-Stuck-end:*

**assumes** *step*:  $\Gamma \vdash (c_1, s) \rightarrow (c_1', s')$

**shows**  $s' = \text{Stuck} \implies$

$s = \text{Stuck} \vee$

$(\exists r x. \text{redex } c_1 = \text{Spec } r \wedge s = \text{Normal } x \wedge (\forall t. (x, t) \notin r)) \vee$

$(\exists p x. \text{redex } c_1 = \text{Call } p \wedge s = \text{Normal } x \wedge \Gamma p = \text{None})$

*\langle proof \rangle*

**lemma** *step-Fault-end:*

**assumes** *step*:  $\Gamma \vdash (c_1, s) \rightarrow (c_1', s')$

**shows**  $s' = \text{Fault } f \implies$

$s = \text{Fault } f \vee$

$(\exists g \ c \ x. \text{redex } c_1 = \text{Guard } f \ g \ c \wedge s = \text{Normal } x \wedge x \notin g)$   
 ⟨proof⟩

**lemma** *exec-redex-Stuck*:  
 $\Gamma \vdash \langle \text{redex } c, s \rangle \Rightarrow \text{Stuck} \Longrightarrow \Gamma \vdash \langle c, s \rangle \Rightarrow \text{Stuck}$   
 ⟨proof⟩

**lemma** *exec-redex-Fault*:  
 $\Gamma \vdash \langle \text{redex } c, s \rangle \Rightarrow \text{Fault } f \Longrightarrow \Gamma \vdash \langle c, s \rangle \Rightarrow \text{Fault } f$   
 ⟨proof⟩

**lemma** *step-extend*:  
 assumes *step*:  $\Gamma \vdash (c, s) \rightarrow (c', s')$   
 shows  $\bigwedge t. \Gamma \vdash \langle c', s' \rangle \Rightarrow t \Longrightarrow \Gamma \vdash \langle c, s \rangle \Rightarrow t$   
 ⟨proof⟩

**theorem** *steps-Skip-impl-exec*:  
 assumes *steps*:  $\Gamma \vdash (c, s) \rightarrow^* (\text{Skip}, t)$   
 shows  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
 ⟨proof⟩

**theorem** *steps-Throw-impl-exec*:  
 assumes *steps*:  $\Gamma \vdash (c, s) \rightarrow^* (\text{Throw}, \text{Normal } t)$   
 shows  $\Gamma \vdash \langle c, s \rangle \Rightarrow \text{Abrupt } t$   
 ⟨proof⟩

## 8.4 Infinite Computations: $\Gamma \vdash (c, s) \rightarrow \dots(\infty)$

**definition** *inf*::  $(s, p, f) \text{ body} \Rightarrow (s, p, f) \text{ config} \Rightarrow \text{bool}$   
 $(\langle \vdash - \rightarrow \dots(\infty) \rangle) [60, 80] 100$  **where**  
 $\Gamma \vdash \text{cfg} \rightarrow \dots(\infty) \equiv (\exists f. f \ (0::\text{nat}) = \text{cfg} \wedge (\forall i. \Gamma \vdash f \ i \rightarrow f \ (i+1)))$

**lemma** *not-infI*:  $\llbracket \bigwedge f. \llbracket f \ 0 = \text{cfg}; \bigwedge i. \Gamma \vdash f \ i \rightarrow f \ (\text{Suc } i) \rrbracket \Longrightarrow \text{False} \rrbracket$   
 $\Longrightarrow \neg \Gamma \vdash \text{cfg} \rightarrow \dots(\infty)$   
 ⟨proof⟩

## 8.5 Equivalence between Termination and the Absence of Infinite Computations

**lemma** *step-preserves-termination*:  
 assumes *step*:  $\Gamma \vdash (c, s) \rightarrow (c', s')$   
 shows  $\Gamma \vdash c \downarrow s \Longrightarrow \Gamma \vdash c' \downarrow s'$   
 ⟨proof⟩

**lemma** *steps-preserves-termination*:  
 assumes *steps*:  $\Gamma \vdash (c, s) \rightarrow^* (c', s')$   
 shows  $\Gamma \vdash c \downarrow s \Longrightarrow \Gamma \vdash c' \downarrow s'$   
 ⟨proof⟩

$\langle ML \rangle$

**lemma** *steps-preserves-termination'*:

**assumes** *steps*:  $\Gamma \vdash (c, s) \rightarrow^+ (c', s')$

**shows**  $\Gamma \vdash c \downarrow s \implies \Gamma \vdash c' \downarrow s'$

$\langle proof \rangle$

**definition** *head-com*::  $(s', p', f) \text{ com} \Rightarrow (s', p', f) \text{ com}$

**where**

*head-com*  $c =$

(*case*  $c$  of

$Seq\ c_1\ c_2 \Rightarrow c_1$

$Catch\ c_1\ c_2 \Rightarrow c_1$

$- \Rightarrow c$ )

**definition** *head*::  $(s', p', f) \text{ config} \Rightarrow (s', p', f) \text{ config}$

**where** *head*  $cfg = (\text{head-com } (fst\ cfg), snd\ cfg)$

**lemma** *le-Suc-cases*:  $\llbracket \bigwedge i. \llbracket i < k \rrbracket \implies P\ i; P\ k \rrbracket \implies \forall i < (Suc\ k). P\ i$

$\langle proof \rangle$

**lemma** *redex-Seq-False*:  $\bigwedge c' c''. (\text{redex } c = Seq\ c''\ c') = False$

$\langle proof \rangle$

**lemma** *redex-Catch-False*:  $\bigwedge c' c''. (\text{redex } c = Catch\ c''\ c') = False$

$\langle proof \rangle$

**lemma** *infinite-computation-extract-head-Seq*:

**assumes** *inf-comp*:  $\forall i::nat. \Gamma \vdash f\ i \rightarrow f\ (i+1)$

**assumes** *f-0*:  $f\ 0 = (Seq\ c_1\ c_2, s)$

**assumes** *not-fin*:  $\forall i < k. \neg final\ (\text{head } (f\ i))$

**shows**  $\forall i < k. (\exists c' s'. f\ (i+1) = (Seq\ c'\ c_2, s')) \wedge$

$\Gamma \vdash \text{head } (f\ i) \rightarrow \text{head } (f\ (i+1))$

(**is**  $\forall i < k. ?P\ i$ )

$\langle proof \rangle$

**lemma** *infinite-computation-extract-head-Catch*:

**assumes** *inf-comp*:  $\forall i::nat. \Gamma \vdash f\ i \rightarrow f\ (i+1)$

**assumes** *f-0*:  $f\ 0 = (Catch\ c_1\ c_2, s)$

**assumes** *not-fin*:  $\forall i < k. \neg final\ (\text{head } (f\ i))$

**shows**  $\forall i < k. (\exists c' s'. f\ (i+1) = (Catch\ c'\ c_2, s')) \wedge$

$\Gamma \vdash \text{head } (f\ i) \rightarrow \text{head } (f\ (i+1))$

(**is**  $\forall i < k. ?P\ i$ )

$\langle proof \rangle$

**lemma** *no-inf-Throw*:  $\neg \Gamma \vdash (\text{Throw}, s) \rightarrow \dots(\infty)$   
 ⟨proof⟩

**lemma** *split-inf-Seq*:  
 assumes *inf-comp*:  $\Gamma \vdash (\text{Seq } c_1 \ c_2, s) \rightarrow \dots(\infty)$   
 shows  $\Gamma \vdash (c_1, s) \rightarrow \dots(\infty) \vee$   
 $(\exists s'. \Gamma \vdash (c_1, s) \rightarrow^* (\text{Skip}, s') \wedge \Gamma \vdash (c_2, s') \rightarrow \dots(\infty))$   
 ⟨proof⟩

**lemma** *split-inf-Catch*:  
 assumes *inf-comp*:  $\Gamma \vdash (\text{Catch } c_1 \ c_2, s) \rightarrow \dots(\infty)$   
 shows  $\Gamma \vdash (c_1, s) \rightarrow \dots(\infty) \vee$   
 $(\exists s'. \Gamma \vdash (c_1, s) \rightarrow^* (\text{Throw}, \text{Normal } s') \wedge \Gamma \vdash (c_2, \text{Normal } s') \rightarrow \dots(\infty))$   
 ⟨proof⟩

**lemma** *Skip-no-step*:  $\Gamma \vdash (\text{Skip}, s) \rightarrow \text{cfg} \implies P$   
 ⟨proof⟩

**lemma** *not-inf-Stuck*:  $\neg \Gamma \vdash (c, \text{Stuck}) \rightarrow \dots(\infty)$   
 ⟨proof⟩

**lemma** *not-inf-Fault*:  $\neg \Gamma \vdash (c, \text{Fault } x) \rightarrow \dots(\infty)$   
 ⟨proof⟩

**lemma** *not-inf-Abrupt*:  $\neg \Gamma \vdash (c, \text{Abrupt } s) \rightarrow \dots(\infty)$   
 ⟨proof⟩

**theorem** *terminates-impl-no-infinite-computation*:  
 assumes *termi*:  $\Gamma \vdash c \downarrow s$   
 shows  $\neg \Gamma \vdash (c, s) \rightarrow \dots(\infty)$   
 ⟨proof⟩

**definition**

*termi-call-steps* ::  $(s', p', f) \text{ body} \Rightarrow ((s' \times p') \times (s' \times p')) \text{ set}$

**where**

*termi-call-steps*  $\Gamma =$   
 $\{((t, q), (s, p)). \Gamma \vdash \text{Call } p \downarrow \text{Normal } s \wedge$   
 $(\exists c. \Gamma \vdash (\text{Call } p, \text{Normal } s) \rightarrow^+ (c, \text{Normal } t) \wedge \text{redex } c = \text{Call } q)\}$

**primrec** *subst-redex*::  $(s', p', f) \text{ com} \Rightarrow (s', p', f) \text{ com} \Rightarrow (s', p', f) \text{ com}$

**where**

*subst-redex* *Skip*  $c = c$  |  
*subst-redex* (*Basic*  $f$ )  $c = c$  |  
*subst-redex* (*Spec*  $r$ )  $c = c$  |  
*subst-redex* (*Seq*  $c_1 \ c_2$ )  $c = \text{Seq } (\text{subst-redex } c_1 \ c) \ c_2$  |  
*subst-redex* (*Cond*  $b \ c_1 \ c_2$ )  $c = c$  |

$\text{subst-redex } (\text{While } b \ c') \ c = c \mid$   
 $\text{subst-redex } (\text{Call } p) \ c = c \mid$   
 $\text{subst-redex } (\text{DynCom } d) \ c = c \mid$   
 $\text{subst-redex } (\text{Guard } f \ b \ c') \ c = c \mid$   
 $\text{subst-redex } (\text{Throw}) \ c = c \mid$   
 $\text{subst-redex } (\text{Catch } c_1 \ c_2) \ c = \text{Catch } (\text{subst-redex } c_1 \ c) \ c_2$

**lemma** *subst-redex-redex*:  
 $\text{subst-redex } c \ (\text{redex } c) = c$   
 $\langle \text{proof} \rangle$

**lemma** *redex-subst-redex*:  $\text{redex } (\text{subst-redex } c \ r) = \text{redex } r$   
 $\langle \text{proof} \rangle$

**lemma** *step-redex'*:  
**shows**  $\Gamma \vdash (\text{redex } c, s) \rightarrow (r', s') \implies \Gamma \vdash (c, s) \rightarrow (\text{subst-redex } c \ r', s')$   
 $\langle \text{proof} \rangle$

**lemma** *step-redex*:  
**shows**  $\Gamma \vdash (r, s) \rightarrow (r', s') \implies \Gamma \vdash (\text{subst-redex } c \ r, s) \rightarrow (\text{subst-redex } c \ r', s')$   
 $\langle \text{proof} \rangle$

**lemma** *steps-redex*:  
**assumes** *steps*:  $\Gamma \vdash (r, s) \rightarrow^* (r', s')$   
**shows**  $\bigwedge c. \Gamma \vdash (\text{subst-redex } c \ r, s) \rightarrow^* (\text{subst-redex } c \ r', s')$   
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

**lemma** *steps-redex'*:  
**assumes** *steps*:  $\Gamma \vdash (r, s) \rightarrow^+ (r', s')$   
**shows**  $\bigwedge c. \Gamma \vdash (\text{subst-redex } c \ r, s) \rightarrow^+ (\text{subst-redex } c \ r', s')$   
 $\langle \text{proof} \rangle$

**primrec** *seq*::  $(\text{nat} \Rightarrow ('s, 'p, 'f)\text{com}) \Rightarrow 'p \Rightarrow \text{nat} \Rightarrow ('s, 'p, 'f)\text{com}$   
**where**

$\text{seq } c \ p \ 0 = \text{Call } p \mid$   
 $\text{seq } c \ p \ (\text{Suc } i) = \text{subst-redex } (\text{seq } c \ p \ i) \ (c \ i)$

**lemma** *renumber'*:  
**assumes** *f*:  $\forall i. (a, f \ i) \in r^* \wedge (f \ i, f(\text{Suc } i)) \in r$   
**assumes** *a-b*:  $(a, b) \in r^*$   
**shows**  $b = f \ 0 \implies (\exists f. f \ 0 = a \wedge (\forall i. (f \ i, f(\text{Suc } i)) \in r))$   
 $\langle \text{proof} \rangle$

**lemma** *renumber*:  
 $\forall i. (a, f \ i) \in r^* \wedge (f \ i, f(\text{Suc } i)) \in r$

$\implies \exists f. f \ 0 = a \wedge (\forall i. (f \ i, f(\text{Suc } i)) \in r)$   
 ⟨proof⟩

**lemma** *lem*:

$\forall y. r^{++} \ a \ y \longrightarrow P \ a \ \longrightarrow P \ y$   
 $\implies ((b,a) \in \{(y,x). P \ x \wedge r \ x \ y\}^+) = ((b,a) \in \{(y,x). P \ x \wedge r^{++} \ x \ y\})$   
 ⟨proof⟩

**corollary** *terminates-impl-no-infinite-trans-computation*:

**assumes** *terminates*:  $\Gamma \vdash c \downarrow s$   
**shows**  $\neg(\exists f. f \ 0 = (c,s) \wedge (\forall i. \Gamma \vdash f \ i \ \rightarrow^+ f(\text{Suc } i)))$   
 ⟨proof⟩

**theorem** *wf-termi-call-steps*: *wf* (*termi-call-steps*  $\Gamma$ )

⟨proof⟩

**lemma** *no-infinite-computation-implies-wf*:

**assumes** *not-inf*:  $\neg \Gamma \vdash (c, s) \rightarrow \dots(\infty)$   
**shows** *wf*  $\{(c2,c1). \Gamma \vdash (c,s) \rightarrow^* c1 \wedge \Gamma \vdash c1 \rightarrow c2\}$   
 ⟨proof⟩

**lemma** *not-final-Stuck-step*:  $\neg \text{final } (c, \text{Stuck}) \implies \exists c' \ s'. \Gamma \vdash (c, \text{Stuck}) \rightarrow (c', s')$

⟨proof⟩

**lemma** *not-final-Abrupt-step*:

$\neg \text{final } (c, \text{Abrupt } s) \implies \exists c' \ s'. \Gamma \vdash (c, \text{Abrupt } s) \rightarrow (c', s')$   
 ⟨proof⟩

**lemma** *not-final-Fault-step*:

$\neg \text{final } (c, \text{Fault } f) \implies \exists c' \ s'. \Gamma \vdash (c, \text{Fault } f) \rightarrow (c', s')$   
 ⟨proof⟩

**lemma** *not-final-Normal-step*:

$\neg \text{final } (c, \text{Normal } s) \implies \exists c' \ s'. \Gamma \vdash (c, \text{Normal } s) \rightarrow (c', s')$   
 ⟨proof⟩

**lemma** *final-termi*:

*final*  $(c,s) \implies \Gamma \vdash c \downarrow s$   
 ⟨proof⟩

**lemma** *split-computation*:

**assumes** *steps*:  $\Gamma \vdash (c, s) \rightarrow^* (c_f, s_f)$   
**assumes** *not-final*:  $\neg \text{final } (c,s)$   
**assumes** *final*: *final*  $(c_f, s_f)$   
**shows**  $\exists c' \ s'. \Gamma \vdash (c, s) \rightarrow (c', s') \wedge \Gamma \vdash (c', s') \rightarrow^* (c_f, s_f)$   
 ⟨proof⟩

**lemma** *wf-implies-termi-reach-step-case:*  
**assumes** *hyp*:  $\bigwedge c' s'. \Gamma \vdash (c, \text{Normal } s) \rightarrow (c', s') \implies \Gamma \vdash c' \downarrow s'$   
**shows**  $\Gamma \vdash c \downarrow \text{Normal } s$   
 $\langle \text{proof} \rangle$

**lemma** *wf-implies-termi-reach:*  
**assumes** *wf*:  $wf \{ (cfg2, cfg1). \Gamma \vdash (c, s) \rightarrow^* cfg1 \wedge \Gamma \vdash cfg1 \rightarrow cfg2 \}$   
**shows**  $\bigwedge c1 s1. \llbracket \Gamma \vdash (c, s) \rightarrow^* cfg1; \text{cfg1} = (c1, s1) \rrbracket \implies \Gamma \vdash c1 \downarrow s1$   
 $\langle \text{proof} \rangle$

**theorem** *no-infinite-computation-impl-terminates:*  
**assumes** *not-inf*:  $\neg \Gamma \vdash (c, s) \rightarrow \dots(\infty)$   
**shows**  $\Gamma \vdash c \downarrow s$   
 $\langle \text{proof} \rangle$

**corollary** *terminates-iff-no-infinite-computation:*  
 $\Gamma \vdash c \downarrow s = (\neg \Gamma \vdash (c, s) \rightarrow \dots(\infty))$   
 $\langle \text{proof} \rangle$

## 8.6 Generalised Redexes

For an important lemma for the completeness proof of the Hoare-logic for total correctness we need a generalisation of *redex* that not only yield the redex itself but all the enclosing statements as well.

**primrec** *redexes*::  $(s, p, f) \text{com} \Rightarrow (s, p, f) \text{com set}$   
**where**

*redexes* *Skip* =  $\{ \text{Skip} \}$  |  
*redexes* (*Basic f*) =  $\{ \text{Basic } f \}$  |  
*redexes* (*Spec r*) =  $\{ \text{Spec } r \}$  |  
*redexes* (*Seq c<sub>1</sub> c<sub>2</sub>*) =  $\{ \text{Seq } c_1 c_2 \} \cup \text{redexes } c_1$  |  
*redexes* (*Cond b c<sub>1</sub> c<sub>2</sub>*) =  $\{ \text{Cond } b c_1 c_2 \}$  |  
*redexes* (*While b c*) =  $\{ \text{While } b c \}$  |  
*redexes* (*Call p*) =  $\{ \text{Call } p \}$  |  
*redexes* (*DynCom d*) =  $\{ \text{DynCom } d \}$  |  
*redexes* (*Guard f b c*) =  $\{ \text{Guard } f b c \}$  |  
*redexes* (*Throw*) =  $\{ \text{Throw} \}$  |  
*redexes* (*Catch c<sub>1</sub> c<sub>2</sub>*) =  $\{ \text{Catch } c_1 c_2 \} \cup \text{redexes } c_1$

**lemma** *root-in-redexes*:  $c \in \text{redexes } c$   
 $\langle \text{proof} \rangle$

**lemma** *redex-in-redexes*:  $\text{redex } c \in \text{redexes } c$   
 $\langle \text{proof} \rangle$

**lemma** *redex-redexes*:  $\bigwedge c'. \llbracket c' \in \text{redexes } c; \text{redex } c' = c \rrbracket \implies \text{redex } c = c'$   
 $\langle \text{proof} \rangle$

**lemma** *step-redexes*:  
**shows**  $\bigwedge r r'. \llbracket \Gamma \vdash (r, s) \rightarrow (r', s'); r \in \text{redexes } c \rrbracket$

$\implies \exists c'. \Gamma \vdash (c, s) \rightarrow (c', s') \wedge r' \in \text{redexes } c'$   
 \langle proof \rangle

**lemma** *steps-redexes*:

**assumes** *steps*:  $\Gamma \vdash (r, s) \rightarrow^* (r', s')$   
**shows**  $\bigwedge c. r \in \text{redexes } c \implies \exists c'. \Gamma \vdash (c, s) \rightarrow^* (c', s') \wedge r' \in \text{redexes } c'$   
 \langle proof \rangle

**lemma** *steps-redexes'*:

**assumes** *steps*:  $\Gamma \vdash (r, s) \rightarrow^+ (r', s')$   
**shows**  $\bigwedge c. r \in \text{redexes } c \implies \exists c'. \Gamma \vdash (c, s) \rightarrow^+ (c', s') \wedge r' \in \text{redexes } c'$   
 \langle proof \rangle

**lemma** *step-redexes-Seq*:

**assumes** *step*:  $\Gamma \vdash (r, s) \rightarrow (r', s')$   
**assumes** *Seq*:  $\text{Seq } r \ c_2 \in \text{redexes } c$   
**shows**  $\exists c'. \Gamma \vdash (c, s) \rightarrow (c', s') \wedge \text{Seq } r' \ c_2 \in \text{redexes } c'$   
 \langle proof \rangle

**lemma** *steps-redexes-Seq*:

**assumes** *steps*:  $\Gamma \vdash (r, s) \rightarrow^* (r', s')$   
**shows**  $\bigwedge c. \text{Seq } r \ c_2 \in \text{redexes } c \implies$   
 $\quad \exists c'. \Gamma \vdash (c, s) \rightarrow^* (c', s') \wedge \text{Seq } r' \ c_2 \in \text{redexes } c'$   
 \langle proof \rangle

**lemma** *steps-redexes-Seq'*:

**assumes** *steps*:  $\Gamma \vdash (r, s) \rightarrow^+ (r', s')$   
**shows**  $\bigwedge c. \text{Seq } r \ c_2 \in \text{redexes } c$   
 $\implies \exists c'. \Gamma \vdash (c, s) \rightarrow^+ (c', s') \wedge \text{Seq } r' \ c_2 \in \text{redexes } c'$   
 \langle proof \rangle

**lemma** *step-redexes-Catch*:

**assumes** *step*:  $\Gamma \vdash (r, s) \rightarrow (r', s')$   
**assumes** *Catch*:  $\text{Catch } r \ c_2 \in \text{redexes } c$   
**shows**  $\exists c'. \Gamma \vdash (c, s) \rightarrow (c', s') \wedge \text{Catch } r' \ c_2 \in \text{redexes } c'$   
 \langle proof \rangle

**lemma** *steps-redexes-Catch*:

**assumes** *steps*:  $\Gamma \vdash (r, s) \rightarrow^* (r', s')$   
**shows**  $\bigwedge c. \text{Catch } r \ c_2 \in \text{redexes } c \implies$   
 $\quad \exists c'. \Gamma \vdash (c, s) \rightarrow^* (c', s') \wedge \text{Catch } r' \ c_2 \in \text{redexes } c'$   
 \langle proof \rangle

**lemma** *steps-redexes-Catch'*:

**assumes** *steps*:  $\Gamma \vdash (r, s) \rightarrow^+ (r', s')$   
**shows**  $\bigwedge c. \text{Catch } r \ c_2 \in \text{redexes } c$   
 $\implies \exists c'. \Gamma \vdash (c, s) \rightarrow^+ (c', s') \wedge \text{Catch } r' \ c_2 \in \text{redexes } c'$

*<proof>*

**lemma** *redexes-subset*:  $\bigwedge c'. c' \in \text{redexes } c \implies \text{redexes } c' \subseteq \text{redexes } c$   
*<proof>*

**lemma** *redexes-preserves-termination*:

**assumes** *termi*:  $\Gamma \vdash c \downarrow s$

**shows**  $\bigwedge c'. c' \in \text{redexes } c \implies \Gamma \vdash c' \downarrow s$

*<proof>*

**end**

## 9 Hoare Logic for Total Correctness

**theory** *HoareTotalDef* **imports** *HoarePartialDef Termination* **begin**

### 9.1 Validity of Hoare Tuples: $\Gamma \models_{t/F} P \ c \ Q, A$

**definition**

*validt* ::  $[(\ 's, 'p, 'f) \text{ body}, 'f \text{ set}, 's \text{ assn}, (\ 's, 'p, 'f) \text{ com}, 's \text{ assn}, 's \text{ assn}] \Rightarrow \text{bool}$   
 $(\ \langle -, \cdot \rangle \models_{t/F} / - \ - \ - \rangle \rightarrow [61, 60, 1000, 20, 1000, 1000] \ 60)$

**where**

$\Gamma \models_{t/F} P \ c \ Q, A \equiv \Gamma \models_{/F} P \ c \ Q, A \wedge (\forall s \in \text{Normal } \langle P. \Gamma \vdash c \downarrow s \rangle)$

**definition**

*cvalidt* ::

$[(\ 's, 'p, 'f) \text{ body}, (\ 's, 'p) \text{ quadruple set}, 'f \text{ set},$   
 $'s \text{ assn}, (\ 's, 'p, 'f) \text{ com}, 's \text{ assn}, 's \text{ assn}] \Rightarrow \text{bool}$   
 $(\ \langle -, \cdot \rangle \models_{t/F} / - \ - \ - \rangle \rightarrow [61, 60, 60, 1000, 20, 1000, 1000] \ 60)$

**where**

$\Gamma, \Theta \models_{t/F} P \ c \ Q, A \equiv (\forall (P, p, Q, A) \in \Theta. \Gamma \models_{t/F} P \ (\text{Call } p) \ Q, A) \longrightarrow \Gamma \models_{t/F} P \ c \ Q, A$

**notation** (*ASCII*)

*validt*  $(\ \langle -, \cdot \rangle \models_{t/F} / - \ - \ - \rangle \rightarrow [61, 60, 1000, 20, 1000, 1000] \ 60)$  **and**  
*cvalidt*  $(\ \langle -, \cdot \rangle \models_{t/F} / - \ - \ - \rangle \rightarrow [61, 60, 60, 1000, 20, 1000, 1000] \ 60)$

### 9.2 Properties of Validity

**lemma** *validtI*:

$\llbracket \bigwedge s \ t. \llbracket \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow t; s \in P; t \notin \text{Fault } \langle F \rangle \rrbracket \implies t \in \text{Normal } \langle Q \cup \text{Abrupt } \langle A \rangle$

$\bigwedge s. s \in P \implies \Gamma \vdash c \downarrow (\text{Normal } s) \rrbracket$

$\implies \Gamma \models_{t/F} P \ c \ Q, A$

*<proof>*

**lemma** *cvalidtI*:

$$\begin{aligned} & \llbracket \bigwedge s. t. \llbracket \forall (P,p,Q,A) \in \Theta. \Gamma \models_{t/F} P \text{ (Call } p) \text{ } Q, A; \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow t; s \in P; \\ & \quad t \notin \text{Fault } ' F \rrbracket \\ & \quad \Rightarrow t \in \text{Normal } ' Q \cup \text{Abrupt } ' A; \\ & \bigwedge s. \llbracket \forall (P,p,Q,A) \in \Theta. \Gamma \models_{t/F} P \text{ (Call } p) \text{ } Q, A; s \in P \rrbracket \Rightarrow \Gamma \vdash c \downarrow (\text{Normal } s) \rrbracket \\ & \Rightarrow \Gamma, \Theta \models_{t/F} P \text{ } c \text{ } Q, A \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *cvalidt-postD*:

$$\begin{aligned} & \llbracket \Gamma, \Theta \models_{t/F} P \text{ } c \text{ } Q, A; \forall (P,p,Q,A) \in \Theta. \Gamma \models_{t/F} P \text{ (Call } p) \text{ } Q, A; \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \\ & t; \\ & \quad s \in P; t \notin \text{Fault } ' F \rrbracket \\ & \Rightarrow t \in \text{Normal } ' Q \cup \text{Abrupt } ' A \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *cvalidt-termD*:

$$\begin{aligned} & \llbracket \Gamma, \Theta \models_{t/F} P \text{ } c \text{ } Q, A; \forall (P,p,Q,A) \in \Theta. \Gamma \models_{t/F} P \text{ (Call } p) \text{ } Q, A; s \in P \rrbracket \\ & \Rightarrow \Gamma \vdash c \downarrow (\text{Normal } s) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *validt-augment-Faults*:

$$\begin{aligned} & \text{assumes } \text{valid} : \Gamma \models_{t/F} P \text{ } c \text{ } Q, A \\ & \text{assumes } F' : F \subseteq F' \\ & \text{shows } \Gamma \models_{t/F'} P \text{ } c \text{ } Q, A \\ & \langle \text{proof} \rangle \end{aligned}$$

### 9.3 The Hoare Rules: $\Gamma, \Theta \vdash_{t/F} P \text{ } c \text{ } Q, A$

**inductive** *hoaret*::[(*'s, 'p, 'f*) *body*, (*'s, 'p*) *quadruple set*, *'f set*,  
*'s assn*, (*'s, 'p, 'f*) *com*, *'s assn*, *'s assn*]  
 $\Rightarrow$  *bool*  
 ( $\langle (\exists -, \vdash_{t'/F} (-) / -) \rangle [61, 60, 60, 1000, 20, 1000, 1000] 60$ )  
**for**  $\Gamma :: ('s, 'p, 'f)$  *body*

**where**

$$\text{Skip} : \Gamma, \Theta \vdash_{t/F} Q \text{ Skip } Q, A$$

$$| \text{Basic} : \Gamma, \Theta \vdash_{t/F} \{s. f \text{ } s \in Q\} (\text{Basic } f) \text{ } Q, A$$

$$| \text{Spec} : \Gamma, \Theta \vdash_{t/F} \{s. (\forall t. (s, t) \in r \longrightarrow t \in Q) \wedge (\exists t. (s, t) \in r)\} (\text{Spec } r) \text{ } Q, A$$

$$\begin{aligned} | \text{Seq} : & \llbracket \Gamma, \Theta \vdash_{t/F} P \text{ } c_1 \text{ } R, A; \Gamma, \Theta \vdash_{t/F} R \text{ } c_2 \text{ } Q, A \rrbracket \\ & \Rightarrow \\ & \Gamma, \Theta \vdash_{t/F} P \text{ Seq } c_1 \text{ } c_2 \text{ } Q, A \end{aligned}$$

$$\begin{aligned} | \text{Cond} : & \llbracket \Gamma, \Theta \vdash_{t/F} (P \cap b) \text{ } c_1 \text{ } Q, A; \Gamma, \Theta \vdash_{t/F} (P \cap - b) \text{ } c_2 \text{ } Q, A \rrbracket \\ & \Rightarrow \end{aligned}$$

- $$\Gamma, \Theta \vdash_{t/F} P \text{ (Cond } b \ c_1 \ c_2) \ Q, A$$
- | *While*:  $\llbracket wf \ r; \forall \sigma. \Gamma, \Theta \vdash_{t/F} (\{\sigma\} \cap P \cap b) \ c \ (\{t. (t, \sigma) \in r\} \cap P), A \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash_{t/F} P \text{ (While } b \ c) \ (P \cap - \ b), A$
- | *Guard*:  $\Gamma, \Theta \vdash_{t/F} (g \cap P) \ c \ Q, A$   
 $\implies$   
 $\Gamma, \Theta \vdash_{t/F} (g \cap P) \text{ Guard } f \ g \ c \ Q, A$
- | *Guarantee*:  $\llbracket f \in F; \Gamma, \Theta \vdash_{t/F} (g \cap P) \ c \ Q, A \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash_{t/F} P \text{ (Guard } f \ g \ c) \ Q, A$
- | *CallRec*:  
 $\llbracket (P, p, Q, A) \in \text{Specs};$   
 $wf \ r;$   
 $\text{Specs-wf} = (\lambda p \ \sigma. (\lambda (P, q, Q, A). (P \cap \{s. ((s, q), (\sigma, p)) \in r\}, q, Q, A))) \text{ 'Specs};$   
 $\forall (P, p, Q, A) \in \text{Specs}.$   
 $p \in \text{dom } \Gamma \wedge (\forall \sigma. \Gamma, \Theta \cup \text{Specs-wf } p \ \sigma \vdash_{t/F} (\{\sigma\} \cap P) \text{ (the } (\Gamma \ p)) \ Q, A)$   
 $\rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash_{t/F} P \text{ (Call } p) \ Q, A$
- | *DynCom*:  $\forall s \in P. \Gamma, \Theta \vdash_{t/F} P \ (c \ s) \ Q, A$   
 $\implies$   
 $\Gamma, \Theta \vdash_{t/F} P \text{ (DynCom } c) \ Q, A$
- | *Throw*:  $\Gamma, \Theta \vdash_{t/F} A \ \text{Throw } Q, A$
- | *Catch*:  $\llbracket \Gamma, \Theta \vdash_{t/F} P \ c_1 \ Q, R; \Gamma, \Theta \vdash_{t/F} R \ c_2 \ Q, A \rrbracket \implies \Gamma, \Theta \vdash_{t/F} P \ \text{Catch } c_1 \ c_2 \ Q, A$
- | *Conseq*:  $\forall s \in P. \exists P' \ Q' \ A'. \Gamma, \Theta \vdash_{t/F} P' \ c \ Q', A' \wedge s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A$   
 $\implies \Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$
- | *Asm*:  $(P, p, Q, A) \in \Theta$   
 $\implies$   
 $\Gamma, \Theta \vdash_{t/F} P \text{ (Call } p) \ Q, A$
- | *ExFalso*:  $\llbracket \Gamma, \Theta \models_{t/F} P \ c \ Q, A; \neg \Gamma \models_{t/F} P \ c \ Q, A \rrbracket \implies \Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$   
— This is a hack rule that enables us to derive completeness for an arbitrary context  $\Theta$ , from completeness for an empty context.

Does not work, because of rule ExFalso, the context  $\Theta$  is to blame. A weaker version with empty context can be derived from soundness later on.

**lemma** *hoaret-to-hoarep*:  
**assumes** *hoaret*:  $\Gamma, \Theta \vdash_{t/F} P \ p \ Q, A$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P \ p \ Q, A$   
*<proof>*

**lemma** *hoaret-augment-context*:  
**assumes** *deriv*:  $\Gamma, \Theta \vdash_{t/F} P \ p \ Q, A$   
**shows**  $\bigwedge \Theta'. \Theta \subseteq \Theta' \implies \Gamma, \Theta' \vdash_{t/F} P \ p \ Q, A$   
*<proof>*

## 9.4 Some Derived Rules

**lemma** *Conseq'*:  $\forall s. s \in P \longrightarrow$   
 $(\exists P' Q' A'.$   
 $(\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \ c \ (Q' Z), (A' Z)) \wedge$   
 $(\exists Z. s \in P' Z \wedge (Q' Z \subseteq Q) \wedge (A' Z \subseteq A)))$   
 $\implies$   
 $\Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$   
*<proof>*

**lemma** *conseq*:  $\llbracket \forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \ c \ (Q' Z), (A' Z);$   
 $\forall s. s \in P \longrightarrow (\exists Z. s \in P' Z \wedge (Q' Z \subseteq Q) \wedge (A' Z \subseteq A)) \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$   
*<proof>*

**theorem** *conseqPrePost*:  
 $\Gamma, \Theta \vdash_{t/F} P' \ c \ Q', A' \implies P \subseteq P' \implies Q' \subseteq Q \implies A' \subseteq A \implies \Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$   
*<proof>*

**lemma** *conseqPre*:  $\Gamma, \Theta \vdash_{t/F} P' \ c \ Q, A \implies P \subseteq P' \implies \Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$   
*<proof>*

**lemma** *conseqPost*:  $\Gamma, \Theta \vdash_{t/F} P \ c \ Q', A' \implies Q' \subseteq Q \implies A' \subseteq A \implies \Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$   
*<proof>*

**lemma** *Spec-wf-conv*:  
 $(\lambda(P, q, Q, A). (P \cap \{s. ((s, q), \tau, p) \in r\}, q, Q, A)) \ ' =$   
 $(\bigcup_{p \in Procs.} \bigcup Z. \{(P \ p \ Z, p, Q \ p \ Z, A \ p \ Z)\}) =$   
 $(\bigcup_{q \in Procs.} \bigcup Z. \{(P \ q \ Z \cap \{s. ((s, q), \tau, p) \in r\}, q, Q \ q \ Z, A \ q \ Z)\})$   
*<proof>*

**lemma** *CallRec'*:  
 $\llbracket p \in Procs; Procs \subseteq dom \Gamma; \rrbracket$

$wf\ r;$   
 $\forall p \in Procs. \forall \tau\ Z.$   
 $\Gamma, \Theta \cup (\bigcup q \in Procs. \bigcup Z.$   
 $\{((P\ q\ Z) \cap \{s. ((s, q), (\tau, p)) \in r\}, q, Q\ q\ Z, (A\ q\ Z))\})$   
 $\vdash_{t/F} (\{\tau\} \cap (P\ p\ Z))\ (the\ (\Gamma\ p))\ (Q\ p\ Z), (A\ p\ Z)]$   
 $\implies$   
 $\Gamma, \Theta \vdash_{t/F} (P\ p\ Z)\ (Call\ p)\ (Q\ p\ Z), (A\ p\ Z)$   
 $\langle proof \rangle$

end

## 10 Properties of Total Correctness Hoare Logic

**theory** *HoareTotalProps* **imports** *SmallStep HoareTotalDef HoarePartialProps* **begin**

### 10.1 Soundness

**lemma** *hoaret-sound*:  
**assumes** *hoare*:  $\Gamma, \Theta \vdash_{t/F} P\ c\ Q, A$   
**shows**  $\Gamma, \Theta \models_{t/F} P\ c\ Q, A$   
 $\langle proof \rangle$

**lemma** *hoaret-sound'*:  
 $\Gamma, \{\}\vdash_{t/F} P\ c\ Q, A \implies \Gamma \models_{t/F} P\ c\ Q, A$   
 $\langle proof \rangle$

**theorem** *total-to-partial*:  
**assumes** *total*:  $\Gamma, \{\}\vdash_{t/F} P\ c\ Q, A$  **shows**  $\Gamma, \{\}\vdash_{t/F} P\ c\ Q, A$   
 $\langle proof \rangle$

### 10.2 Completeness

**lemma** *MGT-valid*:  
 $\Gamma \models_{t/F} \{s. s = Z \wedge \Gamma \vdash \langle c, Normal\ s \rangle \Rightarrow \notin (\{Stuck\} \cup Fault\ '(-F)) \wedge \Gamma \vdash c \downarrow Normal\ s\}$   
 $c$   
 $\{t. \Gamma \vdash \langle c, Normal\ Z \rangle \Rightarrow Normal\ t\}, \{t. \Gamma \vdash \langle c, Normal\ Z \rangle \Rightarrow Abrupt\ t\}$   
 $\langle proof \rangle$

The consequence rule where the existential  $Z$  is instantiated to  $s$ . Usefull in proof of *MGT-lemma*.

**lemma** *ConseqMGT*:  
**assumes** *modif*:  $\forall Z::'a. \Gamma, \Theta \vdash_{t/F} (P'\ Z::'a\ assn)\ c\ (Q'\ Z), (A'\ Z)$   
**assumes** *impl*:  $\bigwedge s. s \in P \implies s \in P'\ s \wedge (\forall t. t \in Q'\ s \longrightarrow t \in Q) \wedge$   
 $(\forall t. t \in A'\ s \longrightarrow t \in A)$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P\ c\ Q, A$   
 $\langle proof \rangle$

**lemma** *MGT-implies-complete:*

**assumes** *MGT*:  $\forall Z. \Gamma, \{\} \vdash_{t/F} \{s. s=Z \wedge \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } (-F)) \wedge$   
 $\Gamma \vdash c \downarrow \text{Normal } s\}$   
 $c$   
 $\{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$   
 $\{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$   
**assumes** *valid*:  $\Gamma \models_{t/F} P \ c \ Q, A$   
**shows**  $\Gamma, \{\} \vdash_{t/F} P \ c \ Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *conseq-extract-state-indep-prop:*

**assumes** *state-indep-prop*:  $\forall s \in P. R$   
**assumes** *to-show*:  $R \implies \Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *MGT-lemma:*

**assumes** *MGT-Calls*:  
 $\forall p \in \text{dom } \Gamma. \forall Z. \Gamma, \Theta \vdash_{t/F}$   
 $\{s. s=Z \wedge \Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } (-F)) \wedge$   
 $\Gamma \vdash (\text{Call } p) \downarrow \text{Normal } s\}$   
 $(\text{Call } p)$   
 $\{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$   
 $\{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$   
**shows**  $\bigwedge Z. \Gamma, \Theta \vdash_{t/F} \{s. s=Z \wedge \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } (-F)) \wedge$   
 $\Gamma \vdash c \downarrow \text{Normal } s\}$   
 $c$   
 $\{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\}, \{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$   
 $\langle \text{proof} \rangle$

**lemma** *Call-lemma':*

**assumes** *Call-hyp*:  
 $\forall q \in \text{dom } \Gamma. \forall Z. \Gamma, \Theta \vdash_{t/F} \{s. s=Z \wedge \Gamma \vdash \langle \text{Call } q, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } (-F)) \wedge$   
 $(-F)) \wedge$   
 $\Gamma \vdash \text{Call } q \downarrow \text{Normal } s \wedge ((s, q), (\sigma, p)) \in \text{termi-call-steps } \Gamma\}$   
 $(\text{Call } q)$   
 $\{t. \Gamma \vdash \langle \text{Call } q, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$   
 $\{t. \Gamma \vdash \langle \text{Call } q, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$   
**shows**  $\bigwedge Z. \Gamma, \Theta \vdash_{t/F}$   
 $\{s. s=Z \wedge \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } (-F)) \wedge \Gamma \vdash \text{Call } p \downarrow \text{Normal } \sigma \wedge$   
 $(\exists c'. \Gamma \vdash (\text{Call } p, \text{Normal } \sigma) \rightarrow^+ (c', \text{Normal } s) \wedge c \in \text{redexes } c')\}$   
 $c$   
 $\{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$   
 $\{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$   
 $\langle \text{proof} \rangle$

To prove a procedure implementation correct it suffices to assume only the procedure specifications of procedures that actually occur during evaluation of the body.

**lemma** *Call-lemma*:

**assumes**  $A$ :  
 $\forall q \in \text{dom } \Gamma. \forall Z. \Gamma, \Theta \vdash_t / F$   
 $\{s. s=Z \wedge \Gamma \vdash \langle \text{Call } q, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } '(-F)) \wedge$   
 $\Gamma \vdash \text{Call } q \downarrow \text{Normal } s \wedge ((s, q), (\sigma, p)) \in \text{termi-call-steps } \Gamma\}$   
 $(\text{Call } q)$   
 $\{t. \Gamma \vdash \langle \text{Call } q, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$   
 $\{t. \Gamma \vdash \langle \text{Call } q, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$   
**assumes**  $p\text{def}$ :  $p \in \text{dom } \Gamma$   
**shows**  $\bigwedge Z. \Gamma, \Theta \vdash_t / F$   
 $(\{\sigma\} \cap \{s. s=Z \wedge \Gamma \vdash \langle \text{the } (\Gamma \text{ } p), \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } '(-F))$   
 $\wedge$   
 $\Gamma \vdash \text{the } (\Gamma \text{ } p) \downarrow \text{Normal } s\})$   
 $\text{the } (\Gamma \text{ } p)$   
 $\{t. \Gamma \vdash \langle \text{the } (\Gamma \text{ } p), \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$   
 $\{t. \Gamma \vdash \langle \text{the } (\Gamma \text{ } p), \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$   
 $\langle \text{proof} \rangle$

**lemma** *Call-lemma-switch-Call-body*:

**assumes**  
 $\text{call}$ :  $\forall q \in \text{dom } \Gamma. \forall Z. \Gamma, \Theta \vdash_t / F$   
 $\{s. s=Z \wedge \Gamma \vdash \langle \text{Call } q, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } '(-F)) \wedge$   
 $\Gamma \vdash \text{Call } q \downarrow \text{Normal } s \wedge ((s, q), (\sigma, p)) \in \text{termi-call-steps } \Gamma\}$   
 $(\text{Call } q)$   
 $\{t. \Gamma \vdash \langle \text{Call } q, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$   
 $\{t. \Gamma \vdash \langle \text{Call } q, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$   
**assumes**  $p\text{-defined}$ :  $p \in \text{dom } \Gamma$   
**shows**  $\bigwedge Z. \Gamma, \Theta \vdash_t / F$   
 $(\{\sigma\} \cap \{s. s=Z \wedge \Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } '(-F)) \wedge$   
 $\Gamma \vdash \text{Call } p \downarrow \text{Normal } s\})$   
 $\text{the } (\Gamma \text{ } p)$   
 $\{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$   
 $\{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$   
 $\langle \text{proof} \rangle$

**lemma** *MGT-Call*:

$\forall p \in \text{dom } \Gamma. \forall Z.$   
 $\Gamma, \Theta \vdash_t / F \{s. s=Z \wedge \Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } '(-F)) \wedge$   
 $\Gamma \vdash (\text{Call } p) \downarrow \text{Normal } s\}$   
 $(\text{Call } p)$   
 $\{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$   
 $\{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$   
 $\langle \text{proof} \rangle$

**lemma** *CollInt-iff*:  $\{s. P s\} \cap \{s. Q s\} = \{s. P s \wedge Q s\}$   
 ⟨proof⟩

**lemma** *image-Un-conv*:  $f \cdot (\bigcup_{p \in \text{dom } \Gamma} \bigcup Z. \{x p Z\}) = (\bigcup_{p \in \text{dom } \Gamma} \bigcup Z. \{f(x p Z)\})$   
 ⟨proof⟩

Another proof of *MGT-Call*, maybe a little more readable

**lemma**

$\forall p \in \text{dom } \Gamma. \forall Z.$

$\Gamma, \{\} \vdash_{t/F} \{s. s=Z \wedge \Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle \Rightarrow \neg(\{\text{Stuck}\} \cup \text{Fault } \cdot (-F)) \wedge$   
 $\Gamma \vdash \langle \text{Call } p \rangle \downarrow \text{Normal } s\}$   
 $(\text{Call } p)$   
 $\{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$   
 $\{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$

⟨proof⟩

**theorem** *hoaret-complete*:  $\Gamma \models_{t/F} P c Q, A \implies \Gamma, \{\} \vdash_{t/F} P c Q, A$   
 ⟨proof⟩

**lemma** *hoaret-complete'*:

**assumes** *valid*:  $\Gamma, \Theta \models_{t/F} P c Q, A$

**shows**  $\Gamma, \Theta \vdash_{t/F} P c Q, A$

⟨proof⟩

## 10.3 And Now: Some Useful Rules

### 10.3.1 Modify Return

**lemma** *Proc-exnModifyReturn-sound*:

**assumes** *valid-call*:  $\Gamma, \Theta \models_{t/F} P \text{ call-exn } \text{init } p \text{ return}' \text{ result-exn } c Q, A$

**assumes** *valid-modif*:

$\forall \sigma. \Gamma, \Theta \models_{UNIV} \{\sigma\} (\text{Call } p) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$

**assumes** *res-modif*:

$\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow \text{return}' s t = \text{return } s t$

**assumes** *ret-modifAbr*:

$\forall s t. t \in \text{ModifAbr } (\text{init } s) \longrightarrow \text{result-exn } (\text{return}' s t) t = \text{result-exn } (\text{return } s t) t$

**shows**  $\Gamma, \Theta \models_{t/F} P (\text{call-exn } \text{init } p \text{ return } \text{result-exn } c) Q, A$

⟨proof⟩

**lemma** *ProcModifyReturn-sound*:

**assumes** *valid-call*:  $\Gamma, \Theta \models_{t/F} P \text{ call } \text{init } p \text{ return}' c Q, A$

**assumes** *valid-modif*:

$\forall \sigma. \Gamma, \Theta \models_{UNIV} \{\sigma\} (\text{Call } p) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$

**assumes** *res-modif*:

$\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow \text{return}' s t = \text{return } s t$

**assumes** *ret-modifAbr*:  
 $\forall s t. t \in \text{ModifAbr } (\text{init } s) \longrightarrow \text{return}' s t = \text{return } s t$   
**shows**  $\Gamma, \Theta \models_{t/F} P (\text{call init } p \text{ return } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *Proc-exnModifyReturn*:

**assumes** *spec*:  $\Gamma, \Theta \vdash_{t/F} P (\text{call-exn init } p \text{ return}' \text{ result-exn } c) Q, A$   
**assumes** *res-modif*:  
 $\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow (\text{return}' s t) = (\text{return } s t)$   
**assumes** *ret-modifAbr*:  
 $\forall s t. t \in \text{ModifAbr } (\text{init } s) \longrightarrow (\text{result-exn } (\text{return}' s t) t) = (\text{result-exn } (\text{return } s t) t)$   
**assumes** *modifies-spec*:  
 $\forall \sigma. \Gamma, \Theta \vdash_{UNIV} \{\sigma\} (\text{Call } p) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{call-exn init } p \text{ return result-exn } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *ProcModifyReturn*:

**assumes** *spec*:  $\Gamma, \Theta \vdash_{t/F} P (\text{call init } p \text{ return}' c) Q, A$   
**assumes** *res-modif*:  
 $\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow (\text{return}' s t) = (\text{return } s t)$   
**assumes** *ret-modifAbr*:  
 $\forall s t. t \in \text{ModifAbr } (\text{init } s) \longrightarrow (\text{return}' s t) = (\text{return } s t)$   
**assumes** *modifies-spec*:  
 $\forall \sigma. \Gamma, \Theta \vdash_{UNIV} \{\sigma\} (\text{Call } p) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{call init } p \text{ return } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *Proc-exnModifyReturnSameFaults-sound*:

**assumes** *valid-call*:  $\Gamma, \Theta \models_{t/F} P \text{ call-exn init } p \text{ return}' \text{ result-exn } c Q, A$   
**assumes** *valid-modif*:  
 $\forall \sigma. \Gamma, \Theta \models_{/F} \{\sigma\} \text{ Call } p (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**assumes** *res-modif*:  
 $\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow \text{return}' s t = \text{return } s t$   
**assumes** *ret-modifAbr*:  
 $\forall s t. t \in \text{ModifAbr } (\text{init } s) \longrightarrow \text{result-exn } (\text{return}' s t) t = \text{result-exn } (\text{return } s t) t$   
**shows**  $\Gamma, \Theta \models_{t/F} P (\text{call-exn init } p \text{ return result-exn } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *ProcModifyReturnSameFaults-sound*:

**assumes** *valid-call*:  $\Gamma, \Theta \models_{t/F} P \text{ call init } p \text{ return}' c Q, A$   
**assumes** *valid-modif*:  
 $\forall \sigma. \Gamma, \Theta \models_{/F} \{\sigma\} \text{ Call } p (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**assumes** *res-modif*:  
 $\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow \text{return}' s t = \text{return } s t$

**assumes** *ret-modifAbr*:  
 $\forall s t. t \in \text{ModifAbr } (\text{init } s) \longrightarrow \text{return}' s t = \text{return } s t$   
**shows**  $\Gamma, \Theta \Vdash_{t/F} P (\text{call init } p \text{ return } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *Proc-exnModifyReturnSameFaults*:  
**assumes** *spec*:  $\Gamma, \Theta \Vdash_{t/F} P (\text{call-exn init } p \text{ return}' \text{ result-exn } c) Q, A$   
**assumes** *res-modif*:  
 $\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow (\text{return}' s t) = (\text{return } s t)$   
**assumes** *ret-modifAbr*:  
 $\forall s t. t \in \text{ModifAbr } (\text{init } s) \longrightarrow \text{result-exn } (\text{return}' s t) t = \text{result-exn } (\text{return } s t) t$   
**assumes** *modifies-spec*:  
 $\forall \sigma. \Gamma, \Theta \Vdash_{/F} \{\sigma\} (\text{Call } p) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\Gamma, \Theta \Vdash_{t/F} P (\text{call-exn init } p \text{ return result-exn } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *ProcModifyReturnSameFaults*:  
**assumes** *spec*:  $\Gamma, \Theta \Vdash_{t/F} P (\text{call init } p \text{ return}' c) Q, A$   
**assumes** *res-modif*:  
 $\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow (\text{return}' s t) = (\text{return } s t)$   
**assumes** *ret-modifAbr*:  
 $\forall s t. t \in \text{ModifAbr } (\text{init } s) \longrightarrow (\text{return}' s t) = (\text{return } s t)$   
**assumes** *modifies-spec*:  
 $\forall \sigma. \Gamma, \Theta \Vdash_{/F} \{\sigma\} (\text{Call } p) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\Gamma, \Theta \Vdash_{t/F} P (\text{call init } p \text{ return } c) Q, A$   
 $\langle \text{proof} \rangle$

### 10.3.2 DynCall

**lemma** *dynProc-exnModifyReturn-sound*:  
**assumes** *valid-call*:  $\Gamma, \Theta \Vdash_{t/F} P \text{ dynCall-exn } f g \text{ init } p \text{ return}' \text{ result-exn } c Q, A$   
**assumes** *valid-modif*:  
 $\forall s \in P. \forall \sigma. \Gamma, \Theta \Vdash_{/UNIV} \{\sigma\} (\text{Call } (p s)) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**assumes** *ret-modif*:  
 $\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow \text{return}' s t = \text{return } s t$   
**assumes** *ret-modifAbr*:  $\forall s t. t \in \text{ModifAbr } (\text{init } s) \longrightarrow \text{result-exn } (\text{return}' s t) t = \text{result-exn } (\text{return } s t) t$   
**shows**  $\Gamma, \Theta \Vdash_{t/F} P (\text{dynCall-exn } f g \text{ init } p \text{ return result-exn } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *dynProcModifyReturn-sound*:  
**assumes** *valid-call*:  $\Gamma, \Theta \Vdash_{t/F} P \text{ dynCall init } p \text{ return}' c Q, A$   
**assumes** *valid-modif*:  
 $\forall s \in P. \forall \sigma. \Gamma, \Theta \Vdash_{/UNIV} \{\sigma\} (\text{Call } (p s)) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$   
**assumes** *ret-modif*:  
 $\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow \text{return}' s t = \text{return } s t$

**assumes** *ret-modifAbr*:  $\forall s t. t \in \text{ModifAbr } (\text{init } s) \longrightarrow \text{return}' s t = \text{return } s t$   
**shows**  $\Gamma, \Theta \models_{t/F} P (\text{dynCall } \text{init } p \text{ return } c) Q, A$   
 ⟨*proof*⟩

**lemma** *dynProc-exnModifyReturn*:

**assumes** *dyn-call*:  $\Gamma, \Theta \vdash_{t/F} P \text{ dynCall-exn } f g \text{ init } p \text{ return}' \text{ result-exn } c Q, A$

**assumes** *ret-modif*:

$\forall s t. t \in \text{Modif } (\text{init } s)$   
 $\longrightarrow \text{return}' s t = \text{return } s t$

**assumes** *ret-modifAbr*:  $\forall s t. t \in \text{ModifAbr } (\text{init } s)$

$\longrightarrow \text{result-exn } (\text{return}' s t) t = \text{result-exn } (\text{return } s t) t$

**assumes** *modif*:

$\forall s \in P. \forall \sigma.$

$\Gamma, \Theta \vdash_{UNIV} \{\sigma\} \text{ Call } (p s) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$

**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{dynCall-exn } f g \text{ init } p \text{ return } \text{result-exn } c) Q, A$

⟨*proof*⟩

**lemma** *dynProcModifyReturn*:

**assumes** *dyn-call*:  $\Gamma, \Theta \vdash_{t/F} P \text{ dynCall } \text{init } p \text{ return}' c Q, A$

**assumes** *ret-modif*:

$\forall s t. t \in \text{Modif } (\text{init } s)$   
 $\longrightarrow \text{return}' s t = \text{return } s t$

**assumes** *ret-modifAbr*:  $\forall s t. t \in \text{ModifAbr } (\text{init } s)$

$\longrightarrow \text{return}' s t = \text{return } s t$

**assumes** *modif*:

$\forall s \in P. \forall \sigma.$

$\Gamma, \Theta \vdash_{UNIV} \{\sigma\} \text{ Call } (p s) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$

**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{dynCall } \text{init } p \text{ return } c) Q, A$

⟨*proof*⟩

**lemma** *dynProc-exnModifyReturnSameFaults-sound*:

**assumes** *valid-call*:  $\Gamma, \Theta \models_{t/F} P \text{ dynCall-exn } f g \text{ init } p \text{ return}' \text{ result-exn } c Q, A$

**assumes** *valid-modif*:

$\forall s \in P. \forall \sigma. \Gamma, \Theta \models_{t/F} \{\sigma\} \text{ Call } (p s) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$

**assumes** *ret-modif*:

$\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow \text{return}' s t = \text{return } s t$

**assumes** *ret-modifAbr*:  $\forall s t. t \in \text{ModifAbr } (\text{init } s) \longrightarrow \text{result-exn } (\text{return}' s t) t$   
 $= \text{result-exn } (\text{return } s t) t$

**shows**  $\Gamma, \Theta \models_{t/F} P (\text{dynCall-exn } f g \text{ init } p \text{ return } \text{result-exn } c) Q, A$

⟨*proof*⟩

**lemma** *dynProcModifyReturnSameFaults-sound*:

**assumes** *valid-call*:  $\Gamma, \Theta \models_{t/F} P \text{ dynCall } \text{init } p \text{ return}' c Q, A$

**assumes** *valid-modif*:

$\forall s \in P. \forall \sigma. \Gamma, \Theta \models_{t/F} \{\sigma\} \text{ Call } (p s) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$

**assumes** *ret-modif*:

$\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow \text{return}' s t = \text{return } s t$   
**assumes** *ret-modifAbr*:  $\forall s t. t \in \text{ModifAbr } (\text{init } s) \longrightarrow \text{return}' s t = \text{return } s t$   
**shows**  $\Gamma, \Theta \models_{t/F} P (\text{dynCall } \text{init } p \text{ return } c) Q, A$   
 ⟨*proof*⟩

**lemma** *dynProc-exnModifyReturnSameFaults*:

**assumes** *dyn-call*:  $\Gamma, \Theta \vdash_{t/F} P \text{ dynCall-exn } f g \text{ init } p \text{ return}' \text{ result-exn } c Q, A$

**assumes** *ret-modif*:

$\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow \text{return}' s t = \text{return } s t$

**assumes** *ret-modifAbr*:  $\forall s t. t \in \text{ModifAbr } (\text{init } s) \longrightarrow \text{result-exn } (\text{return}' s t) t = \text{result-exn } (\text{return } s t) t$

**assumes** *modif*:

$\forall s \in P. \forall \sigma. \Gamma, \Theta \vdash_{t/F} \{\sigma\} \text{ Call } (p s) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$

**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{dynCall-exn } f g \text{ init } p \text{ return } \text{result-exn } c) Q, A$

⟨*proof*⟩

**lemma** *dynProcModifyReturnSameFaults*:

**assumes** *dyn-call*:  $\Gamma, \Theta \vdash_{t/F} P \text{ dynCall } \text{init } p \text{ return}' c Q, A$

**assumes** *ret-modif*:

$\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow \text{return}' s t = \text{return } s t$

**assumes** *ret-modifAbr*:  $\forall s t. t \in \text{ModifAbr } (\text{init } s) \longrightarrow \text{return}' s t = \text{return } s t$

**assumes** *modif*:

$\forall s \in P. \forall \sigma. \Gamma, \Theta \vdash_{t/F} \{\sigma\} \text{ Call } (p s) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$

**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{dynCall } \text{init } p \text{ return } c) Q, A$

⟨*proof*⟩

### 10.3.3 Conjunction of Postcondition

**lemma** *PostConjI-sound*:

**assumes** *valid-Q*:  $\Gamma, \Theta \models_{t/F} P c Q, A$

**assumes** *valid-R*:  $\Gamma, \Theta \models_{t/F} P c R, B$

**shows**  $\Gamma, \Theta \models_{t/F} P c (Q \cap R), (A \cap B)$

⟨*proof*⟩

**lemma** *PostConjI*:

**assumes** *deriv-Q*:  $\Gamma, \Theta \vdash_{t/F} P c Q, A$

**assumes** *deriv-R*:  $\Gamma, \Theta \vdash_{t/F} P c R, B$

**shows**  $\Gamma, \Theta \vdash_{t/F} P c (Q \cap R), (A \cap B)$

⟨*proof*⟩

**lemma** *Merge-PostConj-sound*:

**assumes** *validF*:  $\Gamma, \Theta \models_{t/F} P c Q, A$

**assumes** *validG*:  $\Gamma, \Theta \models_{t/G} P' c R, X$

**assumes** *F-G*:  $F \subseteq G$

**assumes** *P-P'*:  $P \subseteq P'$

**shows**  $\Gamma, \Theta \models_{t/F} P c (Q \cap R), (A \cap X)$

$\langle proof \rangle$

**lemma** *Merge-PostConj*:

**assumes** *validF*:  $\Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$   
**assumes** *validG*:  $\Gamma, \Theta \vdash_{t/G} P' \ c \ R, X$   
**assumes** *F-G*:  $F \subseteq G$   
**assumes** *P-P'*:  $P \subseteq P'$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P \ c \ (Q \cap R), (A \cap X)$

$\langle proof \rangle$

### 10.3.4 Guards and Guarantees

**lemma** *SplitGuards-sound*:

**assumes** *valid-c1*:  $\Gamma, \Theta \models_{t/F} P \ c_1 \ Q, A$   
**assumes** *valid-c2*:  $\Gamma, \Theta \models_{t/F} P \ c_2 \ UNIV, UNIV$   
**assumes** *c*:  $(c_1 \cap_g c_2) = Some \ c$   
**shows**  $\Gamma, \Theta \models_{t/F} P \ c \ Q, A$

$\langle proof \rangle$

**lemma** *SplitGuards*:

**assumes** *c*:  $(c_1 \cap_g c_2) = Some \ c$   
**assumes** *deriv-c1*:  $\Gamma, \Theta \vdash_{t/F} P \ c_1 \ Q, A$   
**assumes** *deriv-c2*:  $\Gamma, \Theta \vdash_{t/F} P \ c_2 \ UNIV, UNIV$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$

$\langle proof \rangle$

**lemma** *CombineStrip-sound*:

**assumes** *valid*:  $\Gamma, \Theta \models_{t/F} P \ c \ Q, A$   
**assumes** *valid-strip*:  $\Gamma, \Theta \models_{t/\{\}} P \ (strip-guards \ (-F) \ c) \ UNIV, UNIV$   
**shows**  $\Gamma, \Theta \models_{t/\{\}} P \ c \ Q, A$

$\langle proof \rangle$

**lemma** *CombineStrip*:

**assumes** *deriv*:  $\Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$   
**assumes** *deriv-strip*:  $\Gamma, \Theta \vdash_{t/\{\}} P \ (strip-guards \ (-F) \ c) \ UNIV, UNIV$   
**shows**  $\Gamma, \Theta \vdash_{t/\{\}} P \ c \ Q, A$

$\langle proof \rangle$

**lemma** *GuardsFlip-sound*:

**assumes** *valid*:  $\Gamma, \Theta \models_{t/F} P \ c \ Q, A$   
**assumes** *validFlip*:  $\Gamma, \Theta \models_{t/-F} P \ c \ UNIV, UNIV$   
**shows**  $\Gamma, \Theta \models_{t/\{\}} P \ c \ Q, A$

$\langle proof \rangle$

**lemma** *GuardsFlip*:  
**assumes** *deriv*:  $\Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A$   
**assumes** *derivFlip*:  $\Gamma, \Theta \vdash_{-F} P \text{ c } UNIV, UNIV$   
**shows**  $\Gamma, \Theta \vdash_{t/\{\}} P \text{ c } Q, A$   
 $\langle proof \rangle$

**lemma** *MarkGuardsI-sound*:  
**assumes** *valid*:  $\Gamma, \Theta \models_{t/\{\}} P \text{ c } Q, A$   
**shows**  $\Gamma, \Theta \models_{t/\{\}} P \text{ mark-guards } f \text{ c } Q, A$   
 $\langle proof \rangle$

**lemma** *MarkGuardsI*:  
**assumes** *deriv*:  $\Gamma, \Theta \vdash_{t/\{\}} P \text{ c } Q, A$   
**shows**  $\Gamma, \Theta \vdash_{t/\{\}} P \text{ mark-guards } f \text{ c } Q, A$   
 $\langle proof \rangle$

**lemma** *MarkGuardsD-sound*:  
**assumes** *valid*:  $\Gamma, \Theta \models_{t/\{\}} P \text{ mark-guards } f \text{ c } Q, A$   
**shows**  $\Gamma, \Theta \models_{t/\{\}} P \text{ c } Q, A$   
 $\langle proof \rangle$

**lemma** *MarkGuardsD*:  
**assumes** *deriv*:  $\Gamma, \Theta \vdash_{t/\{\}} P \text{ mark-guards } f \text{ c } Q, A$   
**shows**  $\Gamma, \Theta \vdash_{t/\{\}} P \text{ c } Q, A$   
 $\langle proof \rangle$

**lemma** *MergeGuardsI-sound*:  
**assumes** *valid*:  $\Gamma, \Theta \models_{t/F} P \text{ c } Q, A$   
**shows**  $\Gamma, \Theta \models_{t/F} P \text{ merge-guards } c \text{ c } Q, A$   
 $\langle proof \rangle$

**lemma** *MergeGuardsI*:  
**assumes** *deriv*:  $\Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ merge-guards } c \text{ c } Q, A$   
 $\langle proof \rangle$

**lemma** *MergeGuardsD-sound*:  
**assumes** *valid*:  $\Gamma, \Theta \models_{t/F} P \text{ merge-guards } c \text{ c } Q, A$   
**shows**  $\Gamma, \Theta \models_{t/F} P \text{ c } Q, A$   
 $\langle proof \rangle$

**lemma** *MergeGuardsD*:  
**assumes** *deriv*:  $\Gamma, \Theta \vdash_{t/F} P \text{ merge-guards } c \text{ c } Q, A$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A$   
 $\langle proof \rangle$

**lemma** *SubsetGuards-sound*:

**assumes**  $c-c'$ :  $c \subseteq_g c'$   
**assumes** *valid*:  $\Gamma, \Theta \models_{t/\{\}} P \ c' \ Q, A$   
**shows**  $\Gamma, \Theta \models_{t/\{\}} P \ c \ Q, A$

*<proof>*

**lemma** *SubsetGuards*:

**assumes**  $c-c'$ :  $c \subseteq_g c'$   
**assumes** *deriv*:  $\Gamma, \Theta \vdash_{t/\{\}} P \ c' \ Q, A$   
**shows**  $\Gamma, \Theta \vdash_{t/\{\}} P \ c \ Q, A$

*<proof>*

**lemma** *NormalizeD-sound*:

**assumes** *valid*:  $\Gamma, \Theta \models_{t/F} P \ (\text{normalize } c) \ Q, A$   
**shows**  $\Gamma, \Theta \models_{t/F} P \ c \ Q, A$

*<proof>*

**lemma** *NormalizeD*:

**assumes** *deriv*:  $\Gamma, \Theta \vdash_{t/F} P \ (\text{normalize } c) \ Q, A$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$

*<proof>*

**lemma** *NormalizeI-sound*:

**assumes** *valid*:  $\Gamma, \Theta \models_{t/F} P \ c \ Q, A$   
**shows**  $\Gamma, \Theta \models_{t/F} P \ (\text{normalize } c) \ Q, A$

*<proof>*

**lemma** *NormalizeI*:

**assumes** *deriv*:  $\Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P \ (\text{normalize } c) \ Q, A$

*<proof>*

### 10.3.5 Restricting the Procedure Environment

**lemma** *validt-restrict-to-validt*:

**assumes** *validt-c*:  $\Gamma|_M \models_{t/F} P \ c \ Q, A$   
**shows**  $\Gamma \models_{t/F} P \ c \ Q, A$

*<proof>*

**lemma** *augment-procs*:

**assumes** *deriv-c*:  $\Gamma|_M, \{\} \vdash_{t/F} P \ c \ Q, A$   
**shows**  $\Gamma, \{\} \vdash_{t/F} P \ c \ Q, A$

*<proof>*

### 10.3.6 Miscellaneous

**lemma** *augment-Faults*:

**assumes** *deriv-c*:  $\Gamma, \{\} \vdash_{t/F} P \text{ c } Q, A$

**assumes** *F*:  $F \subseteq F'$

**shows**  $\Gamma, \{\} \vdash_{t/F'} P \text{ c } Q, A$

*<proof>*

**lemma** *TerminationPartial-sound*:

**assumes** *termination*:  $\forall s \in P. \Gamma \vdash c \downarrow \text{Normal } s$

**assumes** *partial-corr*:  $\Gamma, \Theta \models_{/F} P \text{ c } Q, A$

**shows**  $\Gamma, \Theta \models_{t/F} P \text{ c } Q, A$

*<proof>*

**lemma** *TerminationPartial*:

**assumes** *partial-deriv*:  $\Gamma, \Theta \vdash_{/F} P \text{ c } Q, A$

**assumes** *termination*:  $\forall s \in P. \Gamma \vdash c \downarrow \text{Normal } s$

**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A$

*<proof>*

**lemma** *TerminationPartialStrip*:

**assumes** *partial-deriv*:  $\Gamma, \Theta \vdash_{/F} P \text{ c } Q, A$

**assumes** *termination*:  $\forall s \in P. \text{strip } F' \Gamma \vdash \text{strip-guards } F' \text{ c } \downarrow \text{Normal } s$

**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A$

*<proof>*

**lemma** *SplitTotalPartial*:

**assumes** *termi*:  $\Gamma, \Theta \vdash_{t/F} P \text{ c } Q', A'$

**assumes** *part*:  $\Gamma, \Theta \vdash_{/F} P \text{ c } Q, A$

**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A$

*<proof>*

**lemma** *SplitTotalPartial'*:

**assumes** *termi*:  $\Gamma, \Theta \vdash_{t/UNIV} P \text{ c } Q', A'$

**assumes** *part*:  $\Gamma, \Theta \vdash_{/F} P \text{ c } Q, A$

**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A$

*<proof>*

**end**

## 11 Derived Hoare Rules for Total Correctness

**theory** *HoareTotal* **imports** *HoareTotalProps* **begin**

**lemma** *conseq-no-aux*:

$[[\Gamma, \Theta \vdash_{t/F} P' \text{ c } Q', A';$

$\forall s. s \in P \longrightarrow (s \in P' \wedge (Q' \subseteq Q) \wedge (A' \subseteq A))]]$

$$\begin{aligned} &\Rightarrow \\ &\Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A \\ &\langle \text{proof} \rangle \end{aligned}$$

If for example a specification for a "procedure pointer" parameter is in the precondition we can extract it with this rule

$$\begin{aligned} \text{lemma } \textit{conseq-exploit-pre}: \\ &[\forall s \in P. \Gamma, \Theta \vdash_{t/F} (\{s\} \cap P) \text{ c } Q, A] \\ &\Rightarrow \\ &\Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A \\ &\langle \text{proof} \rangle \end{aligned}$$

$$\begin{aligned} \text{lemma } \textit{conseq}: [\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{ c } (Q' Z), (A' Z); \\ &\forall s. s \in P \longrightarrow (\exists Z. s \in P' Z \wedge (Q' Z \subseteq Q) \wedge (A' Z \subseteq A))] \\ &\Rightarrow \\ &\Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A \\ &\langle \text{proof} \rangle \end{aligned}$$

$$\begin{aligned} \text{lemma } \textit{Lem}: [\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{ c } (Q' Z), (A' Z); \\ &P \subseteq \{s. \exists Z. s \in P' Z \wedge (Q' Z \subseteq Q) \wedge (A' Z \subseteq A)\}] \\ &\Rightarrow \\ &\Gamma, \Theta \vdash_{t/F} P \text{ (lem } x \text{ c) } Q, A \\ &\langle \text{proof} \rangle \end{aligned}$$

$$\begin{aligned} \text{lemma } \textit{LemAnno}: \\ \text{assumes } \textit{conseq}: P \subseteq \{s. \exists Z. s \in P' Z \wedge \\ &(\forall t. t \in Q' Z \longrightarrow t \in Q) \wedge (\forall t. t \in A' Z \longrightarrow t \in A)\} \\ \text{assumes } \textit{lem}: \forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{ c } (Q' Z), (A' Z) \\ \text{shows } \Gamma, \Theta \vdash_{t/F} P \text{ (lem } x \text{ c) } Q, A \\ &\langle \text{proof} \rangle \end{aligned}$$

$$\begin{aligned} \text{lemma } \textit{LemAnnoNoAbrupt}: \\ \text{assumes } \textit{conseq}: P \subseteq \{s. \exists Z. s \in P' Z \wedge (\forall t. t \in Q' Z \longrightarrow t \in Q)\} \\ \text{assumes } \textit{lem}: \forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{ c } (Q' Z), \{\} \\ \text{shows } \Gamma, \Theta \vdash_{t/F} P \text{ (lem } x \text{ c) } Q, \{\} \\ &\langle \text{proof} \rangle \end{aligned}$$

$$\begin{aligned} \text{lemma } \textit{TrivPost}: \forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{ c } (Q' Z), (A' Z) \\ &\Rightarrow \\ &\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{ c } UNIV, UNIV \\ &\langle \text{proof} \rangle \end{aligned}$$

$$\begin{aligned} \text{lemma } \textit{TrivPostNoAbr}: \forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{ c } (Q' Z), \{\} \\ &\Rightarrow \end{aligned}$$

$\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) c \text{ UNIV}, \{\}$

$\langle \text{proof} \rangle$

**lemma** *DynComConseq*:

**assumes**  $P \subseteq \{s. \exists P' Q' A'. \Gamma, \Theta \vdash_{t/F} P' (c s) Q', A' \wedge P \subseteq P' \wedge Q' \subseteq Q \wedge A' \subseteq A\}$

**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ DynCom } c Q, A$

$\langle \text{proof} \rangle$

**lemma** *SpecAnno*:

**assumes** *consequence*:  $P \subseteq \{s. (\exists Z. s \in P' Z \wedge (Q' Z \subseteq Q) \wedge (A' Z \subseteq A))\}$

**assumes** *spec*:  $\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) (c Z) (Q' Z), (A' Z)$

**assumes** *bdy-constant*:  $\forall Z. c Z = c \text{ undefined}$

**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{specAnno } P' c Q' A') Q, A$

$\langle \text{proof} \rangle$

**lemma** *SpecAnno'*:

$\llbracket P \subseteq \{s. \exists Z. s \in P' Z \wedge (\forall t. t \in Q' Z \longrightarrow t \in Q) \wedge (\forall t. t \in A' Z \longrightarrow t \in A)\};$

$\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) (c Z) (Q' Z), (A' Z);$

$\forall Z. c Z = c \text{ undefined}$

$\rrbracket \Longrightarrow$

$\Gamma, \Theta \vdash_{t/F} P (\text{specAnno } P' c Q' A') Q, A$

$\langle \text{proof} \rangle$

**lemma** *SpecAnnoNoAbrupt*:

$\llbracket P \subseteq \{s. \exists Z. s \in P' Z \wedge (\forall t. t \in Q' Z \longrightarrow t \in Q)\};$

$\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) (c Z) (Q' Z), \{\};$

$\forall Z. c Z = c \text{ undefined}$

$\rrbracket \Longrightarrow$

$\Gamma, \Theta \vdash_{t/F} P (\text{specAnno } P' c Q' (\lambda s. \{\})) Q, A$

$\langle \text{proof} \rangle$

**lemma** *Skip*:  $P \subseteq Q \Longrightarrow \Gamma, \Theta \vdash_{t/F} P \text{ Skip } Q, A$

$\langle \text{proof} \rangle$

**lemma** *Basic*:  $P \subseteq \{s. (f s) \in Q\} \Longrightarrow \Gamma, \Theta \vdash_{t/F} P (\text{Basic } f) Q, A$

$\langle \text{proof} \rangle$

**lemma** *BasicCond*:

$\llbracket P \subseteq \{s. (b s \longrightarrow f s \in Q) \wedge (\neg b s \longrightarrow g s \in Q)\} \rrbracket \Longrightarrow$

$\Gamma, \Theta \vdash_{t/F} P \text{ Basic } (\lambda s. \text{if } b s \text{ then } f s \text{ else } g s) Q, A$

$\langle \text{proof} \rangle$

**lemma** *Spec*:  $P \subseteq \{s. (\forall t. (s,t) \in r \longrightarrow t \in Q) \wedge (\exists t. (s,t) \in r)\}$   
 $\implies \Gamma, \Theta \vdash_{t/F} P \text{ (Spec } r) \ Q, A$   
 ⟨proof⟩

**lemma** *SpecIf*:  
 $\llbracket P \subseteq \{s. (b \ s \longrightarrow f \ s \in Q) \wedge (\neg b \ s \longrightarrow g \ s \in Q \wedge h \ s \in Q)\} \rrbracket \implies$   
 $\Gamma, \Theta \vdash_{t/F} P \text{ Spec (if-rel } b \ f \ g \ h) \ Q, A$   
 ⟨proof⟩

**lemma** *Seq* [*trans, intro?*]:  
 $\llbracket \Gamma, \Theta \vdash_{t/F} P \ c_1 \ R, A; \Gamma, \Theta \vdash_{t/F} R \ c_2 \ Q, A \rrbracket \implies \Gamma, \Theta \vdash_{t/F} P \text{ Seq } c_1 \ c_2 \ Q, A$   
 ⟨proof⟩

**lemma** *SeqSwap*:  
 $\llbracket \Gamma, \Theta \vdash_{t/F} R \ c_2 \ Q, A; \Gamma, \Theta \vdash_{t/F} P \ c_1 \ R, A \rrbracket \implies \Gamma, \Theta \vdash_{t/F} P \text{ Seq } c_1 \ c_2 \ Q, A$   
 ⟨proof⟩

**lemma** *BSeq*:  
 $\llbracket \Gamma, \Theta \vdash_{t/F} P \ c_1 \ R, A; \Gamma, \Theta \vdash_{t/F} R \ c_2 \ Q, A \rrbracket \implies \Gamma, \Theta \vdash_{t/F} P \text{ (bseq } c_1 \ c_2) \ Q, A$   
 ⟨proof⟩

**lemma** *Cond*:  
**assumes** *wp*:  $P \subseteq \{s. (s \in b \longrightarrow s \in P_1) \wedge (s \notin b \longrightarrow s \in P_2)\}$   
**assumes** *deriv-c1*:  $\Gamma, \Theta \vdash_{t/F} P_1 \ c_1 \ Q, A$   
**assumes** *deriv-c2*:  $\Gamma, \Theta \vdash_{t/F} P_2 \ c_2 \ Q, A$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ (Cond } b \ c_1 \ c_2) \ Q, A$   
 ⟨proof⟩

**lemma** *CondSwap*:  
 $\llbracket \Gamma, \Theta \vdash_{t/F} P_1 \ c_1 \ Q, A; \Gamma, \Theta \vdash_{t/F} P_2 \ c_2 \ Q, A;$   
 $P \subseteq \{s. (s \in b \longrightarrow s \in P_1) \wedge (s \notin b \longrightarrow s \in P_2)\} \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash_{t/F} P \text{ (Cond } b \ c_1 \ c_2) \ Q, A$   
 ⟨proof⟩

**lemma** *Cond'*:  
 $\llbracket P \subseteq \{s. (b \subseteq P_1) \wedge (\neg b \subseteq P_2)\}; \Gamma, \Theta \vdash_{t/F} P_1 \ c_1 \ Q, A; \Gamma, \Theta \vdash_{t/F} P_2 \ c_2 \ Q, A \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash_{t/F} P \text{ (Cond } b \ c_1 \ c_2) \ Q, A$   
 ⟨proof⟩

**lemma** *CondInv*:  
**assumes** *wp*:  $P \subseteq Q$   
**assumes** *inv*:  $Q \subseteq \{s. (s \in b \longrightarrow s \in P_1) \wedge (s \notin b \longrightarrow s \in P_2)\}$   
**assumes** *deriv-c1*:  $\Gamma, \Theta \vdash_{t/F} P_1 \ c_1 \ Q, A$   
**assumes** *deriv-c2*:  $\Gamma, \Theta \vdash_{t/F} P_2 \ c_2 \ Q, A$

**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ (Cond } b \ c_1 \ c_2) \ Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *CondInv'*:

**assumes** *wp*:  $P \subseteq I$

**assumes** *inv*:  $I \subseteq \{s. (s \in b \longrightarrow s \in P_1) \wedge (s \notin b \longrightarrow s \in P_2)\}$

**assumes** *wp'*:  $I \subseteq Q$

**assumes** *deriv-c1*:  $\Gamma, \Theta \vdash_{t/F} P_1 \ c_1 \ I, A$

**assumes** *deriv-c2*:  $\Gamma, \Theta \vdash_{t/F} P_2 \ c_2 \ I, A$

**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ (Cond } b \ c_1 \ c_2) \ Q, A$

$\langle \text{proof} \rangle$

**lemma** *switchNil*:

$P \subseteq Q \Longrightarrow \Gamma, \Theta \vdash_{t/F} P \text{ (switch } v \ []) \ Q, A$

$\langle \text{proof} \rangle$

**lemma** *switchCons*:

$\llbracket P \subseteq \{s. (v \ s \in V \longrightarrow s \in P_1) \wedge (v \ s \notin V \longrightarrow s \in P_2)\};$

$\Gamma, \Theta \vdash_{t/F} P_1 \ c \ Q, A;$

$\Gamma, \Theta \vdash_{t/F} P_2 \text{ (switch } v \ vs) \ Q, A \rrbracket$

$\Longrightarrow \Gamma, \Theta \vdash_{t/F} P \text{ (switch } v \ ((V, c) \# vs)) \ Q, A$

$\langle \text{proof} \rangle$

**lemma** *Guard*:

$\llbracket P \subseteq g \cap R; \Gamma, \Theta \vdash_{t/F} R \ c \ Q, A \rrbracket$

$\Longrightarrow \Gamma, \Theta \vdash_{t/F} P \text{ Guard } f \ g \ c \ Q, A$

$\langle \text{proof} \rangle$

**lemma** *GuardSwap*:

$\llbracket \Gamma, \Theta \vdash_{t/F} R \ c \ Q, A; P \subseteq g \cap R \rrbracket$

$\Longrightarrow \Gamma, \Theta \vdash_{t/F} P \text{ Guard } f \ g \ c \ Q, A$

$\langle \text{proof} \rangle$

**lemma** *Guarantee*:

$\llbracket P \subseteq \{s. s \in g \longrightarrow s \in R\}; \Gamma, \Theta \vdash_{t/F} R \ c \ Q, A; f \in F \rrbracket$

$\Longrightarrow \Gamma, \Theta \vdash_{t/F} P \text{ (Guard } f \ g \ c) \ Q, A$

$\langle \text{proof} \rangle$

**lemma** *GuaranteeSwap*:

$\llbracket \Gamma, \Theta \vdash_{t/F} R \ c \ Q, A; P \subseteq \{s. s \in g \longrightarrow s \in R\}; f \in F \rrbracket$

$\Longrightarrow \Gamma, \Theta \vdash_{t/F} P \text{ (Guard } f \ g \ c) \ Q, A$

$\langle \text{proof} \rangle$

**lemma** *GuardStrip*:

$$\llbracket P \subseteq R; \Gamma, \Theta \vdash_{t/F} R \ c \ Q, A; f \in F \rrbracket$$

$$\implies \Gamma, \Theta \vdash_{t/F} P \ (\text{Guard } f \ g \ c) \ Q, A$$

$$\langle \text{proof} \rangle$$

**lemma** *GuardStripSwap*:  

$$\llbracket \Gamma, \Theta \vdash_{t/F} R \ c \ Q, A; P \subseteq R; f \in F \rrbracket$$

$$\implies \Gamma, \Theta \vdash_{t/F} P \ (\text{Guard } f \ g \ c) \ Q, A$$

$$\langle \text{proof} \rangle$$

**lemma** *GuaranteeStrip*:  

$$\llbracket P \subseteq R; \Gamma, \Theta \vdash_{t/F} R \ c \ Q, A; f \in F \rrbracket$$

$$\implies \Gamma, \Theta \vdash_{t/F} P \ (\text{guaranteeStrip } f \ g \ c) \ Q, A$$

$$\langle \text{proof} \rangle$$

**lemma** *GuaranteeStripSwap*:  

$$\llbracket \Gamma, \Theta \vdash_{t/F} R \ c \ Q, A; P \subseteq R; f \in F \rrbracket$$

$$\implies \Gamma, \Theta \vdash_{t/F} P \ (\text{guaranteeStrip } f \ g \ c) \ Q, A$$

$$\langle \text{proof} \rangle$$

**lemma** *GuaranteeAsGuard*:  

$$\llbracket P \subseteq g \cap R; \Gamma, \Theta \vdash_{t/F} R \ c \ Q, A \rrbracket$$

$$\implies \Gamma, \Theta \vdash_{t/F} P \ \text{guaranteeStrip } f \ g \ c \ Q, A$$

$$\langle \text{proof} \rangle$$

**lemma** *GuaranteeAsGuardSwap*:  

$$\llbracket \Gamma, \Theta \vdash_{t/F} R \ c \ Q, A; P \subseteq g \cap R \rrbracket$$

$$\implies \Gamma, \Theta \vdash_{t/F} P \ \text{guaranteeStrip } f \ g \ c \ Q, A$$

$$\langle \text{proof} \rangle$$

**lemma** *GuardsNil*:  

$$\Gamma, \Theta \vdash_{t/F} P \ c \ Q, A \implies$$

$$\Gamma, \Theta \vdash_{t/F} P \ (\text{guards } [] \ c) \ Q, A$$

$$\langle \text{proof} \rangle$$

**lemma** *GuardsCons*:  

$$\Gamma, \Theta \vdash_{t/F} P \ \text{Guard } f \ g \ (\text{guards } gs \ c) \ Q, A \implies$$

$$\Gamma, \Theta \vdash_{t/F} P \ (\text{guards } ((f, g) \# gs) \ c) \ Q, A$$

$$\langle \text{proof} \rangle$$

**lemma** *GuardsConsGuaranteeStrip*:  

$$\Gamma, \Theta \vdash_{t/F} P \ \text{guaranteeStrip } f \ g \ (\text{guards } gs \ c) \ Q, A \implies$$

$$\Gamma, \Theta \vdash_{t/F} P \ (\text{guards } (\text{guaranteeStripPair } f \ g \ # gs) \ c) \ Q, A$$

$$\langle \text{proof} \rangle$$

**lemma** *While*:  
**assumes** *P-I*:  $P \subseteq I$

**assumes** *deriv-body*:  
 $\forall \sigma. \Gamma, \Theta \vdash_{t/F} (\{\sigma\} \cap I \cap b) c (\{t. (t, \sigma) \in V\} \cap I), A$   
**assumes** *I-Q*:  $I \cap -b \subseteq Q$   
**assumes** *wf*:  $wf V$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{whileAnno } b I V c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *WhileInvPost*:  
**assumes** *P-I*:  $P \subseteq I$   
**assumes** *termi-body*:  
 $\forall \sigma. \Gamma, \Theta \vdash_{t/UNIV} (\{\sigma\} \cap I \cap b) c (\{t. (t, \sigma) \in V\} \cap P), A$   
**assumes** *deriv-body*:  
 $\Gamma, \Theta \vdash_{t/F} (I \cap b) c I, A$   
**assumes** *I-Q*:  $I \cap -b \subseteq Q$   
**assumes** *wf*:  $wf V$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{whileAnno } b I V c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma**  $\Gamma, \Theta \vdash_{t/F} (P \cap b) c Q, A \implies \Gamma, \Theta \vdash_{t/F} (P \cap b) (\text{Seq } c (\text{Guard } f Q \text{ Skip})) Q, A$   
 $\langle \text{proof} \rangle$

*J* will be instantiated by tactic with  $gs' \cap I$  for those guards that are not stripped.

**lemma** *WhileAnnoG*:  
 $\Gamma, \Theta \vdash_{t/F} P (\text{guards } gs$   
 $\quad (\text{whileAnno } b J V (\text{Seq } c (\text{guards } gs \text{ Skip})))) Q, A$   
 $\implies$   
 $\Gamma, \Theta \vdash_{t/F} P (\text{whileAnnoG } gs b I V c) Q, A$   
 $\langle \text{proof} \rangle$

This form stems from *strip-guards F (whileAnnoG gs b I V c)*

**lemma** *WhileNoGuard'*:  
**assumes** *P-I*:  $P \subseteq I$   
**assumes** *deriv-body*:  $\forall \sigma. \Gamma, \Theta \vdash_{t/F} (\{\sigma\} \cap I \cap b) c (\{t. (t, \sigma) \in V\} \cap I), A$   
**assumes** *I-Q*:  $I \cap -b \subseteq Q$   
**assumes** *wf*:  $wf V$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{whileAnno } b I V (\text{Seq } c \text{ Skip})) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *WhileAnnoFix*:  
**assumes** *consequence*:  $P \subseteq \{s. (\exists Z. s \in I Z \wedge (I Z \cap -b \subseteq Q))\}$   
**assumes** *bdy*:  $\forall Z \sigma. \Gamma, \Theta \vdash_{t/F} (\{\sigma\} \cap I Z \cap b) (c Z) (\{t. (t, \sigma) \in V Z\} \cap I Z), A$   
**assumes** *bdy-constant*:  $\forall Z. c Z = c \text{ undefined}$   
**assumes** *wf*:  $\forall Z. wf (V Z)$

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*whileAnnoFix*  $b I V c$ )  $Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *WhileAnnoFix'*:

**assumes** *consequence*:  $P \subseteq \{s. (\exists Z. s \in I Z \wedge$   
 $(\forall t. t \in I Z \cap \neg b \longrightarrow t \in Q))\}$

**assumes** *bdy*:  $\forall Z \sigma. \Gamma, \Theta \vdash_{t/F} (\{\sigma\} \cap I Z \cap b) (c Z) (\{t. (t, \sigma) \in V Z\} \cap I Z), A$

**assumes** *bdy-constant*:  $\forall Z. c Z = c \text{ undefined}$

**assumes** *wf*:  $\forall Z. wf (V Z)$

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*whileAnnoFix*  $b I V c$ )  $Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *WhileAnnoGFix*:

**assumes** *whileAnnoFix*:

$\Gamma, \Theta \vdash_{t/F} P$  (*guards*  $gs$   
 $(\text{whileAnnoFix } b J V (\lambda Z. (\text{Seq } (c Z) (\text{guards } gs \text{ Skip}))))$ )  $Q, A$

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*whileAnnoGFix*  $gs b I V c$ )  $Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *Bind*:

**assumes** *adapt*:  $P \subseteq \{s. s \in P' s\}$

**assumes** *c*:  $\forall s. \Gamma, \Theta \vdash_{t/F} (P' s) (c (e s)) Q, A$

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*bind*  $e c$ )  $Q, A$

$\langle \text{proof} \rangle$

**lemma** *Block-exn*:

**assumes** *adapt*:  $P \subseteq \{s. \text{init } s \in P' s\}$

**assumes** *bdy*:  $\forall s. \Gamma, \Theta \vdash_{t/F} (P' s) \text{ bdy } \{t. \text{return } s t \in R s t\}, \{t. \text{result-exn } (\text{return}$   
 $s t) t \in A\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*block-exn* *init* *bdy* *return* *result-exn* *c*)  $Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *Block*:

**assumes** *adapt*:  $P \subseteq \{s. \text{init } s \in P' s\}$

**assumes** *bdy*:  $\forall s. \Gamma, \Theta \vdash_{t/F} (P' s) \text{ bdy } \{t. \text{return } s t \in R s t\}, \{t. \text{return } s t \in A\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*block* *init* *bdy* *return* *c*)  $Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *BlockSwap*:

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$

**assumes** *bdy*:  $\forall s. \Gamma, \Theta \vdash_{t/F} (P' s) \text{ bdy } \{t. \text{return } s t \in R s t\}, \{t. \text{return } s t \in A\}$

**assumes** *adapt*:  $P \subseteq \{s. \text{init } s \in P' s\}$

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*block* *init* *bdy* *return* *c*)  $Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *Block-exnSwap*:

**assumes**  $c: \forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$

**assumes**  $bdy: \forall s. \Gamma, \Theta \vdash_{t/F} (P' s) bdy \{t. \text{return } s t \in R s t\}, \{t. \text{result-exn (return } s t) t \in A\}$

**assumes**  $adapt: P \subseteq \{s. \text{init } s \in P' s\}$

**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{block-exn init } bdy \text{ return result-exn } c) Q, A$

*<proof>*

**lemma** *Block-exnSpec*:

**assumes**  $adapt: P \subseteq \{s. \exists Z. \text{init } s \in P' Z \wedge$

$(\forall t. t \in Q' Z \longrightarrow \text{return } s t \in R s t) \wedge$

$(\forall t. t \in A' Z \longrightarrow \text{result-exn (return } s t) t \in A)\}$

**assumes**  $c: \forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$

**assumes**  $bdy: \forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) bdy (Q' Z), (A' Z)$

**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{block-exn init } bdy \text{ return result-exn } c) Q, A$

*<proof>*

**lemma** *BlockSpec*:

**assumes**  $adapt: P \subseteq \{s. \exists Z. \text{init } s \in P' Z \wedge$

$(\forall t. t \in Q' Z \longrightarrow \text{return } s t \in R s t) \wedge$

$(\forall t. t \in A' Z \longrightarrow \text{return } s t \in A)\}$

**assumes**  $c: \forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$

**assumes**  $bdy: \forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) bdy (Q' Z), (A' Z)$

**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{block init } bdy \text{ return } c) Q, A$

*<proof>*

**lemma** *Throw*:  $P \subseteq A \implies \Gamma, \Theta \vdash_{t/F} P \text{ Throw } Q, A$

*<proof>*

**lemmas** *Catch = hoaret.Catch*

**lemma** *CatchSwap*:  $[[\Gamma, \Theta \vdash_{t/F} R c_2 Q, A; \Gamma, \Theta \vdash_{t/F} P c_1 Q, R]] \implies \Gamma, \Theta \vdash_{t/F} P \text{ Catch}$

$c_1 c_2 Q, A$

*<proof>*

**lemma** *raise*:  $P \subseteq \{s. f s \in A\} \implies \Gamma, \Theta \vdash_{t/F} P \text{ raise } f Q, A$

*<proof>*

**lemma** *condCatch*:  $[[\Gamma, \Theta \vdash_{t/F} P c_1 Q, ((b \cap R) \cup (-b \cap A)); \Gamma, \Theta \vdash_{t/F} R c_2 Q, A]]$

$\implies \Gamma, \Theta \vdash_{t/F} P \text{ condCatch } c_1 b c_2 Q, A$

*<proof>*

**lemma** *condCatchSwap*:  $[[\Gamma, \Theta \vdash_{t/F} R c_2 Q, A; \Gamma, \Theta \vdash_{t/F} P c_1 Q, ((b \cap R) \cup (-b \cap A))]]$

$\implies \Gamma, \Theta \vdash_{t/F} P \text{ condCatch } c_1 b c_2 Q, A$

*<proof>*

**lemma** *Proc-exnSpec*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall t. t \in Q' Z \longrightarrow \text{return } s t \in R s t) \wedge$   
 $(\forall t. t \in A' Z \longrightarrow \text{result-exn } (\text{return } s t) t \in A)\}$   
**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$   
**assumes** *p*:  $\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{ Call } p (Q' Z), (A' Z)$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{call-exn init } p \text{ return result-exn } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *ProcSpec*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall t. t \in Q' Z \longrightarrow \text{return } s t \in R s t) \wedge$   
 $(\forall t. t \in A' Z \longrightarrow \text{return } s t \in A)\}$   
**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$   
**assumes** *p*:  $\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{ Call } p (Q' Z), (A' Z)$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{call init } p \text{ return } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *Proc-exnSpec'*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall t \in Q' Z. \text{return } s t \in R s t) \wedge$   
 $(\forall t \in A' Z. \text{result-exn } (\text{return } s t) t \in A)\}$   
**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$   
**assumes** *p*:  $\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{ Call } p (Q' Z), (A' Z)$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{call-exn init } p \text{ return result-exn } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *ProcSpec'*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall t \in Q' Z. \text{return } s t \in R s t) \wedge$   
 $(\forall t \in A' Z. \text{return } s t \in A)\}$   
**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$   
**assumes** *p*:  $\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{ Call } p (Q' Z), (A' Z)$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{call init } p \text{ return } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *Proc-exnSpecNoAbrupt*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall t. t \in Q' Z \longrightarrow \text{return } s t \in R s t)\}$   
**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$   
**assumes** *p*:  $\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{ Call } p (Q' Z), \{\}$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{call-exn init } p \text{ return result-exn } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *ProcSpecNoAbrupt*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' Z \wedge$

$(\forall t. t \in Q' Z \longrightarrow \text{return } s t \in R s t)$

**assumes**  $c: \forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$   
**assumes**  $p: \forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{ Call } p (Q' Z), \{\}$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{call init } p \text{ return } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *FCall*:  
 $\Gamma, \Theta \vdash_{t/F} P (\text{call init } p \text{ return } (\lambda s t. c (\text{result } t))) Q, A$   
 $\implies \Gamma, \Theta \vdash_{t/F} P (\text{fcall init } p \text{ return result } c) Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *ProcRec*:  
**assumes** *deriv-bodies*:  
 $\forall p \in \text{Procs.}$   
 $\forall \sigma Z. \Gamma, \Theta \cup (\bigcup_{q \in \text{Procs.}} \bigcup Z.$   
 $\{ (P q Z \cap \{s. ((s, q), \sigma, p) \in r\}, q, Q q Z, A q Z) \}$   
 $\vdash_{t/F} (\{\sigma\} \cap P p Z) (\text{the } (\Gamma p)) (Q p Z), (A p Z)$   
**assumes** *wf*:  $wf r$   
**assumes** *Procs-defined*:  $\text{Procs} \subseteq \text{dom } \Gamma$   
**shows**  $\forall p \in \text{Procs. } \forall Z.$   
 $\Gamma, \Theta \vdash_{t/F} (P p Z) \text{ Call } p (Q p Z), (A p Z)$   
 $\langle \text{proof} \rangle$

**lemma** *ProcRec'*:  
**assumes** *ctxt*:  
 $\Theta' = (\lambda \sigma p. \Theta \cup (\bigcup_{q \in \text{Procs.}}$   
 $\bigcup Z. \{ (P q Z \cap \{s. ((s, q), \sigma, p) \in r\}, q, Q q Z, A q Z) \}))$   
**assumes** *deriv-bodies*:  
 $\forall p \in \text{Procs.}$   
 $\forall \sigma Z. \Gamma, \Theta' \sigma p \vdash_{t/F} (\{\sigma\} \cap P p Z) (\text{the } (\Gamma p)) (Q p Z), (A p Z)$   
**assumes** *wf*:  $wf r$   
**assumes** *Procs-defined*:  $\text{Procs} \subseteq \text{dom } \Gamma$   
**shows**  $\forall p \in \text{Procs. } \forall Z. \Gamma, \Theta \vdash_{t/F} (P p Z) \text{ Call } p (Q p Z), (A p Z)$   
 $\langle \text{proof} \rangle$

**lemma** *ProcRecList*:  
**assumes** *deriv-bodies*:  
 $\forall p \in \text{set Procs.}$   
 $\forall \sigma Z. \Gamma, \Theta \cup (\bigcup_{q \in \text{set Procs.}} \bigcup Z.$   
 $\{ (P q Z \cap \{s. ((s, q), \sigma, p) \in r\}, q, Q q Z, A q Z) \}$   
 $\vdash_{t/F} (\{\sigma\} \cap P p Z) (\text{the } (\Gamma p)) (Q p Z), (A p Z)$   
**assumes** *wf*:  $wf r$   
**assumes** *dist*: *distinct Procs*  
**assumes** *Procs-defined*:  $\text{set Procs} \subseteq \text{dom } \Gamma$   
**shows**  $\forall p \in \text{set Procs. } \forall Z.$   
 $\Gamma, \Theta \vdash_{t/F} (P p Z) \text{ Call } p (Q p Z), (A p Z)$   
 $\langle \text{proof} \rangle$

**lemma ProcRecSpecs:**

$\llbracket \forall \sigma. \forall (P, p, Q, A) \in \text{Specs}.$   
 $\Gamma, \Theta \cup ((\lambda(P, q, Q, A). (P \cap \{s. ((s, q), (\sigma, p)) \in r\}, q, Q, A)) \text{ 'Specs})$   
 $\vdash_{t/F} (\{\sigma\} \cap P) \text{ (the } (\Gamma p)) Q, A;$   
 $wf r;$   
 $\forall (P, p, Q, A) \in \text{Specs}. p \in \text{dom } \Gamma \rrbracket$   
 $\implies \forall (P, p, Q, A) \in \text{Specs}. \Gamma, \Theta \vdash_{t/F} P \text{ (Call } p) Q, A$   
 <proof>

**lemma ProcRec1:**

**assumes deriv-body:**  
 $\forall \sigma Z. \Gamma, \Theta \cup (\bigcup Z. \{(P Z \cap \{s. ((s, p), \sigma, p) \in r\}, p, Q Z, A Z)\})$   
 $\vdash_{t/F} (\{\sigma\} \cap P Z) \text{ (the } (\Gamma p)) (Q Z), (A Z)$   
**assumes wf:**  $wf r$   
**assumes p-defined:**  $p \in \text{dom } \Gamma$   
**shows**  $\forall Z. \Gamma, \Theta \vdash_{t/F} (P Z) \text{ Call } p (Q Z), (A Z)$   
 <proof>

**lemma ProcNoRec1:**

**assumes deriv-body:**  
 $\forall Z. \Gamma, \Theta \vdash_{t/F} (P Z) \text{ (the } (\Gamma p)) (Q Z), (A Z)$   
**assumes p-defined:**  $p \in \text{dom } \Gamma$   
**shows**  $\forall Z. \Gamma, \Theta \vdash_{t/F} (P Z) \text{ Call } p (Q Z), (A Z)$   
 <proof>

**lemma ProcBody:**

**assumes WP:**  $P \subseteq P'$   
**assumes deriv-body:**  $\Gamma, \Theta \vdash_{t/F} P' \text{ body } Q, A$   
**assumes body:**  $\Gamma p = \text{Some body}$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ Call } p Q, A$   
 <proof>

**lemma CallBody:**

**assumes adapt:**  $P \subseteq \{s. \text{init } s \in P' s\}$   
**assumes bdy:**  $\forall s. \Gamma, \Theta \vdash_{t/F} (P' s) \text{ body } \{t. \text{return } s t \in R s t\}, \{t. \text{return } s t \in A\}$   
**assumes c:**  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$   
**assumes body:**  $\Gamma p = \text{Some body}$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ (call init } p \text{ return } c) Q, A$   
 <proof>

**lemma Call-exnBody:**

**assumes adapt:**  $P \subseteq \{s. \text{init } s \in P' s\}$   
**assumes bdy:**  $\forall s. \Gamma, \Theta \vdash_{t/F} (P' s) \text{ body } \{t. \text{return } s t \in R s t\}, \{t. \text{result-exn (return } s t) t \in A\}$   
**assumes c:**  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$   
**assumes body:**  $\Gamma p = \text{Some body}$

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*call-exn init p return result-exn c*)  $Q, A$   
 ⟨*proof*⟩

**lemmas** *ProcModifyReturn* = *HoareTotalProps.ProcModifyReturn*

**lemmas** *ProcModifyReturnSameFaults* = *HoareTotalProps.ProcModifyReturnSameFaults*

**lemmas** *Proc-exnModifyReturn* = *HoareTotalProps.Proc-exnModifyReturn*

**lemmas** *Proc-exnModifyReturnSameFaults* = *HoareTotalProps.Proc-exnModifyReturnSameFaults*

**lemma** *ProcModifyReturnNoAbr*:

**assumes** *spec*:  $\Gamma, \Theta \vdash_{t/F} P$  (*call init p return' c*)  $Q, A$

**assumes** *result-conform*:

$\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow (\text{return}' s t) = (\text{return } s t)$

**assumes** *modifies-spec*:

$\forall \sigma. \Gamma, \Theta \vdash_{UNIV} \{\sigma\}$  *Call p (Modif  $\sigma$ ), {}*

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*call init p return c*)  $Q, A$

⟨*proof*⟩

**lemma** *Proc-exnModifyReturnNoAbr*:

**assumes** *spec*:  $\Gamma, \Theta \vdash_{t/F} P$  (*call-exn init p return' result-exn c*)  $Q, A$

**assumes** *result-conform*:

$\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow (\text{return}' s t) = (\text{return } s t)$

**assumes** *modifies-spec*:

$\forall \sigma. \Gamma, \Theta \vdash_{UNIV} \{\sigma\}$  *Call p (Modif  $\sigma$ ), {}*

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*call-exn init p return result-exn c*)  $Q, A$

⟨*proof*⟩

**lemma** *ProcModifyReturnNoAbrSameFaults*:

**assumes** *spec*:  $\Gamma, \Theta \vdash_{t/F} P$  (*call init p return' c*)  $Q, A$

**assumes** *result-conform*:

$\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow (\text{return}' s t) = (\text{return } s t)$

**assumes** *modifies-spec*:

$\forall \sigma. \Gamma, \Theta \vdash_{t/F} \{\sigma\}$  *Call p (Modif  $\sigma$ ), {}*

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*call init p return c*)  $Q, A$

⟨*proof*⟩

**lemma** *Proc-exnModifyReturnNoAbrSameFaults*:

**assumes** *spec*:  $\Gamma, \Theta \vdash_{t/F} P$  (*call-exn init p return' result-exn c*)  $Q, A$

**assumes** *result-conform*:

$\forall s t. t \in \text{Modif } (\text{init } s) \longrightarrow (\text{return}' s t) = (\text{return } s t)$

**assumes** *modifies-spec*:

$\forall \sigma. \Gamma, \Theta \vdash_{t/F} \{\sigma\}$  *Call p (Modif  $\sigma$ ), {}*

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*call-exn init p return result-exn c*)  $Q, A$

⟨*proof*⟩

**lemma** *DynProc-exn*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' s Z \wedge$   
 $(\forall t. t \in Q' s Z \longrightarrow \text{return } s t \in R s t) \wedge$   
 $(\forall t. t \in A' s Z \longrightarrow \text{result-exn } (\text{return } s t) t \in A)\}$   
**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$   
**assumes** *p*:  $\forall s \in P. \forall Z. \Gamma, \Theta \vdash_{t/F} (P' s Z) \text{Call } (p s) (Q' s Z), (A' s Z)$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ dynCall-exn } f \text{ UNIV init } p \text{ return result-exn } c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProc-exn-guards-cons*:

**assumes** *p*:  $\Gamma, \Theta \vdash_{t/F} P \text{ dynCall-exn } f \text{ UNIV init } p \text{ return result-exn } c Q, A$   
**shows**  $\Gamma, \Theta \vdash_{t/F} (g \cap P) \text{ dynCall-exn } f g \text{ init } p \text{ return result-exn } c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProc*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' s Z \wedge$   
 $(\forall t. t \in Q' s Z \longrightarrow \text{return } s t \in R s t) \wedge$   
 $(\forall t. t \in A' s Z \longrightarrow \text{return } s t \in A)\}$   
**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$   
**assumes** *p*:  $\forall s \in P. \forall Z. \Gamma, \Theta \vdash_{t/F} (P' s Z) \text{Call } (p s) (Q' s Z), (A' s Z)$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ dynCall init } p \text{ return } c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProc-exn'*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' s Z \wedge$   
 $(\forall t \in Q' s Z. \text{return } s t \in R s t) \wedge$   
 $(\forall t \in A' s Z. \text{result-exn } (\text{return } s t) t \in A)\}$   
**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$   
**assumes** *p*:  $\forall s \in P. \forall Z. \Gamma, \Theta \vdash_{t/F} (P' s Z) \text{Call } (p s) (Q' s Z), (A' s Z)$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ dynCall-exn } f \text{ UNIV init } p \text{ return result-exn } c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProc'*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. \text{init } s \in P' s Z \wedge$   
 $(\forall t \in Q' s Z. \text{return } s t \in R s t) \wedge$   
 $(\forall t \in A' s Z. \text{return } s t \in A)\}$   
**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$   
**assumes** *p*:  $\forall s \in P. \forall Z. \Gamma, \Theta \vdash_{t/F} (P' s Z) \text{Call } (p s) (Q' s Z), (A' s Z)$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ dynCall init } p \text{ return } c Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProc-exnStaticSpec*:

**assumes** *adapt*:  $P \subseteq \{s. s \in S \wedge (\exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall \tau. \tau \in Q' Z \longrightarrow \text{return } s \tau \in R s \tau) \wedge$   
 $(\forall \tau. \tau \in A' Z \longrightarrow \text{result-exn } (\text{return } s \tau) \tau \in A)\}\}$   
**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$   
**assumes** *spec*:  $\forall s \in S. \forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{Call } (p s) (Q' Z), (A' Z)$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{dynCall-exn } f \text{ UNIV init } p \text{ return result-exn } c) Q, A$

$\langle proof \rangle$

**lemma** *DynProcStaticSpec*:

**assumes** *adapt*:  $P \subseteq \{s. s \in S \wedge (\exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall \tau. \tau \in Q' Z \longrightarrow \text{return } s \tau \in R s \tau) \wedge$   
 $(\forall \tau. \tau \in A' Z \longrightarrow \text{return } s \tau \in A))\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$

**assumes** *spec*:  $\forall s \in S. \forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{Call } (p s) (Q' Z), (A' Z)$

**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{dynCall init } p \text{ return } c) Q, A$

$\langle proof \rangle$

**lemma** *DynProc-exnProcPar*:

**assumes** *adapt*:  $P \subseteq \{s. p s = q \wedge (\exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall \tau. \tau \in Q' Z \longrightarrow \text{return } s \tau \in R s \tau) \wedge$   
 $(\forall \tau. \tau \in A' Z \longrightarrow \text{result-exn } (\text{return } s \tau) \tau \in A))\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$

**assumes** *spec*:  $\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{Call } q (Q' Z), (A' Z)$

**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{dynCall-exn } f \text{ UNIV init } p \text{ return result-exn } c) Q, A$

$\langle proof \rangle$

**lemma** *DynProcProcPar*:

**assumes** *adapt*:  $P \subseteq \{s. p s = q \wedge (\exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall \tau. \tau \in Q' Z \longrightarrow \text{return } s \tau \in R s \tau) \wedge$   
 $(\forall \tau. \tau \in A' Z \longrightarrow \text{return } s \tau \in A))\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$

**assumes** *spec*:  $\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{Call } q (Q' Z), (A' Z)$

**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{dynCall init } p \text{ return } c) Q, A$

$\langle proof \rangle$

**lemma** *DynProc-exnProcParNoAbrupt*:

**assumes** *adapt*:  $P \subseteq \{s. p s = q \wedge (\exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall \tau. \tau \in Q' Z \longrightarrow \text{return } s \tau \in R s \tau))\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$

**assumes** *spec*:  $\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{Call } q (Q' Z), \{\}$

**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{dynCall-exn } f \text{ UNIV init } p \text{ return result-exn } c) Q, A$

$\langle proof \rangle$

**lemma** *DynProcProcParNoAbrupt*:

**assumes** *adapt*:  $P \subseteq \{s. p s = q \wedge (\exists Z. \text{init } s \in P' Z \wedge$   
 $(\forall \tau. \tau \in Q' Z \longrightarrow \text{return } s \tau \in R s \tau))\}$

**assumes** *c*:  $\forall s t. \Gamma, \Theta \vdash_{t/F} (R s t) (c s t) Q, A$

**assumes** *spec*:  $\forall Z. \Gamma, \Theta \vdash_{t/F} (P' Z) \text{Call } q (Q' Z), \{\}$

**shows**  $\Gamma, \Theta \vdash_{t/F} P (\text{dynCall init } p \text{ return } c) Q, A$

$\langle proof \rangle$

**lemma** *DynProc-exnModifyReturnNoAbr*:

**assumes** *to-prove*:  $\Gamma, \Theta \vdash_{t/F} P$  (*dynCall-exn*  $f g$  *init*  $p$  *return'* *result-exn*  $c$ )  $Q, A$   
**assumes** *ret-nrm-modif*:  $\forall s t. t \in (\text{Modif } (\text{init } s))$   
 $\longrightarrow \text{return}' s t = \text{return } s t$   
**assumes** *modif-clause*:  
 $\forall s \in P. \forall \sigma. \Gamma, \Theta \vdash_{UNIV} \{\sigma\} \text{Call } (p s) (\text{Modif } \sigma), \{\}$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*dynCall-exn*  $f g$  *init*  $p$  *return* *result-exn*  $c$ )  $Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *DynProcModifyReturnNoAbr*:  
**assumes** *to-prove*:  $\Gamma, \Theta \vdash_{t/F} P$  (*dynCall* *init*  $p$  *return'*  $c$ )  $Q, A$   
**assumes** *ret-nrm-modif*:  $\forall s t. t \in (\text{Modif } (\text{init } s))$   
 $\longrightarrow \text{return}' s t = \text{return } s t$   
**assumes** *modif-clause*:  
 $\forall s \in P. \forall \sigma. \Gamma, \Theta \vdash_{UNIV} \{\sigma\} \text{Call } (p s) (\text{Modif } \sigma), \{\}$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*dynCall* *init*  $p$  *return*  $c$ )  $Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *ProcDyn-exnModifyReturnNoAbrSameFaults*:  
**assumes** *to-prove*:  $\Gamma, \Theta \vdash_{t/F} P$  (*dynCall-exn*  $f g$  *init*  $p$  *return'* *result-exn*  $c$ )  $Q, A$   
**assumes** *ret-nrm-modif*:  $\forall s t. t \in (\text{Modif } (\text{init } s))$   
 $\longrightarrow \text{return}' s t = \text{return } s t$   
**assumes** *modif-clause*:  
 $\forall s \in P. \forall \sigma. \Gamma, \Theta \vdash_{t/F} \{\sigma\} (\text{Call } (p s)) (\text{Modif } \sigma), \{\}$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*dynCall-exn*  $f g$  *init*  $p$  *return* *result-exn*  $c$ )  $Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *ProcDynModifyReturnNoAbrSameFaults*:  
**assumes** *to-prove*:  $\Gamma, \Theta \vdash_{t/F} P$  (*dynCall* *init*  $p$  *return'*  $c$ )  $Q, A$   
**assumes** *ret-nrm-modif*:  $\forall s t. t \in (\text{Modif } (\text{init } s))$   
 $\longrightarrow \text{return}' s t = \text{return } s t$   
**assumes** *modif-clause*:  
 $\forall s \in P. \forall \sigma. \Gamma, \Theta \vdash_{t/F} \{\sigma\} (\text{Call } (p s)) (\text{Modif } \sigma), \{\}$   
**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*dynCall* *init*  $p$  *return*  $c$ )  $Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *Proc-exnProcParModifyReturn*:  
**assumes**  $q: P \subseteq \{s. p s = q\} \cap P'$   
— *DynProcProcPar* introduces the same constraint as first conjunction in  $P'$ , so the veg can simplify it.  
**assumes** *to-prove*:  $\Gamma, \Theta \vdash_{t/F} P'$  (*dynCall-exn*  $f g$  *init*  $p$  *return'* *result-exn*  $c$ )  $Q, A$   
**assumes** *ret-nrm-modif*:  $\forall s t. t \in (\text{Modif } (\text{init } s))$   
 $\longrightarrow \text{return}' s t = \text{return } s t$   
**assumes** *ret-abr-modif*:  $\forall s t. t \in (\text{ModifAbr } (\text{init } s))$   
 $\longrightarrow \text{result-exn } (\text{return}' s t) t = \text{result-exn } (\text{return } s t) t$   
**assumes** *modif-clause*:  
 $\forall \sigma. \Gamma, \Theta \vdash_{UNIV} \{\sigma\} (\text{Call } q) (\text{Modif } \sigma), (\text{ModifAbr } \sigma)$

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*dynCall-exn*  $f g$  *init*  $p$  *return result-exn*  $c$ )  $Q, A$   
 ⟨*proof*⟩

**lemma** *ProcProcParModifyReturn*:

**assumes**  $q: P \subseteq \{s. p s = q\} \cap P'$

— *DynProcProcPar* introduces the same constraint as first conjunction in  $P'$ , so the vcg can simplify it.

**assumes** *to-prove*:  $\Gamma, \Theta \vdash_{t/F} P'$  (*dynCall* *init*  $p$  *return'*  $c$ )  $Q, A$

**assumes** *ret-nrm-modif*:  $\forall s t. t \in (\text{Modif } (\text{init } s))$

→  $\text{return}' s t = \text{return } s t$

**assumes** *ret-abr-modif*:  $\forall s t. t \in (\text{ModifAbr } (\text{init } s))$

→  $\text{return}' s t = \text{return } s t$

**assumes** *modif-clause*:

$\forall \sigma. \Gamma, \Theta \vdash_{UNIV} \{\sigma\}$  (*Call*  $q$ ) (*Modif*  $\sigma$ ), (*ModifAbr*  $\sigma$ )

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*dynCall* *init*  $p$  *return*  $c$ )  $Q, A$

⟨*proof*⟩

**lemma** *Proc-exnProcParModifyReturnSameFaults*:

**assumes**  $q: P \subseteq \{s. p s = q\} \cap P'$

— *DynProcProcPar* introduces the same constraint as first conjunction in  $P'$ , so the vcg can simplify it.

**assumes** *to-prove*:  $\Gamma, \Theta \vdash_{t/F} P'$  (*dynCall-exn*  $f g$  *init*  $p$  *return'* *result-exn*  $c$ )  $Q, A$

**assumes** *ret-nrm-modif*:  $\forall s t. t \in (\text{Modif } (\text{init } s))$

→  $\text{return}' s t = \text{return } s t$

**assumes** *ret-abr-modif*:  $\forall s t. t \in (\text{ModifAbr } (\text{init } s))$

→  $\text{result-exn } (\text{return}' s t) t = \text{result-exn } (\text{return } s t) t$

**assumes** *modif-clause*:

$\forall \sigma. \Gamma, \Theta \vdash_{t/F} \{\sigma\}$  *Call*  $q$  (*Modif*  $\sigma$ ), (*ModifAbr*  $\sigma$ )

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*dynCall-exn*  $f g$  *init*  $p$  *return result-exn*  $c$ )  $Q, A$

⟨*proof*⟩

**lemma** *ProcProcParModifyReturnSameFaults*:

**assumes**  $q: P \subseteq \{s. p s = q\} \cap P'$

— *DynProcProcPar* introduces the same constraint as first conjunction in  $P'$ , so the vcg can simplify it.

**assumes** *to-prove*:  $\Gamma, \Theta \vdash_{t/F} P'$  (*dynCall* *init*  $p$  *return'*  $c$ )  $Q, A$

**assumes** *ret-nrm-modif*:  $\forall s t. t \in (\text{Modif } (\text{init } s))$

→  $\text{return}' s t = \text{return } s t$

**assumes** *ret-abr-modif*:  $\forall s t. t \in (\text{ModifAbr } (\text{init } s))$

→  $\text{return}' s t = \text{return } s t$

**assumes** *modif-clause*:

$\forall \sigma. \Gamma, \Theta \vdash_{t/F} \{\sigma\}$  *Call*  $q$  (*Modif*  $\sigma$ ), (*ModifAbr*  $\sigma$ )

**shows**  $\Gamma, \Theta \vdash_{t/F} P$  (*dynCall* *init*  $p$  *return*  $c$ )  $Q, A$

⟨*proof*⟩

**lemma** *Proc-exnProcParModifyReturnNoAbr*:

**assumes**  $q: P \subseteq \{s. p s = q\} \cap P'$

— *DynProcProcParNoAbrupt* introduces the same constraint as first conjunction in  $P'$ , so the vcg can simplify it.

**assumes** *to-prove*:  $\Gamma, \Theta \vdash_{t/F} P' \text{ (dynCall-exn } f \ g \ \text{init } p \ \text{return}' \ \text{result-exn } c) \ Q, A$

**assumes** *ret-nrm-modif*:  $\forall s \ t. t \in (\text{Modif } (\text{init } s))$   
 $\longrightarrow \text{return}' \ s \ t = \text{return } s \ t$

**assumes** *modif-clause*:

$\forall \sigma. \Gamma, \Theta \vdash_{UNIV} \{\sigma\} \text{ (Call } q) \text{ (Modif } \sigma), \{\}$

**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ (dynCall-exn } f \ g \ \text{init } p \ \text{return} \ \text{result-exn } c) \ Q, A$

*<proof>*

**lemma** *ProcProcParModifyReturnNoAbr*:

**assumes**  $q: P \subseteq \{s. p \ s = q\} \cap P'$

— *DynProcProcParNoAbrupt* introduces the same constraint as first conjunction in  $P'$ , so the vcg can simplify it.

**assumes** *to-prove*:  $\Gamma, \Theta \vdash_{t/F} P' \text{ (dynCall } \text{init } p \ \text{return}' \ c) \ Q, A$

**assumes** *ret-nrm-modif*:  $\forall s \ t. t \in (\text{Modif } (\text{init } s))$   
 $\longrightarrow \text{return}' \ s \ t = \text{return } s \ t$

**assumes** *modif-clause*:

$\forall \sigma. \Gamma, \Theta \vdash_{UNIV} \{\sigma\} \text{ (Call } q) \text{ (Modif } \sigma), \{\}$

**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ (dynCall } \text{init } p \ \text{return} \ c) \ Q, A$

*<proof>*

**lemma** *Proc-exnProcParModifyReturnNoAbrSameFaults*:

**assumes**  $q: P \subseteq \{s. p \ s = q\} \cap P'$

— *DynProcProcParNoAbrupt* introduces the same constraint as first conjunction in  $P'$ , so the vcg can simplify it.

**assumes** *to-prove*:  $\Gamma, \Theta \vdash_{t/F} P' \text{ (dynCall-exn } f \ g \ \text{init } p \ \text{return}' \ \text{result-exn } c) \ Q, A$

**assumes** *ret-nrm-modif*:  $\forall s \ t. t \in (\text{Modif } (\text{init } s))$   
 $\longrightarrow \text{return}' \ s \ t = \text{return } s \ t$

**assumes** *modif-clause*:

$\forall \sigma. \Gamma, \Theta \vdash_{t/F} \{\sigma\} \text{ (Call } q) \text{ (Modif } \sigma), \{\}$

**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ (dynCall-exn } f \ g \ \text{init } p \ \text{return} \ \text{result-exn } c) \ Q, A$

*<proof>*

**lemma** *ProcProcParModifyReturnNoAbrSameFaults*:

**assumes**  $q: P \subseteq \{s. p \ s = q\} \cap P'$

— *DynProcProcParNoAbrupt* introduces the same constraint as first conjunction in  $P'$ , so the vcg can simplify it.

**assumes** *to-prove*:  $\Gamma, \Theta \vdash_{t/F} P' \text{ (dynCall } \text{init } p \ \text{return}' \ c) \ Q, A$

**assumes** *ret-nrm-modif*:  $\forall s \ t. t \in (\text{Modif } (\text{init } s))$   
 $\longrightarrow \text{return}' \ s \ t = \text{return } s \ t$

**assumes** *modif-clause*:

$\forall \sigma. \Gamma, \Theta \vdash_{t/F} \{\sigma\} \text{ (Call } q) \text{ (Modif } \sigma), \{\}$

**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ (dynCall } \text{init } p \ \text{return} \ c) \ Q, A$

*<proof>*

**lemma** *MergeGuards-iff*:  $\Gamma, \Theta \vdash_{t/F} P \text{ merge-guards } c \ Q, A = \Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$   
 ⟨proof⟩

**lemma** *CombineStrip'*:  
**assumes** *deriv*:  $\Gamma, \Theta \vdash_{t/F} P \ c' \ Q, A$   
**assumes** *deriv-strip-triv*:  $\Gamma, \{\} \vdash_{t/\{\}} P \ c'' \ UNIV, UNIV$   
**assumes** *c''*:  $c'' = \text{mark-guards } False \ (\text{strip-guards } (-F) \ c')$   
**assumes** *c*:  $\text{merge-guards } c = \text{merge-guards } (\text{mark-guards } False \ c')$   
**shows**  $\Gamma, \Theta \vdash_{t/\{\}} P \ c \ Q, A$   
 ⟨proof⟩

**lemma** *CombineStrip''*:  
**assumes** *deriv*:  $\Gamma, \Theta \vdash_{t/\{\text{True}\}} P \ c' \ Q, A$   
**assumes** *deriv-strip-triv*:  $\Gamma, \{\} \vdash_{t/\{\}} P \ c'' \ UNIV, UNIV$   
**assumes** *c''*:  $c'' = \text{mark-guards } False \ (\text{strip-guards } (\{\text{False}\}) \ c')$   
**assumes** *c*:  $\text{merge-guards } c = \text{merge-guards } (\text{mark-guards } False \ c')$   
**shows**  $\Gamma, \Theta \vdash_{t/\{\}} P \ c \ Q, A$   
 ⟨proof⟩

**lemma** *AsmUN*:  
 $(\bigcup Z. \{(P \ Z, p, Q \ Z, A \ Z)\}) \subseteq \Theta$   
 $\implies$   
 $\forall Z. \Gamma, \Theta \vdash_{t/F} (P \ Z) \ (\text{Call } p) \ (Q \ Z), (A \ Z)$   
 ⟨proof⟩

**lemma** *hoaret-to-hoarep'*:  
 $\forall Z. \Gamma, \{\} \vdash_{t/F} (P \ Z) \ p \ (Q \ Z), (A \ Z) \implies \forall Z. \Gamma, \{\} \vdash_{t/F} (P \ Z) \ p \ (Q \ Z), (A \ Z)$   
 ⟨proof⟩

**lemma** *augment-context'*:  
 $\llbracket \Theta \subseteq \Theta'; \forall Z. \Gamma, \Theta \vdash_{t/F} (P \ Z) \ p \ (Q \ Z), (A \ Z) \rrbracket$   
 $\implies \forall Z. \Gamma, \Theta' \vdash_{t/F} (P \ Z) \ p \ (Q \ Z), (A \ Z)$   
 ⟨proof⟩

**lemma** *augment-emptyFaults*:  
 $\llbracket \forall Z. \Gamma, \{\} \vdash_{t/\{\}} (P \ Z) \ p \ (Q \ Z), (A \ Z) \rrbracket \implies$   
 $\forall Z. \Gamma, \{\} \vdash_{t/F} (P \ Z) \ p \ (Q \ Z), (A \ Z)$   
 ⟨proof⟩

**lemma** *augment-FaultsUNIV*:  
 $\llbracket \forall Z. \Gamma, \{\} \vdash_{t/F} (P \ Z) \ p \ (Q \ Z), (A \ Z) \rrbracket \implies$   
 $\forall Z. \Gamma, \{\} \vdash_{t/UNIV} (P \ Z) \ p \ (Q \ Z), (A \ Z)$   
 ⟨proof⟩

**lemma** *PostConjI* [*trans*]:

$\llbracket \Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A; \Gamma, \Theta \vdash_{t/F} P \text{ c } R, B \rrbracket \Longrightarrow \Gamma, \Theta \vdash_{t/F} P \text{ c } (Q \cap R), (A \cap B)$   
 ⟨*proof*⟩

**lemma** *PostConjI'* :

$\llbracket \Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A; \Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A \rrbracket \Longrightarrow \Gamma, \Theta \vdash_{t/F} P \text{ c } R, B$   
 $\Longrightarrow \Gamma, \Theta \vdash_{t/F} P \text{ c } (Q \cap R), (A \cap B)$   
 ⟨*proof*⟩

**lemma** *PostConjE* [*consumes 1*]:

**assumes** *conj*:  $\Gamma, \Theta \vdash_{t/F} P \text{ c } (Q \cap R), (A \cap B)$   
**assumes** *E*:  $\llbracket \Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A; \Gamma, \Theta \vdash_{t/F} P \text{ c } R, B \rrbracket \Longrightarrow S$   
**shows** *S*  
 ⟨*proof*⟩

### 11.0.1 Rules for Single-Step Proof

We are now ready to introduce a set of Hoare rules to be used in single-step structured proofs in Isabelle/Isar.

Assertions of Hoare Logic may be manipulated in calculational proofs, with the inclusion expressed in terms of sets or predicates. Reversed order is supported as well.

**lemma** *annotateI* [*trans*]:

$\llbracket \Gamma, \Theta \vdash_{t/F} P \text{ anno } Q, A; c = \text{anno} \rrbracket \Longrightarrow \Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A$   
 ⟨*proof*⟩

**lemma** *annotate-normI*:

**assumes** *deriv-anno*:  $\Gamma, \Theta \vdash_{t/F} P \text{ anno } Q, A$   
**assumes** *norm-eq*: *normalize c = normalize anno*  
**shows**  $\Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A$   
 ⟨*proof*⟩

**lemma** *annotateWhile*:

$\llbracket \Gamma, \Theta \vdash_{t/F} P \text{ (whileAnnoG gs b I V c) } Q, A \rrbracket \Longrightarrow \Gamma, \Theta \vdash_{t/F} P \text{ (while gs b c) } Q, A$   
 ⟨*proof*⟩

**lemma** *reannotateWhile*:

$\llbracket \Gamma, \Theta \vdash_{t/F} P \text{ (whileAnnoG gs b I V c) } Q, A \rrbracket \Longrightarrow \Gamma, \Theta \vdash_{t/F} P \text{ (whileAnnoG gs b J V c) } Q, A$   
 ⟨*proof*⟩

**lemma** *reannotateWhileNoGuard*:

$\llbracket \Gamma, \Theta \vdash_{t/F} P \text{ (whileAnno b I V c) } Q, A \rrbracket \Longrightarrow \Gamma, \Theta \vdash_{t/F} P \text{ (whileAnno b J V c) } Q, A$   
 ⟨*proof*⟩

**lemma** [trans]:  $P' \subseteq P \implies \Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A \implies \Gamma, \Theta \vdash_{t/F} P' \text{ c } Q, A$   
 ⟨proof⟩

**lemma** [trans]:  $Q \subseteq Q' \implies \Gamma, \Theta \vdash_{t/F} P \text{ c } Q, A \implies \Gamma, \Theta \vdash_{t/F} P \text{ c } Q', A$   
 ⟨proof⟩

**lemma** [trans]:  
 $\Gamma, \Theta \vdash_{t/F} \{s. P s\} \text{ c } Q, A \implies (\bigwedge s. P' s \longrightarrow P s) \implies \Gamma, \Theta \vdash_{t/F} \{s. P' s\} \text{ c } Q, A$   
 ⟨proof⟩

**lemma** [trans]:  
 $(\bigwedge s. P' s \longrightarrow P s) \implies \Gamma, \Theta \vdash_{t/F} \{s. P s\} \text{ c } Q, A \implies \Gamma, \Theta \vdash_{t/F} \{s. P' s\} \text{ c } Q, A$   
 ⟨proof⟩

**lemma** [trans]:  
 $\Gamma, \Theta \vdash_{t/F} P \text{ c } \{s. Q s\}, A \implies (\bigwedge s. Q s \longrightarrow Q' s) \implies \Gamma, \Theta \vdash_{t/F} P \text{ c } \{s. Q' s\}, A$   
 ⟨proof⟩

**lemma** [trans]:  
 $(\bigwedge s. Q s \longrightarrow Q' s) \implies \Gamma, \Theta \vdash_{t/F} P \text{ c } \{s. Q s\}, A \implies \Gamma, \Theta \vdash_{t/F} P \text{ c } \{s. Q' s\}, A$   
 ⟨proof⟩

**lemma** [intro?]:  $\Gamma, \Theta \vdash_{t/F} P \text{ Skip } P, A$   
 ⟨proof⟩

**lemma** CondInt [trans, intro?]:  
 $[[\Gamma, \Theta \vdash_{t/F} (P \cap b) \text{ c1 } Q, A; \Gamma, \Theta \vdash_{t/F} (P \cap \neg b) \text{ c2 } Q, A]]$   
 $\implies$   
 $\Gamma, \Theta \vdash_{t/F} P \text{ (Cond } b \text{ c1 c2) } Q, A$   
 ⟨proof⟩

**lemma** CondConj [trans, intro?]:  
 $[[\Gamma, \Theta \vdash_{t/F} \{s. P s \wedge b s\} \text{ c1 } Q, A; \Gamma, \Theta \vdash_{t/F} \{s. P s \wedge \neg b s\} \text{ c2 } Q, A]]$   
 $\implies$   
 $\Gamma, \Theta \vdash_{t/F} \{s. P s\} \text{ (Cond } \{s. b s\} \text{ c1 c2) } Q, A$   
 ⟨proof⟩

end

## 12 Auxiliary Definitions/Lemmas to Facilitate Hoare Logic

**theory** Hoare **imports** HoarePartial HoareTotal **begin**

**syntax**

-hoarep-emptyFaults::

$[(\text{'s}, \text{'p}, \text{'f}) \text{ body}, (\text{'s}, \text{'p}) \text{ quadruple set},$   
 $\text{'f set}, \text{'s assn}, (\text{'s}, \text{'p}, \text{'f}) \text{ com}, \text{'s assn}, \text{'s assn}] \Rightarrow \text{bool}$   
 $(\langle (\exists -, -/\vdash (-) / -, -) \rangle [61, 60, 1000, 20, 1000, 1000] 60)$

-hoarep-emptyCtx::

$[(\text{'s}, \text{'p}, \text{'f}) \text{ body}, \text{'f set}, \text{'s assn}, (\text{'s}, \text{'p}, \text{'f}) \text{ com}, \text{'s assn}, \text{'s assn}] \Rightarrow \text{bool}$   
 $(\langle (\exists -/\vdash \prime_{/-} (-) / -, -) \rangle [61, 60, 1000, 20, 1000, 1000] 60)$

-hoarep-emptyCtx-emptyFaults::

$[(\text{'s}, \text{'p}, \text{'f}) \text{ body}, \text{'s assn}, (\text{'s}, \text{'p}, \text{'f}) \text{ com}, \text{'s assn}, \text{'s assn}] \Rightarrow \text{bool}$   
 $(\langle (\exists -/\vdash (-) / -, -) \rangle [61, 1000, 20, 1000, 1000] 60)$

-hoarep-noAbr::

$[(\text{'s}, \text{'p}, \text{'f}) \text{ body}, (\text{'s}, \text{'p}) \text{ quadruple set}, \text{'f set},$   
 $\text{'s assn}, (\text{'s}, \text{'p}, \text{'f}) \text{ com}, \text{'s assn}] \Rightarrow \text{bool}$   
 $(\langle (\exists -, -/\vdash \prime_{/-} (-) / -) \rangle [61, 60, 60, 1000, 20, 1000] 60)$

-hoarep-noAbr-emptyFaults::

$[(\text{'s}, \text{'p}, \text{'f}) \text{ body}, (\text{'s}, \text{'p}) \text{ quadruple set}, \text{'s assn}, (\text{'s}, \text{'p}, \text{'f}) \text{ com}, \text{'s assn}] \Rightarrow \text{bool}$   
 $(\langle (\exists -, -/\vdash (-) / -) \rangle [61, 60, 1000, 20, 1000] 60)$

-hoarep-emptyCtx-noAbr::

$[(\text{'s}, \text{'p}, \text{'f}) \text{ body}, \text{'f set}, \text{'s assn}, (\text{'s}, \text{'p}, \text{'f}) \text{ com}, \text{'s assn}] \Rightarrow \text{bool}$   
 $(\langle (\exists -/\vdash \prime_{/-} (-) / -) \rangle [61, 60, 1000, 20, 1000] 60)$

-hoarep-emptyCtx-noAbr-emptyFaults::

$[(\text{'s}, \text{'p}, \text{'f}) \text{ body}, \text{'s assn}, (\text{'s}, \text{'p}, \text{'f}) \text{ com}, \text{'s assn}] \Rightarrow \text{bool}$   
 $(\langle (\exists -/\vdash (-) / -) \rangle [61, 1000, 20, 1000] 60)$

-hoaret-emptyFaults::

$[(\text{'s}, \text{'p}, \text{'f}) \text{ body}, (\text{'s}, \text{'p}) \text{ quadruple set},$   
 $\text{'s assn}, (\text{'s}, \text{'p}, \text{'f}) \text{ com}, \text{'s assn}, \text{'s assn}] \Rightarrow \text{bool}$   
 $(\langle (\exists -, -/\vdash \prime_t (-) / -, -) \rangle [61, 60, 1000, 20, 1000, 1000] 60)$

-hoaret-emptyCtx::

$[(\text{'s}, \text{'p}, \text{'f}) \text{ body}, \text{'f set}, \text{'s assn}, (\text{'s}, \text{'p}, \text{'f}) \text{ com}, \text{'s assn}, \text{'s assn}] \Rightarrow \text{bool}$   
 $(\langle (\exists -/\vdash \prime_{t/-} (-) / -, -) \rangle [61, 60, 1000, 20, 1000, 1000] 60)$

-hoaret-emptyCtx-emptyFaults::

$[(\text{'s}, \text{'p}, \text{'f}) \text{ body}, \text{'s assn}, (\text{'s}, \text{'p}, \text{'f}) \text{ com}, \text{'s assn}, \text{'s assn}] \Rightarrow \text{bool}$   
 $(\langle (\exists -/\vdash \prime_t (-) / -, -) \rangle [61, 1000, 20, 1000, 1000] 60)$

-hoaret-noAbr::

$[(\text{'s}, \text{'p}, \text{'f}) \text{ body}, \text{'f set}, (\text{'s}, \text{'p}) \text{ quadruple set},$   
 $\text{'s assn}, (\text{'s}, \text{'p}, \text{'f}) \text{ com}, \text{'s assn}] \Rightarrow \text{bool}$   
 $(\langle (\exists -, -/\vdash \prime_{t/-} (-) / -) \rangle [61, 60, 60, 1000, 20, 1000] 60)$

-hoaret-noAbr-emptyFaults::  
 $[(\text{'s','p','f'} \text{ body}, (\text{'s','p'} \text{ quadruple set}, \text{'s assn}, (\text{'s','p','f'} \text{ com}, \text{'s assn}]) \Rightarrow \text{bool}$   
 $(\langle (\exists -, -/\vdash_t (-) (-) -) \rangle [61, 60, 1000, 20, 1000] 60)$

-hoaret-emptyCtx-noAbr::  
 $[(\text{'s','p','f'} \text{ body}, \text{'f set}, \text{'s assn}, (\text{'s','p','f'} \text{ com}, \text{'s assn}]) \Rightarrow \text{bool}$   
 $(\langle (\exists -/\vdash_t \text{'_} (-) (-) -) \rangle [61, 60, 1000, 20, 1000] 60)$

-hoaret-emptyCtx-noAbr-emptyFaults::  
 $[(\text{'s','p','f'} \text{ body}, \text{'s assn}, (\text{'s','p','f'} \text{ com}, \text{'s assn}]) \Rightarrow \text{bool}$   
 $(\langle (\exists -/\vdash_t (-) (-) -) \rangle [61, 1000, 20, 1000] 60)$

## **syntax (ASCII)**

-hoarep-emptyFaults::  
 $[(\text{'s','p','f'} \text{ body}, (\text{'s','p'} \text{ quadruple set},$   
 $\text{'s assn}, (\text{'s','p','f'} \text{ com}, \text{'s assn}, \text{'s assn}]) \Rightarrow \text{bool}$   
 $(\langle (\exists -, -/|- (-) (-) -/-) \rangle [61, 60, 1000, 20, 1000, 1000] 60)$

-hoarep-emptyCtx::  
 $[(\text{'s','p','f'} \text{ body}, \text{'f set}, \text{'s assn}, (\text{'s','p','f'} \text{ com}, \text{'s assn}, \text{'s assn}]) \Rightarrow \text{bool}$   
 $(\langle (\exists -/|- \text{'_} - (-) (-) -/-) \rangle [61, 60, 1000, 20, 1000, 1000] 60)$

-hoarep-emptyCtx-emptyFaults::  
 $[(\text{'s','p','f'} \text{ body}, \text{'s assn}, (\text{'s','p','f'} \text{ com}, \text{'s assn}, \text{'s assn}]) \Rightarrow \text{bool}$   
 $(\langle (\exists -/|- (-) (-) -/-) \rangle [61, 1000, 20, 1000, 1000] 60)$

-hoarep-noAbr::  
 $[(\text{'s','p','f'} \text{ body}, (\text{'s','p'} \text{ quadruple set}, \text{'f set},$   
 $\text{'s assn}, (\text{'s','p','f'} \text{ com}, \text{'s assn}]) \Rightarrow \text{bool}$   
 $(\langle (\exists -, -/|- \text{'_} - (-) (-) -) \rangle [61, 60, 60, 1000, 20, 1000] 60)$

-hoarep-noAbr-emptyFaults::  
 $[(\text{'s','p','f'} \text{ body}, (\text{'s','p'} \text{ quadruple set}, \text{'s assn}, (\text{'s','p','f'} \text{ com}, \text{'s assn}]) \Rightarrow \text{bool}$   
 $(\langle (\exists -, -/|- (-) (-) -) \rangle [61, 60, 1000, 20, 1000] 60)$

-hoarep-emptyCtx-noAbr::  
 $[(\text{'s','p','f'} \text{ body}, \text{'f set}, \text{'s assn}, (\text{'s','p','f'} \text{ com}, \text{'s assn}]) \Rightarrow \text{bool}$   
 $(\langle (\exists -/|- \text{'_} - (-) (-) -) \rangle [61, 60, 1000, 20, 1000] 60)$

-hoarep-emptyCtx-noAbr-emptyFaults::  
 $[(\text{'s','p','f'} \text{ body}, \text{'s assn}, (\text{'s','p','f'} \text{ com}, \text{'s assn}]) \Rightarrow \text{bool}$   
 $(\langle (\exists -/|- (-) (-) -) \rangle [61, 1000, 20, 1000] 60)$

-hoaret-emptyFault::  
 $[(\text{'s','p','f'} \text{ body}, (\text{'s','p'} \text{ quadruple set},$   
 $\text{'s assn}, (\text{'s','p','f'} \text{ com}, \text{'s assn}, \text{'s assn}]) \Rightarrow \text{bool}$

$\langle \langle 3, - / - | - t \ (- / \ (-) / \ - / -) \rangle [61, 60, 1000, 20, 1000, 1000] 60 \rangle$

*-hoaret-emptyCtx::*

$[(\ 's, 'p, 'f \ body, 'f \ set, 's \ assn, (\ 's, 'p, 'f \ com, 's \ assn, 's \ assn) \Rightarrow \ bool$   
 $\langle \langle 3, - / - | - t' / - \ (- / \ (-) / \ - / -) \rangle [61, 60, 1000, 20, 1000, 1000] 60 \rangle$

*-hoaret-emptyCtx-emptyFaults::*

$[(\ 's, 'p, 'f \ body, 's \ assn, (\ 's, 'p, 'f \ com, 's \ assn, 's \ assn) \Rightarrow \ bool$   
 $\langle \langle 3, - / - | - t \ (- / \ (-) / \ - / -) \rangle [61, 1000, 20, 1000, 1000] 60 \rangle$

*-hoaret-noAbr::*

$[(\ 's, 'p, 'f \ body, (\ 's, 'p \ quadruple \ set, 'f \ set,$   
 $\ 's \ assn, (\ 's, 'p, 'f \ com, 's \ assn) \Rightarrow \ bool$   
 $\langle \langle 3, - / - | - t' / - \ (- / \ (-) / \ -) \rangle [61, 60, 60, 1000, 20, 1000] 60 \rangle$

*-hoaret-noAbr-emptyFaults::*

$[(\ 's, 'p, 'f \ body, (\ 's, 'p \ quadruple \ set, 's \ assn, (\ 's, 'p, 'f \ com, 's \ assn) \Rightarrow \ bool$   
 $\langle \langle 3, - / - | - t \ (- / \ (-) / \ -) \rangle [61, 60, 1000, 20, 1000] 60 \rangle$

*-hoaret-emptyCtx-noAbr::*

$[(\ 's, 'p, 'f \ body, 'f \ set, 's \ assn, (\ 's, 'p, 'f \ com, 's \ assn) \Rightarrow \ bool$   
 $\langle \langle 3, - / - | - t' / - \ (- / \ (-) / \ -) \rangle [61, 60, 1000, 20, 1000] 60 \rangle$

*-hoaret-emptyCtx-noAbr-emptyFaults::*

$[(\ 's, 'p, 'f \ body, 's \ assn, (\ 's, 'p, 'f \ com, 's \ assn) \Rightarrow \ bool$   
 $\langle \langle 3, - / - | - t \ (- / \ (-) / \ -) \rangle [61, 1000, 20, 1000] 60 \rangle$

## translations

$\Gamma \vdash P \ c \ Q, A \ == \ \Gamma \vdash / \{ \} \ P \ c \ Q, A$   
 $\Gamma \vdash /_F \ P \ c \ Q, A \ == \ \Gamma, \{ \} \vdash /_F \ P \ c \ Q, A$

$\Gamma, \Theta \vdash P \ c \ Q \ == \ \Gamma, \Theta \vdash / \{ \} \ P \ c \ Q$   
 $\Gamma, \Theta \vdash /_F \ P \ c \ Q \ == \ \Gamma, \Theta \vdash /_F \ P \ c \ Q, \{ \}$   
 $\Gamma, \Theta \vdash P \ c \ Q, A \ == \ \Gamma, \Theta \vdash / \{ \} \ P \ c \ Q, A$

$\Gamma \vdash P \ c \ Q \ == \ \Gamma \vdash / \{ \} \ P \ c \ Q$   
 $\Gamma \vdash /_F \ P \ c \ Q \ == \ \Gamma, \{ \} \vdash /_F \ P \ c \ Q$   
 $\Gamma \vdash /_F \ P \ c \ Q \ <= \ \Gamma \vdash /_F \ P \ c \ Q, \{ \}$   
 $\Gamma \vdash P \ c \ Q \ <= \ \Gamma \vdash P \ c \ Q, \{ \}$

$\Gamma \vdash_t \ P \ c \ Q, A \ == \ \Gamma \vdash_t / \{ \} \ P \ c \ Q, A$   
 $\Gamma \vdash_t /_F \ P \ c \ Q, A \ == \ \Gamma, \{ \} \vdash_t /_F \ P \ c \ Q, A$

$$\begin{aligned} \Gamma, \Theta \vdash_t P \ c \ Q & \equiv \Gamma, \Theta \vdash_{t/\{\}} P \ c \ Q \\ \Gamma, \Theta \vdash_{t/F} P \ c \ Q & \equiv \Gamma, \Theta \vdash_{t/F} P \ c \ Q, \{\} \\ \Gamma, \Theta \vdash_t P \ c \ Q, A & \equiv \Gamma, \Theta \vdash_{t/\{\}} P \ c \ Q, A \end{aligned}$$

$$\begin{aligned} \Gamma \vdash_t P \ c \ Q & \equiv \Gamma \vdash_{t/\{\}} P \ c \ Q \\ \Gamma \vdash_{t/F} P \ c \ Q & \equiv \Gamma, \{\} \vdash_{t/F} P \ c \ Q \\ \Gamma \vdash_{t/F} P \ c \ Q & \leq \Gamma \vdash_{t/F} P \ c \ Q, \{\} \\ \Gamma \vdash_t P \ c \ Q & \leq \Gamma \vdash_t P \ c \ Q, \{\} \end{aligned}$$

**term**  $\Gamma \vdash P \ c \ Q$   
**term**  $\Gamma \vdash P \ c \ Q, A$

**term**  $\Gamma \vdash_{/F} P \ c \ Q$   
**term**  $\Gamma \vdash_{/F} P \ c \ Q, A$

**term**  $\Gamma, \Theta \vdash P \ c \ Q$   
**term**  $\Gamma, \Theta \vdash_{/F} P \ c \ Q$

**term**  $\Gamma, \Theta \vdash P \ c \ Q, A$   
**term**  $\Gamma, \Theta \vdash_{/F} P \ c \ Q, A$

**term**  $\Gamma \vdash_t P \ c \ Q$   
**term**  $\Gamma \vdash_t P \ c \ Q, A$

**term**  $\Gamma \vdash_{t/F} P \ c \ Q$   
**term**  $\Gamma \vdash_{t/F} P \ c \ Q, A$

**term**  $\Gamma, \Theta \vdash P \ c \ Q$   
**term**  $\Gamma, \Theta \vdash_{t/F} P \ c \ Q$

**term**  $\Gamma, \Theta \vdash P \ c \ Q, A$   
**term**  $\Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$

**locale** *hoare* =  
**fixes**  $\Gamma :: ('s, 'p, 'f)$  *body*

**primrec** *assoc* ::  $('a \times 'b)$  *list*  $\Rightarrow 'a \Rightarrow 'b$   
**where**

*assoc* []  $x = \text{undefined}$  |  
*assoc* ( $p \# ps$ )  $x = (\text{if } \text{fst } p = x \text{ then } (\text{snd } p) \text{ else } \text{assoc } ps \ x)$

**lemma** *conjE-simp*:  $(P \wedge Q \Longrightarrow \text{PROP } R) \equiv (P \Longrightarrow Q \Longrightarrow \text{PROP } R)$   
*<proof>*

**lemma** *CollectInt-iff*:  $\{s. P s\} \cap \{s. Q s\} = \{s. P s \wedge Q s\}$   
*<proof>*

**lemma** *Compl-Collect*:  $\neg(\text{Collect } b) = \{x. \neg(b x)\}$   
*<proof>*

**lemma** *Collect-False*:  $\{s. \text{False}\} = \{\}$   
*<proof>*

**lemma** *Collect-True*:  $\{s. \text{True}\} = \text{UNIV}$   
*<proof>*

**lemma** *triv-All-eq*:  $\forall x. P \equiv P$   
*<proof>*

**lemma** *triv-Ex-eq*:  $\exists x. P \equiv P$   
*<proof>*

**lemma** *Ex-True*:  $\exists b. b$   
*<proof>*

**lemma** *Ex-False*:  $\exists b. \neg b$   
*<proof>*

**definition** *mex*::('a  $\Rightarrow$  bool)  $\Rightarrow$  bool  
**where** *mex* P = *Ex* P

**definition** *meq*::'a  $\Rightarrow$  'a  $\Rightarrow$  bool  
**where** *meq* s Z = (s = Z)

**lemma** *subset-unI1*:  $A \subseteq B \Longrightarrow A \subseteq B \cup C$   
*<proof>*

**lemma** *subset-unI2*:  $A \subseteq C \Longrightarrow A \subseteq B \cup C$   
*<proof>*

**lemma** *split-paired-UN*:  $(\bigcup p. (P p)) = (\bigcup a b. (P (a,b)))$   
*<proof>*

**lemma** *in-insert-hd*:  $f \in \text{insert } f X$   
*<proof>*

**lemma** *lookup-Some-in-dom*:  $\Gamma p = \text{Some } \text{bdy} \Longrightarrow p \in \text{dom } \Gamma$   
*<proof>*

**lemma** *unit-object*:  $(\forall u::\text{unit}. P u) = P ()$   
*<proof>*

**lemma** *unit-ex*:  $(\exists u::\text{unit}. P u) = P ()$

$\langle \text{proof} \rangle$

**lemma** *unit-meta*:  $(\bigwedge (u::\text{unit}). \text{PROP } P \ u) \equiv \text{PROP } P \ ()$   
 $\langle \text{proof} \rangle$

**lemma** *unit-UN*:  $(\bigcup z::\text{unit}. P \ z) = P \ ()$   
 $\langle \text{proof} \rangle$

**lemma** *subset-singleton-insert1*:  $y = x \implies \{y\} \subseteq \text{insert } x \ A$   
 $\langle \text{proof} \rangle$

**lemma** *subset-singleton-insert2*:  $\{y\} \subseteq A \implies \{y\} \subseteq \text{insert } x \ A$   
 $\langle \text{proof} \rangle$

**lemma** *in-Specs-simp*:  $(\forall x \in \bigcup Z. \{(P \ Z, p, Q \ Z, A \ Z)\}. \text{Prop } x) =$   
 $(\forall Z. \text{Prop } (P \ Z, p, Q \ Z, A \ Z))$   
 $\langle \text{proof} \rangle$

**lemma** *in-set-Un-simp*:  $(\forall x \in A \cup B. P \ x) = ((\forall x \in A. P \ x) \wedge (\forall x \in B. P \ x))$   
 $\langle \text{proof} \rangle$

**lemma** *split-all-conj*:  $(\forall x. P \ x \wedge Q \ x) = ((\forall x. P \ x) \wedge (\forall x. Q \ x))$   
 $\langle \text{proof} \rangle$

**lemma** *image-Un-single-simp*:  $f \ ` (\bigcup Z. \{P \ Z\}) = (\bigcup Z. \{f \ (P \ Z)\})$   
 $\langle \text{proof} \rangle$

**lemma** *measure-lex-prod-def'*:  
 $f \ <*\text{mlex}*\> r \equiv (\{(x,y). (x,y) \in \text{measure } f \vee f \ x=f \ y \wedge (x,y) \in r\})$   
 $\langle \text{proof} \rangle$

**lemma** *in-measure-iff*:  $(x,y) \in \text{measure } f = (f \ x < f \ y)$   
 $\langle \text{proof} \rangle$

**lemma** *in-lex-iff*:  
 $((a,b),(x,y)) \in r \ <*\text{lex}*\> s = ((a,x) \in r \vee (a=x \wedge (b,y) \in s))$   
 $\langle \text{proof} \rangle$

**lemma** *in-mlex-iff*:  
 $(x,y) \in f \ <*\text{mlex}*\> r = (f \ x < f \ y \vee (f \ x=f \ y \wedge (x,y) \in r))$   
 $\langle \text{proof} \rangle$

**lemma** *in-inv-image-iff*:  $(x,y) \in \text{inv-image } r \ f = ((f \ x, f \ y) \in r)$   
 $\langle \text{proof} \rangle$

This is actually the same as *wf-mlex*. However, this basic proof took me so long that I'm not willing to delete it.

**lemma** *wf-measure-lex-prod* [*simp,intro*]:  
**assumes** *wf-r*: *wf r*  
**shows** *wf* (*f* *<\*mlex\*>* *r*)  
*<proof>*

**lemmas** *all-imp-to-ex = all-simps* (5)

**lemma** *all-imp-eq-triv*:  $(\forall x. x = k \longrightarrow Q) = Q$   
 $(\forall x. k = x \longrightarrow Q) = Q$   
*<proof>*

**end**

## 13 State Space Template

**theory** *StateSpace* **imports** *Hoare*  
**begin**

**record** *'g state = globals::'g*

**definition**

*upd-globals:: ('g  $\Rightarrow$  'g)  $\Rightarrow$  ('g,'z) state-scheme  $\Rightarrow$  ('g,'z) state-scheme*

**where**

*upd-globals upd s = s(globals := upd (globals s))*

**named-theorems** *state-simp*

**lemma** *upd-globals-conv* [*state-simp*]: *upd-globals f = ( $\lambda s. s(globals := f (globals s))$ )*  
*<proof>*

**record** (*'g, 'l*) *state-locals = 'g state +*  
*locals :: 'l*

**type-synonym** (*'g, 'n, 'val*) *stateSP = ('g, 'n  $\Rightarrow$  'val) state-locals*

**type-synonym** (*'g, 'n, 'val, 'x*) *stateSP-scheme = ('g, 'n  $\Rightarrow$  'val, 'x) state-locals-scheme*

**end**

## 14 Alternative Small Step Semantics

**theory** *AlternativeSmallStep* **imports** *HoareTotalDef*  
**begin**

This is the small-step semantics, which is described and used in my PhD-

thesis [9]. It decomposes the statement into a list of statements and finally executes the head. So the redex is always the head of the list. The equivalence between termination (based on the big-step semantics) and the absence of infinite computations in this small-step semantics follows the same lines of reasoning as for the new small-step semantics. However, it is technically more involved since the configurations are more complicated. That's why I switched to the new small-step semantics in the "main trunk". I keep this alternative version and the important proofs in this theory, so that one can compare both approaches.

**14.1 Small-Step Computation:**  $\Gamma \vdash (cs, css, s) \rightarrow (cs', css', s')$

**type-synonym**  $(s, p, f)$  continuation =  $(s, p, f)$  com list  $\times$   $(s, p, f)$  com list

**type-synonym**  $(s, p, f)$  config =  
 $(s, p, f)$  com list  $\times$   $(s, p, f)$  continuation list  $\times$   $(s, f)$  xstate

**inductive** step:: $[(s, p, f)$  body,  $(s, p, f)$  config,  $(s, p, f)$  config]  $\Rightarrow$  bool  
 $(\vdash (- \rightarrow / -) \rangle [81, 81, 81] 100)$

**for**  $\Gamma :: (s, p, f)$  body

**where**

*Skip*:  $\Gamma \vdash (\text{Skip} \# cs, css, \text{Normal } s) \rightarrow (cs, css, \text{Normal } s)$

| *Guard*:  $s \in g \implies \Gamma \vdash (\text{Guard } f g \ c \# cs, css, \text{Normal } s) \rightarrow (c \# cs, css, \text{Normal } s)$

| *GuardFault*:  $s \notin g \implies \Gamma \vdash (\text{Guard } f g \ c \# cs, css, \text{Normal } s) \rightarrow (cs, css, \text{Fault } f)$

| *FaultProp*:  $\Gamma \vdash (c \# cs, css, \text{Fault } f) \rightarrow (cs, css, \text{Fault } f)$

| *FaultPropBlock*:  $\Gamma \vdash ([, (nrms, abrs) \# css, \text{Fault } f) \rightarrow (nrms, css, \text{Fault } f)$

| *AbruptProp*:  $\Gamma \vdash (c \# cs, css, \text{Abrupt } s) \rightarrow (cs, css, \text{Abrupt } s)$

| *ExitBlockNormal*:

$\Gamma \vdash ([, (nrms, abrs) \# css, \text{Normal } s) \rightarrow (nrms, css, \text{Normal } s)$

| *ExitBlockAbrupt*:

$\Gamma \vdash ([, (nrms, abrs) \# css, \text{Abrupt } s) \rightarrow (abrs, css, \text{Normal } s)$

| *Basic*:  $\Gamma \vdash (\text{Basic } f \# cs, css, \text{Normal } s) \rightarrow (cs, css, \text{Normal } (f s))$

| *Spec*:  $(s, t) \in r \implies \Gamma \vdash (\text{Spec } r \# cs, css, \text{Normal } s) \rightarrow (cs, css, \text{Normal } t)$

| *SpecStuck*:  $\forall t. (s, t) \notin r \implies \Gamma \vdash (\text{Spec } r \# cs, css, \text{Normal } s) \rightarrow (cs, css, \text{Stuck})$

| *Seq*:  $\Gamma \vdash (\text{Seq } c_1 \ c_2 \# cs, css, \text{Normal } s) \rightarrow (c_1 \# c_2 \# cs, css, \text{Normal } s)$

| *CondTrue*:  $s \in b \implies \Gamma \vdash (\text{Cond } b \ c_1 \ c_2 \# cs, css, \text{Normal } s) \rightarrow (c_1 \# cs, css, \text{Normal } s)$

| *CondFalse*:  $s \notin b \implies \Gamma \vdash (\text{Cond } b \ c_1 \ c_2 \# cs, css, \text{Normal } s) \rightarrow (c_2 \# cs, css, \text{Normal } s)$

| *WhileTrue*:  $\llbracket s \in b \rrbracket$   
 $\implies$   
 $\Gamma \vdash (\text{While } b \ c \# cs, css, Normal \ s) \rightarrow (c \# \text{While } b \ c \# cs, css, Normal \ s)$   
 | *WhileFalse*:  $\llbracket s \notin b \rrbracket$   
 $\implies$   
 $\Gamma \vdash (\text{While } b \ c \# cs, css, Normal \ s) \rightarrow (cs, css, Normal \ s)$

| *Call*:  $\Gamma \ p = \text{Some } bdy \implies$   
 $\Gamma \vdash (\text{Call } p \# cs, css, Normal \ s) \rightarrow ([bdy], (cs, \text{Throw} \# cs) \# cs, Normal \ s)$

| *CallUndefined*:  $\Gamma \ p = \text{None} \implies$   
 $\Gamma \vdash (\text{Call } p \# cs, css, Normal \ s) \rightarrow (cs, css, Stuck)$

| *StuckProp*:  $\Gamma \vdash (c \# cs, css, Stuck) \rightarrow (cs, css, Stuck)$   
 | *StuckPropBlock*:  $\Gamma \vdash (\llbracket \cdot \rrbracket, (nrms, abrs) \# cs, Stuck) \rightarrow (nrms, css, Stuck)$

| *DynCom*:  $\Gamma \vdash (\text{DynCom } c \# cs, css, Normal \ s) \rightarrow (c \ s \# cs, css, Normal \ s)$

| *Throw*:  $\Gamma \vdash (\text{Throw} \# cs, css, Normal \ s) \rightarrow (cs, css, Abrupt \ s)$   
 | *Catch*:  $\Gamma \vdash (\text{Catch } c_1 \ c_2 \# cs, css, Normal \ s) \rightarrow ([c_1], (cs, c_2 \# cs) \# cs, Normal \ s)$

**lemmas** *step-induct* = *step.induct* [*of* - (*c*, *css*, *s*) (*c'*, *css'*, *s'*), *split-format* (*complete*), *case-names*  
*Skip Guard GuardFault FaultProp FaultPropBlock AbruptProp ExitBlockNormal ExitBlockAbrupt*  
*Basic Spec SpecStuck Seq CondTrue CondFalse WhileTrue WhileFalse Call CallUndefined*  
*StuckProp StuckPropBlock DynCom Throw Catch, induct set*]

**inductive-cases** *step-elim-cases* [*cases set*]:

$\Gamma \vdash (c \# cs, css, Fault \ f) \rightarrow u$   
 $\Gamma \vdash (\llbracket \cdot \rrbracket, css, Fault \ f) \rightarrow u$   
 $\Gamma \vdash (c \# cs, css, Stuck) \rightarrow u$   
 $\Gamma \vdash (\llbracket \cdot \rrbracket, css, Stuck) \rightarrow u$   
 $\Gamma \vdash (c \# cs, css, Abrupt \ s) \rightarrow u$   
 $\Gamma \vdash (\llbracket \cdot \rrbracket, css, Abrupt \ s) \rightarrow u$   
 $\Gamma \vdash (\llbracket \cdot \rrbracket, css, Normal \ s) \rightarrow u$   
 $\Gamma \vdash (\text{Skip} \# cs, css, s) \rightarrow u$   
 $\Gamma \vdash (\text{Guard } f \ g \ c \# cs, css, s) \rightarrow u$   
 $\Gamma \vdash (\text{Basic } f \# cs, css, s) \rightarrow u$   
 $\Gamma \vdash (\text{Spec } r \# cs, css, s) \rightarrow u$   
 $\Gamma \vdash (\text{Seq } c_1 \ c_2 \# cs, css, s) \rightarrow u$   
 $\Gamma \vdash (\text{Cond } b \ c_1 \ c_2 \# cs, css, s) \rightarrow u$   
 $\Gamma \vdash (\text{While } b \ c \# cs, css, s) \rightarrow u$   
 $\Gamma \vdash (\text{Call } p \# cs, css, s) \rightarrow u$   
 $\Gamma \vdash (\text{DynCom } c \# cs, css, s) \rightarrow u$   
 $\Gamma \vdash (\text{Throw} \# cs, css, s) \rightarrow u$   
 $\Gamma \vdash (\text{Catch } c_1 \ c_2 \# cs, css, s) \rightarrow u$

**inductive-cases** *step-Normal-elim-cases* [cases set]:

$$\begin{aligned}
& \Gamma \vdash (c \# cs, css, Fault\ f) \rightarrow u \\
& \Gamma \vdash (\square, css, Fault\ f) \rightarrow u \\
& \Gamma \vdash (c \# cs, css, Stuck) \rightarrow u \\
& \Gamma \vdash (\square, css, Stuck) \rightarrow u \\
& \Gamma \vdash (\square, (nrms, abrs) \# css, Normal\ s) \rightarrow u \\
& \Gamma \vdash (\square, (nrms, abrs) \# css, Abrupt\ s) \rightarrow u \\
& \Gamma \vdash (Skip \# cs, css, Normal\ s) \rightarrow u \\
& \Gamma \vdash (Guard\ f\ g\ c \# cs, css, Normal\ s) \rightarrow u \\
& \Gamma \vdash (Basic\ f \# cs, css, Normal\ s) \rightarrow u \\
& \Gamma \vdash (Spec\ r \# cs, css, Normal\ s) \rightarrow u \\
& \Gamma \vdash (Seq\ c1\ c2 \# cs, css, Normal\ s) \rightarrow u \\
& \Gamma \vdash (Cond\ b\ c1\ c2 \# cs, css, Normal\ s) \rightarrow u \\
& \Gamma \vdash (While\ b\ c \# cs, css, Normal\ s) \rightarrow u \\
& \Gamma \vdash (Call\ p \# cs, css, Normal\ s) \rightarrow u \\
& \Gamma \vdash (DynCom\ c \# cs, css, Normal\ s) \rightarrow u \\
& \Gamma \vdash (Throw \# cs, css, Normal\ s) \rightarrow u \\
& \Gamma \vdash (Catch\ c1\ c2 \# cs, css, Normal\ s) \rightarrow u
\end{aligned}$$

**abbreviation**

$$step\text{-}rtrancl :: [(s', p', f)\ body, (s', p', f)\ config, (s', p', f)\ config] \Rightarrow bool$$

( $\vdash$  ( $- \rightarrow^*$  /  $-$ ) $\rangle$  [81,81,81] 100)

**where**

$$\Gamma \vdash cs0 \rightarrow^* cs1 \quad == (step\ \Gamma)^{**}\ cs0\ cs1$$

**abbreviation**

$$step\text{-}trancl :: [(s', p', f)\ body, (s', p', f)\ config, (s', p', f)\ config] \Rightarrow bool$$

( $\vdash$  ( $- \rightarrow^+$  /  $-$ ) $\rangle$  [81,81,81] 100)

**where**

$$\Gamma \vdash cs0 \rightarrow^+ cs1 \quad == (step\ \Gamma)^{++}\ cs0\ cs1$$

### 14.1.1 Structural Properties of Small Step Computations

**lemma** *Fault-app-steps*:  $\Gamma \vdash (cs @ xs, css, Fault\ f) \rightarrow^* (xs, css, Fault\ f)$

*<proof>*

**lemma** *Stuck-app-steps*:  $\Gamma \vdash (cs @ xs, css, Stuck) \rightarrow^* (xs, css, Stuck)$

*<proof>*

We can only append commands inside a block, if execution does not enter or exit a block.

**lemma** *app-step*:

**assumes** *step*:  $\Gamma \vdash (cs, css, s) \rightarrow (cs', css', t)$

**shows**  $css = css' \implies \Gamma \vdash (cs @ xs, css, s) \rightarrow (cs' @ xs, css', t)$

*<proof>*

We can append whole blocks, without interfering with the actual block. Outer blocks do not influence execution of inner blocks.

**lemma** *app-css-step*:

**assumes** *step*:  $\Gamma \vdash (cs, css, s) \rightarrow (cs', css', t)$

**shows**  $\Gamma \vdash (cs, css@xs, s) \rightarrow (cs', css'@xs, t)$

*<proof>*

*<ML>*

**lemma** *app-css-steps*:

**assumes** *step*:  $\Gamma \vdash (cs, css, s) \rightarrow^+ (cs', css', t)$

**shows**  $\Gamma \vdash (cs, css@xs, s) \rightarrow^+ (cs', css'@xs, t)$

*<proof>*

**lemma** *step-Cons'*:

**assumes** *step*:  $\Gamma \vdash (cs, css, s) \rightarrow (cs', css', t)$

**shows**

$\bigwedge c \text{ cs. } ccs = c\#cs \implies \exists css''. \text{ css}' = \text{css}''@css \wedge$

(if  $\text{css}'' = []$  then  $\exists p. \text{cs}' = p@cs$

else  $(\exists pnorm \text{ pabr. } \text{css}'' = [(pnorm@cs, pabr@cs)]))$

*<proof>*

**lemma** *step-Cons*:

**assumes** *step*:  $\Gamma \vdash (c\#cs, css, s) \rightarrow (cs', css', t)$

**shows**  $\exists pcss. \text{css}' = pcss@css \wedge$

(if  $pcss = []$  then  $\exists ps. \text{cs}' = ps@cs$

else  $(\exists pcs\text{-normal } pcs\text{-abrupt. } pcss = [(pcs\text{-normal}@cs, pcs\text{-abrupt}@cs)]))$

*<proof>*

**lemma** *step-Nil'*:

**assumes** *step*:  $\Gamma \vdash (cs, asscss, s) \rightarrow (cs', css', t)$

**shows**

$\bigwedge ass. [cs = []; \text{asscss} = \text{ass}@css; \text{ass} \neq Nil] \implies$

$\text{css}' = tl \text{ ass}@css \wedge$

(case *s* of

*Abrupt*  $s' \Rightarrow \text{cs}' = \text{snd} (hd \text{ ass}) \wedge t = \text{Normal } s'$

|  $- \Rightarrow \text{cs}' = \text{fst} (hd \text{ ass}) \wedge t = s$ )

*<proof>*

**lemma** *step-Nil*:

**assumes** *step*:  $\Gamma \vdash ([], \text{ass}@css, s) \rightarrow (cs', css', t)$

**assumes** *ass-not-Nil*:  $\text{ass} \neq []$

**shows**  $\text{css}' = tl \text{ ass}@css \wedge$

(case *s* of

*Abrupt*  $s' \Rightarrow \text{cs}' = \text{snd} (hd \text{ ass}) \wedge t = \text{Normal } s'$

|  $- \Rightarrow \text{cs}' = \text{fst} (hd \text{ ass}) \wedge t = s$ )

*<proof>*

**lemma** *step-Nil''*:

**assumes** *step*:  $\Gamma \vdash ([], (pcs\text{-normal}, pcs\text{-abrupt})\#pcss@css, s) \rightarrow (cs', pcss@css, t)$

**shows** (case  $s$  of  
 $Abrupt\ s' \Rightarrow cs'=pcs-abrupt \wedge t=Normal\ s'$   
 $| \ - \Rightarrow cs'=pcs-normal \wedge t=s$ )  
 $\langle proof \rangle$

**lemma** *drop-suffix-css-step'*:  
**assumes**  $step: \Gamma \vdash (cs, cssxs, s) \rightarrow (cs', css'xs, t)$   
**shows**  $\bigwedge_{css\ css'} xs. \llbracket cssxs = css@xs; css'xs = css'@xs \rrbracket$   
 $\implies \Gamma \vdash (cs, css, s) \rightarrow (cs', css', t)$   
 $\langle proof \rangle$

**lemma** *drop-suffix-css-step*:  
**assumes**  $step: \Gamma \vdash (cs, pcss@css, s) \rightarrow (cs', pcss'@css, t)$   
**shows**  $\Gamma \vdash (cs, pcss, s) \rightarrow (cs', pcss', t)$   
 $\langle proof \rangle$

**lemma** *drop-suffix-hd-css-step'*:  
**assumes**  $step: \Gamma \vdash (pcs, css, s) \rightarrow (cs', css'css, t)$   
**shows**  $\bigwedge p\ ps\ cs\ pnorm\ pabr. \llbracket pcs = p\#\ ps@cs; css'css = (pnorm@cs, pabr@cs)\#css \rrbracket$   
 $\implies \Gamma \vdash (p\#\ ps, css, s) \rightarrow (cs', (pnorm, pabr)\#css, t)$   
 $\langle proof \rangle$

**lemma** *drop-suffix-hd-css-step''*:  
**assumes**  $step: \Gamma \vdash (p\#\ ps@cs, css, s) \rightarrow (cs', (pnorm@cs, pabr@cs)\#css, t)$   
**shows**  $\Gamma \vdash (p\#\ ps, css, s) \rightarrow (cs', (pnorm, pabr)\#css, t)$   
 $\langle proof \rangle$

**lemma** *drop-suffix-hd-css-step*:  
**assumes**  $step: \Gamma \vdash (p\#\ ps@cs, css, s) \rightarrow (cs', [(pnorm@ps@cs, pabr@ps@cs)]@css, t)$   
**shows**  $\Gamma \vdash (p\#\ ps, css, s) \rightarrow (cs', [(pnorm@ps, pabr@ps)]@css, t)$   
 $\langle proof \rangle$

**lemma** *drop-suffix'*:  
**assumes**  $step: \Gamma \vdash (csss, css, s) \rightarrow (cs'xs, css', t)$   
**shows**  $\bigwedge xs\ cs\ cs'. \llbracket css = css'; csss = cs@xs; cs'xs = cs'@xs; cs \neq [] \rrbracket$   
 $\implies \Gamma \vdash (cs, css, s) \rightarrow (cs', css, t)$   
 $\langle proof \rangle$

**lemma** *drop-suffix*:  
**assumes**  $step: \Gamma \vdash (c\#\ cs@xs, css, s) \rightarrow (cs'@xs, css, t)$   
**shows**  $\Gamma \vdash (c\#\ cs, css, s) \rightarrow (cs', css, t)$   
 $\langle proof \rangle$

**lemma** *drop-suffix-same-css-step*:  
**assumes**  $step: \Gamma \vdash (cs@xs, css, s) \rightarrow (cs'@xs, css, t)$   
**assumes** *not-Nil*:  $cs \neq []$   
**shows**  $\Gamma \vdash (cs, xss, s) \rightarrow (cs', xss, t)$   
 $\langle proof \rangle$

**lemma** *Cons-change-css-step*:  
**assumes** *step*:  $\Gamma \vdash (cs, css, s) \rightarrow (cs', css' @ css, t)$   
**shows**  $\Gamma \vdash (cs, xss, s) \rightarrow (cs', css' @ xss, t)$   
 $\langle proof \rangle$

**lemma** *Nil-change-css-step*:  
**assumes** *step*:  $\Gamma \vdash (\square, ass @ css, s) \rightarrow (cs', ass' @ css, t)$   
**assumes** *ass-not-Nil*:  $ass \neq \square$   
**shows**  $\Gamma \vdash (\square, ass @ xss, s) \rightarrow (cs', ass' @ xss, t)$   
 $\langle proof \rangle$

### 14.1.2 Equivalence between Big and Small-Step Semantics

**lemma** *exec-impl-steps*:  
**assumes** *exec*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
**shows**  $\bigwedge cs\ css. \Gamma \vdash (c \# cs, css, s) \rightarrow^* (cs, css, t)$   
 $\langle proof \rangle$

**inductive** *execs*::[(*'s, 'p, 'f*) *body*, (*'s, 'p, 'f*) *com list*,  
(*'s, 'p, 'f*) *continuation list*,  
(*'s, 'f*) *xstate*, (*'s, 'f*) *xstate*]  $\Rightarrow$  *bool*  
( $\langle + \rangle \langle -, -, - \rangle \Rightarrow - \rightarrow [50, 50, 50, 50, 50] 50$ )  
**for**  $\Gamma ::$  (*'s, 'p, 'f*) *body*

**where**

*Nil*:  $\Gamma \vdash \langle \square, \square, s \rangle \Rightarrow s$

| *ExitBlockNormal*:  $\Gamma \vdash \langle nrms, css, Normal\ s \rangle \Rightarrow t$   
 $\Rightarrow$   
 $\Gamma \vdash \langle \square, (nrms, abrs) \# css, Normal\ s \rangle \Rightarrow t$

| *ExitBlockAbrupt*:  $\Gamma \vdash \langle abrs, css, Normal\ s \rangle \Rightarrow t$   
 $\Rightarrow$   
 $\Gamma \vdash \langle \square, (nrms, abrs) \# css, Abrupt\ s \rangle \Rightarrow t$

| *ExitBlockFault*:  $\Gamma \vdash \langle nrms, css, Fault\ f \rangle \Rightarrow t$   
 $\Rightarrow$   
 $\Gamma \vdash \langle \square, (nrms, abrs) \# css, Fault\ f \rangle \Rightarrow t$

| *ExitBlockStuck*:  $\Gamma \vdash \langle nrms, css, Stuck \rangle \Rightarrow t$   
 $\Rightarrow$   
 $\Gamma \vdash \langle \square, (nrms, abrs) \# css, Stuck \rangle \Rightarrow t$

| *Cons*:  $[\Gamma \vdash \langle c, s \rangle \Rightarrow t; \Gamma \vdash \langle cs, css, t \rangle \Rightarrow u]$   
 $\Rightarrow$   
 $\Gamma \vdash \langle c \# cs, css, s \rangle \Rightarrow u$

**inductive-cases** *execs-elim-cases* [*cases set*]:

$\Gamma \vdash \langle [], css, s \rangle \Rightarrow t$   
 $\Gamma \vdash \langle c \# cs, css, s \rangle \Rightarrow t$

$\langle ML \rangle$

**lemma** *execs-Fault-end*:

**assumes** *execs*:  $\Gamma \vdash \langle cs, css, s \rangle \Rightarrow t$  **shows**  $s = \text{Fault } f \Longrightarrow t = \text{Fault } f$   
 $\langle \text{proof} \rangle$

**lemma** *execs-Stuck-end*:

**assumes** *execs*:  $\Gamma \vdash \langle cs, css, s \rangle \Rightarrow t$  **shows**  $s = \text{Stuck} \Longrightarrow t = \text{Stuck}$   
 $\langle \text{proof} \rangle$

**theorem** *steps-impl-exec*:

**assumes** *steps*:  $\Gamma \vdash (cs, css, s) \rightarrow^* ([], [], t)$   
**shows**  $\Gamma \vdash \langle cs, css, s \rangle \Rightarrow t$   
 $\langle \text{proof} \rangle$

**theorem** *steps-impl-exec*:

**assumes** *steps*:  $\Gamma \vdash ([c], [], s) \rightarrow^* ([], [], t)$   
**shows**  $\Gamma \vdash \langle c, s \rangle \Rightarrow t$   
 $\langle \text{proof} \rangle$

**corollary** *steps-eq-exec*:  $\Gamma \vdash ([c], [], s) \rightarrow^* ([], [], t) = \Gamma \vdash \langle c, s \rangle \Rightarrow t$   
 $\langle \text{proof} \rangle$

## 14.2 Infinite Computations: $\text{inf } \Gamma \text{ } cs \text{ } css \text{ } s$

**definition** *inf* ::

$[(\text{'s}, \text{'p}, \text{'f}) \text{ body}, (\text{'s}, \text{'p}, \text{'f}) \text{ com list}, (\text{'s}, \text{'p}, \text{'f}) \text{ continuation list}, (\text{'s}, \text{'f}) \text{ xstate}]$   
 $\Rightarrow \text{bool}$

**where**  $\text{inf } \Gamma \text{ } cs \text{ } css \text{ } s = (\exists f. f \text{ } 0 = (cs, css, s) \wedge (\forall i. \Gamma \vdash f \text{ } i \rightarrow f(\text{Suc } i)))$

**lemma** *not-infI*:  $\llbracket \bigwedge f. \llbracket f \text{ } 0 = (cs, css, s); \bigwedge i. \Gamma \vdash f \text{ } i \rightarrow f(\text{Suc } i) \rrbracket \Longrightarrow \text{False} \rrbracket$   
 $\Longrightarrow \neg \text{inf } \Gamma \text{ } cs \text{ } css \text{ } s$

$\langle \text{proof} \rangle$

## 14.3 Equivalence of Termination and Absence of Infinite Computations

**inductive** *terminatess*::  $[(\text{'s}, \text{'p}, \text{'f}) \text{ body}, (\text{'s}, \text{'p}, \text{'f}) \text{ com list},$   
 $(\text{'s}, \text{'p}, \text{'f}) \text{ continuation list}, (\text{'s}, \text{'f}) \text{ xstate}] \Rightarrow \text{bool}$

$(\text{!} \vdash \text{!}, \text{!} \Downarrow \text{!} \rightarrow \text{!} [60, 20, 60] \text{ } 89)$

**for**  $\Gamma :: (\text{'s}, \text{'p}, \text{'f}) \text{ body}$

**where**

*Nil*:  $\Gamma \vdash [], [], \Downarrow s$

| *ExitBlockNormal*:  $\Gamma \vdash \text{nrms}, css \Downarrow \text{Normal } s$

$$\begin{aligned} & \implies \\ & \Gamma \vdash [], (nrms, abrs) \# css \Downarrow Normal \ s \\ | \textit{ExitBlockAbrupt}: & \Gamma \vdash abrs, css \Downarrow Normal \ s \\ & \implies \\ & \Gamma \vdash [], (nrms, abrs) \# css \Downarrow Abrupt \ s \\ | \textit{ExitBlockFault}: & \Gamma \vdash nrms, css \Downarrow Fault \ f \\ & \implies \\ & \Gamma \vdash [], (nrms, abrs) \# css \Downarrow Fault \ f \\ | \textit{ExitBlockStuck}: & \Gamma \vdash nrms, css \Downarrow Stuck \\ & \implies \\ & \Gamma \vdash [], (nrms, abrs) \# css \Downarrow Stuck \\ | \textit{Cons}: & \llbracket \Gamma \vdash c \downarrow s; (\forall t. \Gamma \vdash \langle c, s \rangle \Rightarrow t \longrightarrow \Gamma \vdash cs, css \Downarrow t) \rrbracket \\ & \implies \\ & \Gamma \vdash c \# cs, css \Downarrow s \end{aligned}$$

**inductive-cases** *terminatess-elim-cases* [cases set]:

$$\begin{aligned} & \Gamma \vdash [], css \Downarrow t \\ & \Gamma \vdash c \# cs, css \Downarrow t \end{aligned}$$

**lemma** *terminatess-Fault*:  $\bigwedge cs. \Gamma \vdash cs, css \Downarrow Fault \ f$   
 $\langle proof \rangle$

**lemma** *terminatess-Stuck*:  $\bigwedge cs. \Gamma \vdash cs, css \Downarrow Stuck$   
 $\langle proof \rangle$

**lemma** *Basic-terminates*:  $\Gamma \vdash Basic \ f \downarrow t$   
 $\langle proof \rangle$

**lemma** *step-preserves-terminations*:  
**assumes** *step*:  $\Gamma \vdash (cs, css, s) \rightarrow (cs', css', t)$   
**shows**  $\Gamma \vdash cs, css \Downarrow s \implies \Gamma \vdash cs', css' \Downarrow t$   
 $\langle proof \rangle$

$\langle ML \rangle$

**lemma** *steps-preserves-terminations*:  
**assumes** *steps*:  $\Gamma \vdash (cs, css, s) \rightarrow^* (cs', css', t)$   
**shows**  $\Gamma \vdash cs, css \Downarrow s \implies \Gamma \vdash cs', css' \Downarrow t$   
 $\langle proof \rangle$

**theorem** *steps-preserves-termination*:  
**assumes** *steps*:  $\Gamma \vdash ([c], [], s) \rightarrow^* (c' \# cs', css', t)$

**assumes** *term-c*:  $\Gamma \vdash c \downarrow s$   
**shows**  $\Gamma \vdash c' \downarrow t$   
 ⟨*proof*⟩

**lemma** *renumber'*:  
**assumes** *f*:  $\forall i. (a, f\ i) \in r^* \wedge (f\ i, f(Suc\ i)) \in r$   
**assumes** *a-b*:  $(a, b) \in r^*$   
**shows**  $b = f\ 0 \implies (\exists f. f\ 0 = a \wedge (\forall i. (f\ i, f(Suc\ i)) \in r))$   
 ⟨*proof*⟩

**lemma** *renumber*:  
 $\forall i. (a, f\ i) \in r^* \wedge (f\ i, f(Suc\ i)) \in r$   
 $\implies \exists f. f\ 0 = a \wedge (\forall i. (f\ i, f(Suc\ i)) \in r)$   
 ⟨*proof*⟩

**lemma** *not-inf-Fault'*:  
**assumes** *enum-step*:  $\forall i. \Gamma \vdash f\ i \rightarrow f(Suc\ i)$   
**shows**  $\bigwedge k\ cs. f\ k = (cs, css, Fault\ m) \implies False$   
 ⟨*proof*⟩

**lemma** *not-inf-Fault*:  
 $\neg\ inf\ \Gamma\ cs\ css\ (Fault\ m)$   
 ⟨*proof*⟩

**lemma** *not-inf-Stuck'*:  
**assumes** *enum-step*:  $\forall i. \Gamma \vdash f\ i \rightarrow f(Suc\ i)$   
**shows**  $\bigwedge k\ cs. f\ k = (cs, css, Stuck) \implies False$   
 ⟨*proof*⟩

**lemma** *not-inf-Stuck*:  
 $\neg\ inf\ \Gamma\ cs\ css\ Stuck$   
 ⟨*proof*⟩

**lemma** *last-butlast-app*:  
**assumes** *butlast*:  $butlast\ as = xs\ @\ butlast\ bs$   
**assumes** *not-Nil*:  $bs \neq []\ as \neq []$   
**assumes** *last*:  $fst\ (last\ as) = fst\ (last\ bs)\ snd\ (last\ as) = snd\ (last\ bs)$   
**shows**  $as = xs\ @\ bs$   
 ⟨*proof*⟩

**lemma** *last-butlast-tl*:  
**assumes** *butlast*:  $butlast\ bs = x\ \#\ butlast\ as$   
**assumes** *not-Nil*:  $bs \neq []\ as \neq []$   
**assumes** *last*:  $fst\ (last\ as) = fst\ (last\ bs)\ snd\ (last\ as) = snd\ (last\ bs)$

**shows**  $as = tl\ bs$   
 $\langle proof \rangle$

**locale**  $inf =$   
**fixes**  $CS:: ('s, 'p, 'f) config \Rightarrow ('s, 'p, 'f) com\ list$   
**and**  $CSS:: ('s, 'p, 'f) config \Rightarrow ('s, 'p, 'f) continuation\ list$   
**and**  $S:: ('s, 'p, 'f) config \Rightarrow ('s, 'f) xstate$   
**defines**  $CS-def : CS \equiv fst$   
**defines**  $CSS-def : CSS \equiv \lambda c. fst\ (snd\ c)$   
**defines**  $S-def : S \equiv \lambda c. snd\ (snd\ c)$

**lemma** (**in**  $inf$ )  $steps-hd-drop-suffix$ :  
**assumes**  $f-0: f\ 0 = (c\#\ cs, css, s)$   
**assumes**  $f-step: \forall i. \Gamma \vdash f(i) \rightarrow f(Suc\ i)$   
**assumes**  $not-finished: \forall i < k. \neg (CS\ (f\ i) = cs \wedge CSS\ (f\ i) = css)$   
**assumes**  $simul: \forall i \leq k.$   
 $(if\ pcss\ i = []\ then\ CSS\ (f\ i) = css \wedge CS\ (f\ i) = pcs\ i\ @\ cs$   
 $else\ CS\ (f\ i) = pcs\ i \wedge$   
 $CSS\ (f\ i) = butlast\ (pcss\ i)\ @$   
 $[(fst\ (last\ (pcss\ i))\ @\ cs, (snd\ (last\ (pcss\ i)))\ @\ cs)]\ @$   
 $css)$   
**defines**  $p \equiv \lambda i. (pcs\ i, pcss\ i, S\ (f\ i))$   
**shows**  $\forall i < k. \Gamma \vdash p\ i \rightarrow p\ (Suc\ i)$   
 $\langle proof \rangle$

**lemma**  $k-steps-to-rtrancl$ :  
**assumes**  $steps: \forall i < k. \Gamma \vdash p\ i \rightarrow p\ (Suc\ i)$   
**shows**  $\Gamma \vdash p\ 0 \rightarrow^* p\ k$   
 $\langle proof \rangle$

**lemma** (**in**  $inf$ )  $steps-hd-drop-suffix-finite$ :  
**assumes**  $f-0: f\ 0 = (c\#\ cs, css, s)$   
**assumes**  $f-step: \forall i. \Gamma \vdash f(i) \rightarrow f(Suc\ i)$   
**assumes**  $not-finished: \forall i < k. \neg (CS\ (f\ i) = cs \wedge CSS\ (f\ i) = css)$   
**assumes**  $simul: \forall i \leq k.$   
 $(if\ pcss\ i = []\ then\ CSS\ (f\ i) = css \wedge CS\ (f\ i) = pcs\ i\ @\ cs$   
 $else\ CS\ (f\ i) = pcs\ i \wedge$   
 $CSS\ (f\ i) = butlast\ (pcss\ i)\ @$   
 $[(fst\ (last\ (pcss\ i))\ @\ cs, (snd\ (last\ (pcss\ i)))\ @\ cs)]\ @$   
 $css)$   
**shows**  $\Gamma \vdash ([c], [], s) \rightarrow^* (pcs\ k, pcss\ k, S\ (f\ k))$   
 $\langle proof \rangle$

**lemma** (**in**  $inf$ )  $steps-hd-drop-suffix-infinite$ :  
**assumes**  $f-0: f\ 0 = (c\#\ cs, css, s)$   
**assumes**  $f-step: \forall i. \Gamma \vdash f(i) \rightarrow f(Suc\ i)$   
**assumes**  $not-finished: \forall i. \neg (CS\ (f\ i) = cs \wedge CSS\ (f\ i) = css)$

**assumes** *simul*:  $\forall i.$

(if  $pcss\ i = []$  then  $CSS\ (f\ i) = css \wedge CS\ (f\ i) = pcs\ i@cs$   
 else  $CS\ (f\ i) = pcs\ i \wedge$   
 $CSS\ (f\ i) = butlast\ (pcss\ i)@$   
 $[(fst\ (last\ (pcss\ i))@cs, (snd\ (last\ (pcss\ i)))@cs)]@$   
 $css$ )

**defines**  $p \equiv \lambda i. (pcs\ i, pcss\ i, S\ (f\ i))$

**shows**  $\Gamma \vdash p\ i \rightarrow p\ (Suc\ i)$

*<proof>*

**lemma** (in *inf*) *steps-hd-progress*:

**assumes** *f-0*:  $f\ 0 = (c\#cs, css, s)$

**assumes** *f-step*:  $\forall i. \Gamma \vdash f(i) \rightarrow f(Suc\ i)$

**assumes** *c-unfinished*:  $\forall i < k. \neg (CS\ (f\ i) = cs \wedge CSS\ (f\ i) = css)$

**shows**  $\forall i \leq k. (\exists pcs\ pcss.$

(if  $pcss = []$  then  $CSS\ (f\ i) = css \wedge CS\ (f\ i) = pcs@cs$   
 else  $CS\ (f\ i) = pcs \wedge$   
 $CSS\ (f\ i) = butlast\ pcss@$   
 $[(fst\ (last\ pcss)@cs, (snd\ (last\ pcss))@cs)]@$   
 $css)$ )

*<proof>*

**lemma** (in *inf*) *inf-progress*:

**assumes** *f-0*:  $f\ 0 = (c\#cs, css, s)$

**assumes** *f-step*:  $\forall i. \Gamma \vdash f(i) \rightarrow f(Suc\ i)$

**assumes** *unfinished*:  $\forall i. \neg ((CS\ (f\ i) = cs) \wedge (CSS\ (f\ i) = css))$

**shows**  $\exists pcs\ pcss.$

(if  $pcss = []$  then  $CSS\ (f\ i) = css \wedge CS\ (f\ i) = pcs@cs$   
 else  $CS\ (f\ i) = pcs \wedge$   
 $CSS\ (f\ i) = butlast\ pcss@$   
 $[(fst\ (last\ pcss)@cs, (snd\ (last\ pcss))@cs)]@$   
 $css)$ )

*<proof>*

**lemma** *skolemize1*:  $\forall x. P\ x \longrightarrow (\exists y. Q\ x\ y) \Longrightarrow \exists f. \forall x. P\ x \longrightarrow Q\ x\ (f\ x)$

*<proof>*

**lemma** *skolemize2*:  $\forall x. P\ x \longrightarrow (\exists y\ z. Q\ x\ y\ z) \Longrightarrow \exists f\ g. \forall x. P\ x \longrightarrow Q\ x\ (f\ x)\ (g\ x)$

*<proof>*

*<proof>*

**lemma** *skolemize2'*:  $\forall x. \exists y\ z. P\ x\ y\ z \Longrightarrow \exists f\ g. \forall x. P\ x\ (f\ x)\ (g\ x)$

*<proof>*

**theorem** (in *inf*) *inf-cases*:

**fixes**  $c::('s, 'p, 'f)\ com$

**assumes** *inf*:  $inf\ \Gamma\ (c\#cs)\ css\ s$

**shows**  $inf\ \Gamma\ [c]\ []\ s \vee (\exists t. \Gamma \vdash \langle c, s \rangle \Rightarrow t \wedge inf\ \Gamma\ cs\ css\ t)$

$\langle \text{proof} \rangle$

**lemma** *infE* [*consumes 1*]:

**assumes** *inf*:  $\text{inf } \Gamma (c \# cs) \text{ css } s$

**assumes** *cases*:  $\text{inf } \Gamma [c] [] s \implies P$

$\wedge t. [\Gamma \vdash \langle c, s \rangle \Rightarrow t; \text{inf } \Gamma cs \text{ css } t] \implies P$

**shows**  $P$

$\langle \text{proof} \rangle$

**lemma** *inf-Seq*:

$\text{inf } \Gamma (\text{Seq } c1 \ c2 \# cs) \text{ css } (\text{Normal } s) = \text{inf } \Gamma (c1 \# c2 \# cs) \text{ css } (\text{Normal } s)$

$\langle \text{proof} \rangle$

**lemma** *inf-WhileTrue*:

**assumes**  $b: s \in b$

**shows**  $\text{inf } \Gamma (\text{While } b \ c \# cs) \text{ css } (\text{Normal } s) =$

$\text{inf } \Gamma (c \# \text{While } b \ c \# cs) \text{ css } (\text{Normal } s)$

$\langle \text{proof} \rangle$

**lemma** *inf-Catch*:

$\text{inf } \Gamma (\text{Catch } c1 \ c2 \# cs) \text{ css } (\text{Normal } s) = \text{inf } \Gamma [c1] ((cs, c2 \# cs) \# css) (\text{Normal } s)$

$\langle \text{proof} \rangle$

**theorem** *terminates-impl-not-inf*:

**assumes** *termi*:  $\Gamma \vdash c \downarrow s$

**shows**  $\neg \text{inf } \Gamma [c] [] s$

$\langle \text{proof} \rangle$

**lemma** *terminatess-impl-not-inf*:

**assumes** *termi*:  $\Gamma \vdash cs, css \downarrow s$

**shows**  $\neg \text{inf } \Gamma cs \text{ css } s$

$\langle \text{proof} \rangle$

**lemma** *lem*:

$\forall y. r^{++} a \ y \longrightarrow P \ a \ \longrightarrow P \ y$

$\implies ((b, a) \in \{(y, x). P \ x \ \wedge \ r \ x \ y\}^+) = ((b, a) \in \{(y, x). P \ x \ \wedge \ r^{++} \ x \ y\})$

$\langle \text{proof} \rangle$

**corollary** *terminatess-impl-no-inf-chain*:

**assumes** *terminatess*:  $\Gamma \vdash cs, css \downarrow s$

**shows**  $\neg (\exists f. f \ 0 = (cs, css, s) \ \wedge \ (\forall i :: \text{nat}. \Gamma \vdash f \ i \ \rightarrow^+ f(\text{Suc } i)))$

$\langle \text{proof} \rangle$

**corollary** *terminates-impl-no-inf-chain*:

$\Gamma \vdash c \downarrow s \implies \neg (\exists f. f \ 0 = ([c], [], s) \ \wedge \ (\forall i :: \text{nat}. \Gamma \vdash f \ i \ \rightarrow^+ f(\text{Suc } i)))$

$\langle \text{proof} \rangle$

**definition**

$termi-call-steps :: ('s, 'p, 'f) body \Rightarrow (('s \times 'p) \times ('s \times 'p))set$   
**where**  
 $termi-call-steps \Gamma =$   
 $\{((t, q), (s, p)). \Gamma \vdash the (\Gamma p) \downarrow Normal s \wedge$   
 $(\exists css. \Gamma \vdash ([the (\Gamma p)], [], Normal s) \rightarrow^+ ([the (\Gamma q)], css, Normal t))\}$

Sequencing computations, or more exactly continuation stacks

**primrec**  $seq :: (nat \Rightarrow 'a list) \Rightarrow nat \Rightarrow 'a list$   
**where**  
 $seq\ css\ 0 = []$   
 $seq\ css\ (Suc\ i) = css\ i @ seq\ css\ i$

**theorem**  $wf-termi-call-steps: wf (termi-call-steps \Gamma)$   
 $\langle proof \rangle$

An alternative proof using Hilbert-choice instead of axiom of choice.

**theorem**  $wf (termi-call-steps \Gamma)$   
 $\langle proof \rangle$

**lemma**  $not-inf-implies-wf: assumes\ not-inf: \neg inf\ \Gamma\ cs\ css\ s$   
**shows**  $wf\ \{(c2, c1). \Gamma \vdash (cs, css, s) \rightarrow^* c1 \wedge \Gamma \vdash c1 \rightarrow c2\}$   
 $\langle proof \rangle$

**lemma**  $wf-implies-termi-reach:$   
**assumes**  $wf: wf\ \{(c2, c1). \Gamma \vdash (cs, css, s) \rightarrow^* c1 \wedge \Gamma \vdash c1 \rightarrow c2\}$   
**shows**  $\bigwedge cs1\ css1\ s1. [\Gamma \vdash (cs, css, s) \rightarrow^* c1; c1 = (cs1, css1, s1)] \Longrightarrow \Gamma \vdash cs1, css1 \downarrow s1$   
 $\langle proof \rangle$

**lemma**  $not-inf-impl-terminatess:$   
**assumes**  $not-inf: \neg inf\ \Gamma\ cs\ css\ s$   
**shows**  $\Gamma \vdash cs, css \downarrow s$   
 $\langle proof \rangle$

**lemma**  $not-inf-impl-terminates:$   
**assumes**  $not-inf: \neg inf\ \Gamma\ [c] []\ s$   
**shows**  $\Gamma \vdash c \downarrow s$   
 $\langle proof \rangle$

**theorem**  $terminatess-iff-not-inf:$   
 $\Gamma \vdash cs, css \downarrow s = (\neg inf\ \Gamma\ cs\ css\ s)$   
 $\langle proof \rangle$

**corollary**  $terminates-iff-not-inf:$   
 $\Gamma \vdash c \downarrow s = (\neg inf\ \Gamma\ [c] []\ s)$   
 $\langle proof \rangle$

## 14.4 Completeness of Total Correctness Hoare Logic

**lemma** *ConseqMGT*:

**assumes** *modif*:  $\forall Z::'a. \Gamma, \Theta \vdash_{t/F} (P' Z::'a \text{ assn}) \ c \ (Q' Z), (A' Z)$   
**assumes** *impl*:  $\bigwedge s. s \in P \implies s \in P' \ s \wedge (\forall t. t \in Q' \ s \longrightarrow t \in Q) \wedge$   
 $(\forall t. t \in A' \ s \longrightarrow t \in A)$

**shows**  $\Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$

*<proof>*

**lemma** *conseq-extract-state-indep-prop*:

**assumes** *state-indep-prop*:  $\forall s \in P. R$   
**assumes** *to-show*:  $R \implies \Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$

**shows**  $\Gamma, \Theta \vdash_{t/F} P \ c \ Q, A$

*<proof>*

**lemma** *Call-lemma'*:

**assumes** *Call-hyp*:

$\forall q \in \text{dom } \Gamma. \forall Z. \Gamma, \Theta \vdash_{t/F} \{s. s=Z \wedge \Gamma \vdash \langle \text{Call } q, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } '(-F)) \wedge$

$\Gamma \vdash \text{Call } q \downarrow \text{Normal } s \wedge ((s, q), (\sigma, p)) \in \text{termi-call-steps } \Gamma\}$

*(Call q)*

$\{t. \Gamma \vdash \langle \text{Call } q, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$

$\{t. \Gamma \vdash \langle \text{Call } q, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$

**shows**  $\bigwedge Z. \Gamma, \Theta \vdash_{t/F}$

$\{s. s=Z \wedge \Gamma \vdash \langle c, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } '(-F)) \wedge \Gamma \vdash \text{the } (\Gamma \ p) \downarrow \text{Normal}$   
 $\sigma \wedge$

$(\exists \text{cs } \text{css}. \Gamma \vdash ([\text{the } (\Gamma \ p)], [], \text{Normal } \sigma) \rightarrow^* (c \# \text{cs}, \text{css}, \text{Normal } s))\}$

*c*

$\{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$

$\{t. \Gamma \vdash \langle c, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$

*<proof>*

To prove a procedure implementation correct it suffices to assume only the procedure specifications of procedures that actually occur during evaluation of the body.

**lemma** *Call-lemma*:

**assumes**

*Call*:  $\forall q \in \text{dom } \Gamma. \forall Z. \Gamma, \Theta \vdash_{t/F}$

$\{s. s=Z \wedge \Gamma \vdash \langle \text{Call } q, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } '(-F)) \wedge$

$\Gamma \vdash \text{Call } q \downarrow \text{Normal } s \wedge ((s, q), (\sigma, p)) \in \text{termi-call-steps } \Gamma\}$

*(Call q)*

$\{t. \Gamma \vdash \langle \text{Call } q, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\},$

$\{t. \Gamma \vdash \langle \text{Call } q, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\}$

**shows**  $\bigwedge Z. \Gamma, \Theta \vdash_{t/F}$

$(\{\sigma\} \cap \{s. s=Z \wedge \Gamma \vdash \langle \text{the } (\Gamma \ p), \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } '(-F))$

$\wedge$

$\Gamma \vdash \text{the } (\Gamma \ p) \downarrow \text{Normal } s\}$

*the } (\Gamma \ p)*

$$\begin{aligned} & \{t. \Gamma \vdash \langle \text{the } (\Gamma \ p), \text{Normal } Z \rangle \Rightarrow \text{Normal } t\}, \\ & \{t. \Gamma \vdash \langle \text{the } (\Gamma \ p), \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\} \end{aligned}$$

$\langle \text{proof} \rangle$

**lemma** *Call-lemma-switch-Call-body*:

**assumes**

*call*:  $\forall q \in \text{dom } \Gamma. \forall Z. \Gamma, \Theta \vdash_{t/F}$

$$\begin{aligned} & \{s. s=Z \wedge \Gamma \vdash \langle \text{Call } q, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } '(-F)) \wedge \\ & \quad \Gamma \vdash \text{Call } q \downarrow \text{Normal } s \wedge ((s, q), (\sigma, p)) \in \text{termi-call-steps } \Gamma\} \\ & (\text{Call } q) \\ & \{t. \Gamma \vdash \langle \text{Call } q, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\}, \\ & \{t. \Gamma \vdash \langle \text{Call } q, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\} \end{aligned}$$

**assumes** *p-defined*:  $p \in \text{dom } \Gamma$

**shows**  $\wedge Z. \Gamma, \Theta \vdash_{t/F}$

$$\begin{aligned} & (\{\sigma\} \cap \{s. s=Z \wedge \Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } '(-F)) \wedge \\ & \quad \Gamma \vdash \text{Call } p \downarrow \text{Normal } s\}) \\ & \quad \text{the } (\Gamma \ p) \\ & \{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\}, \\ & \{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\} \end{aligned}$$

$\langle \text{proof} \rangle$

**lemma** *MGT-Call*:

$\forall p \in \text{dom } \Gamma. \forall Z.$

$$\begin{aligned} & \Gamma, \Theta \vdash_{t/F} \{s. s=Z \wedge \Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } '(-F)) \wedge \\ & \quad \Gamma \vdash (\text{Call } p) \downarrow \text{Normal } s\} \\ & (\text{Call } p) \\ & \{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\}, \\ & \{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\} \end{aligned}$$

$\langle \text{proof} \rangle$

**lemma** *CollInt-iff*:  $\{s. P \ s\} \cap \{s. Q \ s\} = \{s. P \ s \wedge Q \ s\}$

$\langle \text{proof} \rangle$

**lemma** *image-Un-conv*:  $f \ ' (\bigcup_{p \in \text{dom } \Gamma} \bigcup Z. \{x \ p \ Z\}) = (\bigcup_{p \in \text{dom } \Gamma} \bigcup Z. \{f \ (x \ p \ Z)\})$

$\langle \text{proof} \rangle$

Another proof of *MGT-Call*, maybe a little more readable

**lemma**

$\forall p \in \text{dom } \Gamma. \forall Z.$

$$\begin{aligned} & \Gamma, \{\} \vdash_{t/F} \{s. s=Z \wedge \Gamma \vdash \langle \text{Call } p, \text{Normal } s \rangle \Rightarrow \notin(\{\text{Stuck}\} \cup \text{Fault } '(-F)) \wedge \\ & \quad \Gamma \vdash (\text{Call } p) \downarrow \text{Normal } s\} \\ & (\text{Call } p) \\ & \{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Normal } t\}, \\ & \{t. \Gamma \vdash \langle \text{Call } p, \text{Normal } Z \rangle \Rightarrow \text{Abrupt } t\} \end{aligned}$$

$\langle \text{proof} \rangle$

**end**

**theory** *Simpl-Heap*  
**imports** *Main*  
**begin**

## 14.5 References

**definition** *ref* = (*UNIV*::*nat set*)

**typedef** *ref* = *ref* *<proof>*

**code-datatype** *Abs-ref*

**lemma** *finite-nat-ex-max*:  
  **assumes** *fin*: *finite* (*N*::*nat set*)  
  **shows**  $\exists m. \forall n \in N. n < m$   
*<proof>*

**lemma** *infinite-nat*:  $\neg$ *finite* (*UNIV*::*nat set*)  
*<proof>*

**lemma** *infinite-ref* [*simp,intro*]:  $\neg$ *finite* (*UNIV*::*ref set*)  
*<proof>*

**consts** *Null* :: *ref*

**definition** *new* :: *ref set*  $\Rightarrow$  *ref* **where**  
  *new* *A* = (*SOME* *a*. *a*  $\notin$  {*Null*}  $\cup$  *A*)

Constant *Null* can be defined later on. Conceptually *Null* and *new* are *fixes* of a locale with *finite* *A*  $\implies$  *new* *A*  $\notin$  *A*  $\cup$  {*Null*}. But since definitions relative to a locale do not yet work in Isabelle2005 we use this workaround to avoid lots of parameters in definitions.

**lemma** *new-notin* [*simp,intro*]:  
  *finite* *A*  $\implies$  *new* (*A*)  $\notin$  *A*  
*<proof>*

**lemma** *new-not-Null* [*simp,intro*]:  
  *finite* *A*  $\implies$  *new* (*A*)  $\neq$  *Null*  
*<proof>*

**end**

## 15 Paths and Lists in the Heap

**theory** *HeapList*

**imports** *Simpl-Heap*  
**begin**

Adapted from 'HOL/Hoare/Heap.thy'.

## 15.1 Paths in The Heap

**primrec**

$Path :: ref \Rightarrow (ref \Rightarrow ref) \Rightarrow ref \Rightarrow ref\ list \Rightarrow bool$

**where**

$Path\ x\ h\ y\ [] = (x = y) \mid$

$Path\ x\ h\ y\ (p\#\!ps) = (x = p \wedge x \neq Null \wedge Path\ (h\ x)\ h\ y\ ps)$

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

**lemma** *Path-not-Null-iff* [simp]:  $p \neq Null \implies$

$Path\ p\ h\ q\ as = (as = [] \wedge q = p \vee (\exists ps. as = p\#\!ps \wedge Path\ (h\ p)\ h\ q\ ps))$   
 <proof>

**lemma** *Path-append* [simp]:

$\bigwedge p. Path\ p\ f\ q\ (as@bs) = (\exists y. Path\ p\ f\ y\ as \wedge Path\ y\ f\ q\ bs)$   
 <proof>

**lemma** *notin-Path-update*[simp]:

$\bigwedge p. u \notin set\ ps \implies Path\ p\ (f(u := v))\ q\ ps = Path\ p\ f\ q\ ps$   
 <proof>

**lemma** *Path-upd-same* [simp]:

$Path\ p\ (f(p:=p))\ q\ qs =$   
 $((p=Null \wedge q=Null \wedge qs = []) \vee (p \neq Null \wedge q=p \wedge (\forall x \in set\ qs. x=p)))$   
 <proof>

*Path-upd-same* prevents  $p \neq Null \implies Path\ p\ (f(p := p))\ q\ qs = X$  from looping, because of *Path-not-Null-iff* and *fun-upd-apply*.

**lemma** *notin-Path-updateI* [intro]:

$\llbracket Path\ p\ h\ q\ ps ; r \notin set\ ps \rrbracket \implies Path\ p\ (h(r := y))\ q\ ps$   
 <proof>

**lemma** *Path-update-new* [simp]:  $\llbracket set\ ps \subseteq set\ alloc \rrbracket$

$\implies Path\ p\ (f(new\ (set\ alloc) := x))\ q\ ps = Path\ p\ f\ q\ ps$   
 <proof>

**lemma** *Null-notin-Path* [simp,intro]:

$\bigwedge p. Path\ p\ f\ q\ ps \implies Null \notin set\ ps$   
 <proof>

**lemma** *Path-snoc*:

$\llbracket Path\ p\ (f(a := q))\ a\ as ; a \neq Null \rrbracket \implies Path\ p\ (f(a := q))\ q\ (as @ [a])$

*<proof>*

## 15.2 Lists on The Heap

### 15.2.1 Relational Abstraction

#### definition

$List :: ref \Rightarrow (ref \Rightarrow ref) \Rightarrow ref\ list \Rightarrow bool$  **where**  
 $List\ p\ h\ ps = Path\ p\ h\ Null\ ps$

**lemma** *List-empty* [simp]:  $List\ p\ h\ [] = (p = Null)$

*<proof>*

**lemma** *List-cons* [simp]:  $List\ p\ h\ (a\#\ps) = (p = a \wedge p \neq Null \wedge List\ (h\ p)\ h\ ps)$

*<proof>*

**lemma** *List-Null* [simp]:  $List\ Null\ h\ ps = (ps = [])$

*<proof>*

**lemma** *List-not-Null* [simp]:  $p \neq Null \implies$

$List\ p\ h\ as = (\exists ps. as = p\#\ps \wedge List\ (h\ p)\ h\ ps)$

*<proof>*

**lemma** *Null-notin-List* [simp,intro]:  $\bigwedge p. List\ p\ h\ ps \implies Null \notin set\ ps$

*<proof>*

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

$\bigwedge p. q \notin set\ ps \implies List\ p\ (h(q := y))\ ps = List\ p\ h\ ps$

*<proof>*

**lemma** *List-upd-same-lemma*:  $\bigwedge p. p \neq Null \implies \neg List\ p\ (h(p := p))\ ps$

*<proof>*

**lemma** *List-upd-same* [simp]:  $List\ p\ (h(p:=p))\ ps = (p = Null \wedge ps = [])$

*<proof>*

*List-upd-same* prevents  $p \neq Null \implies List\ p\ (h(p := p))\ as = X$  from looping, because of *List-not-Null* and *fun-upd-apply*.

**lemma** *List-update-new* [simp]:  $\llbracket set\ ps \subseteq set\ alloc \rrbracket$

$\implies List\ p\ (h(new\ (set\ alloc) := x))\ ps = List\ p\ h\ ps$

*<proof>*

**lemma** *List-updateI* [intro]:

$\llbracket List\ p\ h\ ps; q \notin set\ ps \rrbracket \implies List\ p\ (h(q := y))\ ps$

*<proof>*

**lemma** *List-unique*:  $\bigwedge p\ bs. List\ p\ h\ as \implies List\ p\ h\ bs \implies as = bs$

*<proof>*

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

**lemma** *List-app*:  $\bigwedge p. List\ p\ h\ (as@bs) = (\exists y. Path\ p\ h\ y\ as \wedge List\ y\ h\ bs)$   
 ⟨proof⟩

**lemma** *List-hd-not-in-tl*[simp]:  $List\ (h\ p)\ h\ ps \implies p \notin set\ ps$   
 ⟨proof⟩

**lemma** *List-distinct*[simp]:  $\bigwedge p. List\ p\ h\ ps \implies distinct\ ps$   
 ⟨proof⟩

**lemma** *heap-eq-List-eq*:  
 $\bigwedge p. \forall x \in set\ ps. h\ x = g\ x \implies List\ p\ h\ ps = List\ p\ g\ ps$   
 ⟨proof⟩

**lemma** *heap-eq-ListI*:  
**assumes** *list*:  $List\ p\ h\ ps$   
**assumes** *hp-eq*:  $\forall x \in set\ ps. h\ x = g\ x$   
**shows**  $List\ p\ g\ ps$   
 ⟨proof⟩

**lemma** *heap-eq-ListII*:  
**assumes** *list*:  $List\ p\ h\ ps$   
**assumes** *hp-eq*:  $\forall x \in set\ ps. g\ x = h\ x$   
**shows**  $List\ p\ g\ ps$   
 ⟨proof⟩

The following lemmata are useful for the simplifier to instantiate bound variables in the assumptions resp. conclusion, using the uniqueness of the List predicate

**lemma** *conj-impl-simp*:  $(P \wedge Q \longrightarrow K) = (P \longrightarrow Q \longrightarrow K)$   
 ⟨proof⟩

**lemma** *List-unique-all-impl-simp* [simp]:  
 $List\ p\ h\ ps \implies (\forall ps. List\ p\ h\ ps \longrightarrow P\ ps) = P\ ps$   
 ⟨proof⟩

**lemma** *List-unique-ex-conj-simp* [simp]:  
 $List\ p\ h\ ps \implies (\exists ps. List\ p\ h\ ps \wedge P\ ps) = P\ ps$   
 ⟨proof⟩

### 15.3 Functional abstraction

**definition**

*islist* :: *ref*  $\Rightarrow$  (*ref*  $\Rightarrow$  *ref*)  $\Rightarrow$  *bool* **where**  
*islist* *p h* = ( $\exists$  *ps*. *List p h ps*)

**definition**

*list* :: *ref*  $\Rightarrow$  (*ref*  $\Rightarrow$  *ref*)  $\Rightarrow$  *ref list* **where**  
*list* *p h* = (*THE ps*. *List p h ps*)

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

**lemma** *List-islist [intro]*:

*List p h ps*  $\Longrightarrow$  *islist p h*  
 $\langle$ *proof* $\rangle$

**lemma** *List-list*:

*List p h ps*  $\Longrightarrow$  *list p h* = *ps*  
 $\langle$ *proof* $\rangle$

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

$\langle$ *proof* $\rangle$

**lemma** [*simp*]: *p*  $\neq$  *Null*  $\Longrightarrow$  *islist (h p) h* = *islist p h*

$\langle$ *proof* $\rangle$

**lemma** [*simp*]: *list Null h* = []

$\langle$ *proof* $\rangle$

**lemma** *list-Ref-conv[simp]*:

$\llbracket$ *islist (h p) h*; *p*  $\neq$  *Null*  $\rrbracket$   $\Longrightarrow$  *list p h* = *p* # *list (h p) h*  
 $\langle$ *proof* $\rangle$

**lemma** [*simp*]: *islist (h p) h*  $\Longrightarrow$  *p*  $\notin$  *set(list (h p) h)*

$\langle$ *proof* $\rangle$

**lemma** *list-upd-conv[simp]*:

*islist p h*  $\Longrightarrow$  *y*  $\notin$  *set(list p h)*  $\Longrightarrow$  *list p (h(y := q))* = *list p h*  
 $\langle$ *proof* $\rangle$

**lemma** *islist-upd[simp]*:

*islist p h*  $\Longrightarrow$  *y*  $\notin$  *set(list p h)*  $\Longrightarrow$  *islist p (h(y := q))*  
 $\langle$ *proof* $\rangle$

**lemma** *list-distinct[simp]*: *islist p h*  $\Longrightarrow$  *distinct (list p h)*

$\langle$ *proof* $\rangle$

**lemma** *Null-notin-list [simp,intro]*: *islist p h*  $\Longrightarrow$  *Null*  $\notin$  *set (list p h)*

$\langle$ *proof* $\rangle$

**end**

**theory** *Generalise* **imports** *HOL-Statespace.DistinctTreeProver*  
**begin**

**lemma** *protectReft*:  $PROP\ Pure.prop\ (PROP\ C) \implies PROP\ Pure.prop\ (PROP\ C)$   
*<proof>*

**lemma** *protectImp*:  
**assumes**  $i$ :  $PROP\ Pure.prop\ (PROP\ P \implies PROP\ Q)$   
**shows**  $PROP\ Pure.prop\ (PROP\ Pure.prop\ P \implies PROP\ Pure.prop\ Q)$   
*<proof>*

**lemma** *generaliseConj*:  
**assumes**  $i1$ :  $PROP\ Pure.prop\ (PROP\ Pure.prop\ (Trueprop\ P) \implies PROP\ Pure.prop\ (Trueprop\ Q))$   
**assumes**  $i2$ :  $PROP\ Pure.prop\ (PROP\ Pure.prop\ (Trueprop\ P') \implies PROP\ Pure.prop\ (Trueprop\ Q'))$   
**shows**  $PROP\ Pure.prop\ (PROP\ Pure.prop\ (Trueprop\ (P \wedge P')) \implies (PROP\ Pure.prop\ (Trueprop\ (Q \wedge Q'))))$   
*<proof>*

**lemma** *generaliseAll*:  
**assumes**  $i$ :  $PROP\ Pure.prop\ (\bigwedge s. PROP\ Pure.prop\ (Trueprop\ (P\ s)) \implies PROP\ Pure.prop\ (Trueprop\ (Q\ s)))$   
**shows**  $PROP\ Pure.prop\ (PROP\ Pure.prop\ (Trueprop\ (\forall s. P\ s)) \implies PROP\ Pure.prop\ (Trueprop\ (\forall s. Q\ s)))$   
*<proof>*

**lemma** *generalise-all*:  
**assumes**  $i$ :  $PROP\ Pure.prop\ (\bigwedge s. PROP\ Pure.prop\ (PROP\ P\ s) \implies PROP\ Pure.prop\ (PROP\ Q\ s))$   
**shows**  $PROP\ Pure.prop\ ((PROP\ Pure.prop\ (\bigwedge s. PROP\ P\ s)) \implies (PROP\ Pure.prop\ (\bigwedge s. PROP\ Q\ s)))$   
*<proof>*

**lemma** *generaliseTrans*:  
**assumes**  $i1$ :  $PROP\ Pure.prop\ (PROP\ P \implies PROP\ Q)$   
**assumes**  $i2$ :  $PROP\ Pure.prop\ (PROP\ Q \implies PROP\ R)$   
**shows**  $PROP\ Pure.prop\ (PROP\ P \implies PROP\ R)$   
*<proof>*

**lemma** *meta-spec*:  
**assumes**  $\bigwedge x. PROP\ P\ x$   
**shows**  $PROP\ P\ x$  *<proof>*

**lemma** *meta-spec-protect*:

**assumes**  $g: \bigwedge x. PROP P x$

**shows**  $PROP Pure.prop (PROP P x)$

$\langle proof \rangle$

**lemma** *generaliseImp*:

**assumes**  $i: PROP Pure.prop (PROP Pure.prop (Trueprop P) \implies PROP Pure.prop (Trueprop Q))$

**shows**  $PROP Pure.prop (PROP Pure.prop (Trueprop (X \longrightarrow P)) \implies PROP Pure.prop (Trueprop (X \longrightarrow Q)))$

$\langle proof \rangle$

**lemma** *generaliseEx*:

**assumes**  $i: PROP Pure.prop (\bigwedge s. PROP Pure.prop (Trueprop (P s)) \implies PROP Pure.prop (Trueprop (Q s)))$

**shows**  $PROP Pure.prop (PROP Pure.prop (Trueprop (\exists s. P s)) \implies PROP Pure.prop (Trueprop (\exists s. Q s)))$

$\langle proof \rangle$

**lemma** *generaliseReft*:  $PROP Pure.prop (PROP Pure.prop (Trueprop P) \implies PROP Pure.prop (Trueprop P))$

$\langle proof \rangle$

**lemma** *generaliseReft'*:  $PROP Pure.prop (PROP P \implies PROP P)$

$\langle proof \rangle$

**lemma** *generaliseAllShift*:

**assumes**  $i: PROP Pure.prop (\bigwedge s. P \implies Q s)$

**shows**  $PROP Pure.prop (PROP Pure.prop (Trueprop P) \implies PROP Pure.prop (Trueprop (\forall s. Q s)))$

$\langle proof \rangle$

**lemma** *generalise-allShift*:

**assumes**  $i: PROP Pure.prop (\bigwedge s. PROP P \implies PROP Q s)$

**shows**  $PROP Pure.prop (PROP Pure.prop (PROP P) \implies PROP Pure.prop (\bigwedge s. PROP Q s))$

$\langle proof \rangle$

**lemma** *generaliseImpl*:

**assumes**  $i: PROP Pure.prop (PROP Pure.prop P \implies PROP Pure.prop Q)$

**shows**  $PROP Pure.prop ((PROP Pure.prop (PROP X \implies PROP P)) \implies (PROP Pure.prop (PROP X \implies PROP Q)))$

$\langle proof \rangle$

$\langle ML \rangle$

end

## 16 Facilitating the Hoare Logic

```
theory Vcg
imports StateSpace HOL-Statespace.StateSpaceLocale Generalise
keywords procedures hoarestate :: thy-defn
begin
```

```
axiomatization NoBody::('s,'p,'f) com
```

$\langle ML \rangle$

Variables of the programming language are represented as components of a record. To avoid cluttering up the namespace of Isabelle with lots of typical variable names, we append a unusual suffix at the end of each name by parsing

```
definition list-multsel:: 'a list  $\Rightarrow$  nat list  $\Rightarrow$  'a list (infixl  $\langle !! \rangle$  100)
  where xs !! ns = map (nth xs) ns
```

```
definition list-multupd:: 'a list  $\Rightarrow$  nat list  $\Rightarrow$  'a list  $\Rightarrow$  'a list
  where list-multupd xs ns ys = foldl ( $\lambda$ xs (n,v). xs[n:=v]) xs (zip ns ys)
```

**nonterminal** *lmupbinds* and *lmupbind*

**syntax**

— multiple list update

```
-lmupbind:: ['a, 'a]  $\Rightarrow$  lmupbind ( $\langle \langle 2 \text{ -} [:=] / \text{ -} \rangle \rangle$ )
:: lmupbind  $\Rightarrow$  lmupbinds ( $\langle \langle \text{ -} \rangle \rangle$ )
-lmupbinds:: [lmupbind, lmupbinds]  $\Rightarrow$  lmupbinds ( $\langle \langle \text{ -} / \text{ -} \rangle \rangle$ )
-LMUpdate:: ['a, lmupbinds]  $\Rightarrow$  'a ( $\langle \langle \text{ -} / [(-)] \rangle \rangle$  [900,0] 900)
```

**syntax-consts**

```
-lmupbind -lmupbinds -LMUpdate == list-multupd
```

**translations**

```
-LMUpdate xs (-lmupbinds b bs) == -LMUpdate (-LMUpdate xs b) bs
xs[is[:=]ys] == CONST list-multupd xs is ys
```

### 16.1 Some Fancy Syntax

reverse application

```
definition rapp:: 'a  $\Rightarrow$  ('a  $\Rightarrow$  'b)  $\Rightarrow$  'b (infixr  $\langle | \rangle$  60)
  where rapp x f = f x
```

**nonterminal**

*newinit* and



-Cond :: 'a bexp => ('a,'p,'f) com => ('a,'p,'f) com => ('a,'p,'f) com  
(⟨(OIF (-)/ (2THEN / -)/ (2ELSE -)/ FI)⟩ [0, 0, 0] 71)  
-Cond-no-else:: 'a bexp => ('a,'p,'f) com => ('a,'p,'f) com  
(⟨(OIF (-)/ (2THEN / -)/ FI)⟩ [0, 0] 71)  
-GuardedCond :: 'a bexp => ('a,'p,'f) com => ('a,'p,'f) com => ('a,'p,'f) com  
(⟨(OIF<sub>g</sub> (-)/ (2THEN -)/ (2ELSE -)/ FI)⟩ [0, 0, 0] 71)  
-GuardedCond-no-else:: 'a bexp => ('a,'p,'f) com => ('a,'p,'f) com  
(⟨(OIF<sub>g</sub> (-)/ (2THEN -)/ FI)⟩ [0, 0] 71)  
-While-inv-var :: 'a bexp => 'a assn ⇒ ('a × 'a) set ⇒ bdy  
⇒ ('a,'p,'f) com  
(⟨(0WHILE (-)/ INV (-)/ VAR (-) /-)⟩ [25, 0, 0, 81] 71)  
-WhileFix-inv-var :: 'a bexp => pptrn ⇒ ('z ⇒ 'a assn) ⇒  
('z ⇒ ('a × 'a) set) ⇒ bdy  
⇒ ('a,'p,'f) com  
(⟨(0WHILE (-)/ FIX -./ INV (-)/ VAR (-) /-)⟩ [25, 0, 0, 0, 81] 71)  
-WhileFix-inv :: 'a bexp => pptrn ⇒ ('z ⇒ 'a assn) ⇒ bdy  
⇒ ('a,'p,'f) com  
(⟨(0WHILE (-)/ FIX -./ INV (-) /-)⟩ [25, 0, 0, 81] 71)  
-GuardedWhileFix-inv-var :: 'a bexp => pptrn ⇒ ('z ⇒ 'a assn) ⇒  
('z ⇒ ('a × 'a) set) ⇒ bdy  
⇒ ('a,'p,'f) com  
(⟨(0WHILE<sub>g</sub> (-)/ FIX -./ INV (-)/ VAR (-) /-)⟩ [25, 0, 0, 0, 81] 71)  
-GuardedWhileFix-inv-var-hook :: 'a bexp ⇒ ('z ⇒ 'a assn) ⇒  
('z ⇒ ('a × 'a) set) ⇒ bdy  
⇒ ('a,'p,'f) com  
-GuardedWhileFix-inv :: 'a bexp => pptrn ⇒ ('z ⇒ 'a assn) ⇒ bdy  
⇒ ('a,'p,'f) com  
(⟨(0WHILE<sub>g</sub> (-)/ FIX -./ INV (-) /-)⟩ [25, 0, 0, 81] 71)  
-GuardedWhile-inv-var::  
'a bexp => 'a assn ⇒ ('a × 'a) set ⇒ bdy ⇒ ('a,'p,'f) com  
(⟨(0WHILE<sub>g</sub> (-)/ INV (-)/ VAR (-) /-)⟩ [25, 0, 0, 81] 71)  
-While-inv :: 'a bexp => 'a assn ⇒ bdy ⇒ ('a,'p,'f) com  
(⟨(0WHILE (-)/ INV (-) /-)⟩ [25, 0, 81] 71)  
-GuardedWhile-inv :: 'a bexp => 'a assn ⇒ ('a,'p,'f) com => ('a,'p,'f) com  
(⟨(0WHILE<sub>g</sub> (-)/ INV (-) /-)⟩ [25, 0, 81] 71)  
-While :: 'a bexp => bdy => ('a,'p,'f) com  
(⟨(0WHILE (-) /-)⟩ [25, 81] 71)  
-GuardedWhile :: 'a bexp => bdy => ('a,'p,'f) com  
(⟨(0WHILE<sub>g</sub> (-) /-)⟩ [25, 81] 71)  
-While-guard :: grds => 'a bexp => bdy => ('a,'p,'f) com  
(⟨(0WHILE (-/!→ (1-)) /-)⟩ [1000,25,81] 71)  
-While-guard-inv:: grds ⇒ 'a bexp ⇒ 'a assn ⇒ bdy ⇒ ('a,'p,'f) com  
(⟨(0WHILE (-/!→ (1-)) INV (-) /-)⟩ [1000,25,0,81] 71)  
-While-guard-inv-var:: grds ⇒ 'a bexp ⇒ 'a assn ⇒ ('a × 'a) set  
⇒ bdy ⇒ ('a,'p,'f) com  
(⟨(0WHILE (-/!→ (1-)) INV (-)/ VAR (-) /-)⟩ [1000,25,0,0,81] 71)  
-WhileFix-guard-inv-var:: grds ⇒ 'a bexp ⇒ pptrn ⇒ ('z ⇒ 'a assn) ⇒ ('z ⇒ ('a × 'a) set)  
⇒ bdy ⇒ ('a,'p,'f) com

$\langle \langle 0\text{WHILE } (-/\mapsto (1-)) \text{ FIX } -./ \text{ INV } (-)/ \text{ VAR } (-) /- \rangle \rangle [1000,25,0,0,0,81]$   
 71)

-WhileFix-guard-inv::  $\text{grds} \Rightarrow 'a \text{ bexp} \Rightarrow \text{pttrn} \Rightarrow ('z \Rightarrow 'a \text{ assn})$   
 $\Rightarrow \text{bdy} \Rightarrow ('a, 'p, 'f) \text{ com}$   
 $\langle \langle 0\text{WHILE } (-/\mapsto (1-)) \text{ FIX } -./ \text{ INV } (-)/- \rangle \rangle [1000,25,0,0,81] \text{ 71}$

-Try-Catch::  $('a, 'p, 'f) \text{ com} \Rightarrow ('a, 'p, 'f) \text{ com} \Rightarrow ('a, 'p, 'f) \text{ com}$   
 $\langle \langle 0\text{TRY } (-)/ (2\text{CATCH } -)/ \text{ END} \rangle \rangle [0,0] \text{ 71}$

-DoPre ::  $('a, 'p, 'f) \text{ com} \Rightarrow ('a, 'p, 'f) \text{ com}$   
 -Do ::  $('a, 'p, 'f) \text{ com} \Rightarrow \text{bdy} \langle \langle 2\text{DO} / (-) / \text{OD} \rangle \rangle [0] \text{ 1000}$   
 -Lab::  $'a \text{ bexp} \Rightarrow ('a, 'p, 'f) \text{ com} \Rightarrow \text{bdy}$   
 $\langle \langle -./ \rightarrow [1000,71] \text{ 81} \rangle \rangle$   
 ::  $\text{bdy} \Rightarrow ('a, 'p, 'f) \text{ com} \langle \langle - \rangle \rangle$

-Spec::  $\text{pttrn} \Rightarrow 's \text{ set} \Rightarrow ('s, 'p, 'f) \text{ com} \Rightarrow 's \text{ set} \Rightarrow 's \text{ set} \Rightarrow ('s, 'p, 'f) \text{ com}$   
 $\langle \langle \text{ANNO } -./ (-)/ -./- \rangle \rangle [0,1000,20,1000,1000] \text{ 60}$   
 -SpecNoAbrupt::  $\text{pttrn} \Rightarrow 's \text{ set} \Rightarrow ('s, 'p, 'f) \text{ com} \Rightarrow 's \text{ set} \Rightarrow ('s, 'p, 'f) \text{ com}$   
 $\langle \langle \text{ANNO } -./ (-)/ - \rangle \rangle [0,1000,20,1000] \text{ 60}$

-LemAnno::  $'n \Rightarrow ('s, 'p, 'f) \text{ com} \Rightarrow ('s, 'p, 'f) \text{ com}$   
 $\langle \langle 0 \text{ LEMMA } (-)/ - \text{ END} \rangle \rangle [1000,0] \text{ 71}$

-locnoinit ::  $\text{ident} \Rightarrow \text{locinit} \langle \langle - \rangle \rangle$   
 -locinit ::  $[\text{ident}, 'a] \Rightarrow \text{locinit} \langle \langle 2' - :==/ - \rangle \rangle$   
 ::  $\text{locinit} \Rightarrow \text{locinits} \langle \langle - \rangle \rangle$

-locinits ::  $[\text{locinit}, \text{locinits}] \Rightarrow \text{locinits} \langle \langle -, / - \rangle \rangle$   
 -Loc::  $[\text{locinits}, ('s, 'p, 'f) \text{ com}] \Rightarrow ('s, 'p, 'f) \text{ com}$   
 $\langle \langle 2 \text{ LOC } -; / (-) \text{ COL} \rangle \rangle [0,0] \text{ 71}$

-Switch::  $('s \Rightarrow 'v) \Rightarrow \text{switchcases} \Rightarrow ('s, 'p, 'f) \text{ com}$   
 $\langle \langle 0 \text{ SWITCH } (-)/ - \text{ END} \rangle \rangle [22,0] \text{ 71}$

-switchcase::  $'v \text{ set} \Rightarrow ('s, 'p, 'f) \text{ com} \Rightarrow \text{switchcase} \langle \langle - \Rightarrow / - \rangle \rangle$   
 -switchcasesSingle ::  $\text{switchcase} \Rightarrow \text{switchcases} \langle \langle - \rangle \rangle$   
 -switchcasesCons::  $\text{switchcase} \Rightarrow \text{switchcases} \Rightarrow \text{switchcases}$   
 $\langle \langle - / | - \rangle \rangle$

-Basic::  $\text{basicblock} \Rightarrow ('s, 'p, 'f) \text{ com} \langle \langle 0\text{BASIC} / (-) / \text{END} \rangle \rangle [22] \text{ 71}$   
 -BasicBlock::  $\text{basics} \Rightarrow \text{basicblock} \langle \langle - \rangle \rangle$   
 -BAssign ::  $'b \Rightarrow 'b \Rightarrow \text{basic} \langle \langle - :==/ - \rangle \rangle [30, 30] \text{ 23}$   
 ::  $\text{basic} \Rightarrow \text{basics} \langle \langle - \rangle \rangle$

-basics ::  $[\text{basic}, \text{basics}] \Rightarrow \text{basics} \langle \langle -, / - \rangle \rangle$

### syntax (ASCII)

-Assert ::  $'a \Rightarrow 'a \text{ set} \langle \langle \{ \} | - \rangle \rangle [0] \text{ 1000}$   
 -AssertState ::  $\text{idt} \Rightarrow 'a \Rightarrow 'a \text{ set} \langle \langle \{ | - . - \} \rangle \rangle [1000,0] \text{ 1000}$   
 -While-guard ::  $\text{grds} \Rightarrow 'a \text{ bexp} \Rightarrow \text{bdy} \Rightarrow ('a, 'p, 'f) \text{ com}$   
 $\langle \langle 0\text{WHILE } (-|\rightarrow /- ) /- \rangle \rangle [0,0,1000] \text{ 71}$   
 -While-guard-inv::  $\text{grds} \Rightarrow 'a \text{ bexp} \Rightarrow 'a \text{ assn} \Rightarrow \text{bdy} \Rightarrow ('a, 'p, 'f) \text{ com}$   
 $\langle \langle 0\text{WHILE } (-|\rightarrow /- ) \text{ INV } (-) /- \rangle \rangle [0,0,0,1000] \text{ 71}$   
 -guards ::  $\text{grds} \Rightarrow ('s, 'p, 'f) \text{ com} \Rightarrow ('s, 'p, 'f) \text{ com} \langle \langle (-|\rightarrow -) \rangle \rangle [60, 21] \text{ 23}$

### syntax (output)

-hidden-grds ::  $\text{grds} \langle \langle \dots \rangle \rangle$

## translations

*-Do*  $c \Rightarrow c$   
*b* ·  $c \Rightarrow \text{CONST condCatch } c \text{ } b \text{ SKIP}$   
*b* ·  $(\text{-DoPre } c) \Leftarrow \text{CONST condCatch } c \text{ } b \text{ SKIP}$   
*l* ·  $(\text{CONST whileAnnoG } gs \text{ } b \text{ } I \text{ } V \text{ } c) \Leftarrow l \cdot (\text{-DoPre } (\text{CONST whileAnnoG } gs \text{ } b \text{ } I \text{ } V \text{ } c))$   
*l* ·  $(\text{CONST whileAnno } b \text{ } I \text{ } V \text{ } c) \Leftarrow l \cdot (\text{-DoPre } (\text{CONST whileAnno } b \text{ } I \text{ } V \text{ } c))$   
 $\text{CONST condCatch } c \text{ } b \text{ SKIP} \Leftarrow (\text{-DoPre } (\text{CONST condCatch } c \text{ } b \text{ SKIP}))$   
*-Do*  $c \Leftarrow \text{-DoPre } c$   
*c*;; *d* ==  $\text{CONST Seq } c \text{ } d$   
*-guarantee*  $g \Rightarrow (\text{CONST True}, g)$   
*-guaranteeStrip*  $g == \text{CONST guaranteeStripPair } (\text{CONST True}) \text{ } g$   
*-grd*  $g \Rightarrow (\text{CONST False}, g)$   
*-grds*  $g \text{ } gs \Rightarrow g\#gs$   
*-last-grd*  $g \Rightarrow [g]$   
*-guards*  $gs \text{ } c == \text{CONST guards } gs \text{ } c$

$\{ |s. P| \} \quad == \{ | \text{-antiquoteCur}((=) \text{ } s) \wedge P \text{ } | \}$   
 $\{ |b| \} \quad == \text{CONST Collect } (\text{-quote } b)$   
 $\text{IF } b \text{ THEN } c1 \text{ ELSE } c2 \text{ FI} \Rightarrow \text{CONST Cond } \{ |b| \} \text{ } c1 \text{ } c2$   
 $\text{IF } b \text{ THEN } c1 \text{ FI} \quad == \text{IF } b \text{ THEN } c1 \text{ ELSE SKIP FI}$   
 $\text{IF}_g \text{ } b \text{ THEN } c1 \text{ FI} \quad == \text{IF}_g \text{ } b \text{ THEN } c1 \text{ ELSE SKIP FI}$

*-While-inv-var*  $b \text{ } I \text{ } V \text{ } c \quad == \text{CONST whileAnno } \{ |b| \} \text{ } I \text{ } V \text{ } c$   
*-While-inv-var*  $b \text{ } I \text{ } V \text{ } (\text{-DoPre } c) \Leftarrow \text{CONST whileAnno } \{ |b| \} \text{ } I \text{ } V \text{ } c$   
*-While-inv*  $b \text{ } I \text{ } c \quad == \text{-While-inv-var } b \text{ } I \text{ } (\text{CONST undefined}) \text{ } c$   
*-While*  $b \text{ } c \quad == \text{-While-inv } b \text{ } \{ | \text{CONST undefined} | \} \text{ } c$

*-While-guard-inv-var*  $gs \text{ } b \text{ } I \text{ } V \text{ } c \quad == \text{CONST whileAnnoG } gs \text{ } \{ |b| \} \text{ } I \text{ } V \text{ } c$

*-While-guard-inv*  $gs \text{ } b \text{ } I \text{ } c \quad == \text{-While-guard-inv-var } gs \text{ } b \text{ } I \text{ } (\text{CONST undefined})$   
*c*  
*-While-guard*  $gs \text{ } b \text{ } c \quad == \text{-While-guard-inv } gs \text{ } b \text{ } \{ | \text{CONST undefined} | \} \text{ } c$

*-GuardedWhile-inv*  $b \text{ } I \text{ } c \quad == \text{-GuardedWhile-inv-var } b \text{ } I \text{ } (\text{CONST undefined}) \text{ } c$   
*-GuardedWhile*  $b \text{ } c \quad == \text{-GuardedWhile-inv } b \text{ } \{ | \text{CONST undefined} | \} \text{ } c$

$\text{TRY } c1 \text{ CATCH } c2 \text{ END} \quad == \text{CONST Catch } c1 \text{ } c2$   
 $\text{ANNO } s. P \text{ } c \text{ } Q, A \Rightarrow \text{CONST specAnno } (\lambda s. P) (\lambda s. c) (\lambda s. Q) (\lambda s. A)$   
 $\text{ANNO } s. P \text{ } c \text{ } Q == \text{ANNO } s. P \text{ } c \text{ } Q, \{ \}$

*-WhileFix-inv-var*  $b \text{ } z \text{ } I \text{ } V \text{ } c \Rightarrow \text{CONST whileAnnoFix } \{ |b| \} (\lambda z. I) (\lambda z. V) (\lambda z. c)$   
*-WhileFix-inv-var*  $b \text{ } z \text{ } I \text{ } V \text{ } (\text{-DoPre } c) \Leftarrow \text{-WhileFix-inv-var } \{ |b| \} \text{ } z \text{ } I \text{ } V \text{ } c$   
*-WhileFix-inv*  $b \text{ } z \text{ } I \text{ } c == \text{-WhileFix-inv-var } b \text{ } z \text{ } I \text{ } (\text{CONST undefined}) \text{ } c$

*-GuardedWhileFix-inv*  $b \text{ } z \text{ } I \text{ } c == \text{-GuardedWhileFix-inv-var } b \text{ } z \text{ } I \text{ } (\text{CONST undefined}) \text{ } c$

-GuardedWhileFix-inv-var  $b z I V c \Rightarrow$   
     -GuardedWhileFix-inv-var-hook  $\{|b|\} (\lambda z. I) (\lambda z. V) (\lambda z. c)$

-WhileFix-guard-inv-var  $gs b z I V c \Rightarrow$   
     CONST whileAnnoGFix  $gs \{|b|\} (\lambda z. I) (\lambda z. V)$   
 ( $\lambda z. c$ )

-WhileFix-guard-inv-var  $gs b z I V (-DoPre c) \Leftarrow$   
     -WhileFix-guard-inv-var  $gs \{|b|\} z I V c$

-WhileFix-guard-inv  $gs b z I c \Leftarrow$  -WhileFix-guard-inv-var  $gs b z I (CONST$   
 undefined)  $c$

LEMMA  $x c END \Leftarrow$  CONST lem  $x c$

**translations**

(-switchcase  $V c) \Rightarrow (V, c)$   
 (-switchcasesSingle  $b) \Rightarrow [b]$   
 (-switchcasesCons  $b bs) \Rightarrow CONST Cons b bs$   
 (-Switch  $v vs) \Rightarrow CONST switch (-quote v) vs$

$\langle ML \rangle$

**syntax**

-faccess  $:: 'ref \Rightarrow ('ref \Rightarrow 'v) \Rightarrow 'v$   
 ( $\langle \rightarrow \rightarrow \rangle [65,1000] 100$ )

**syntax (ASCII)**

-faccess  $:: 'ref \Rightarrow ('ref \Rightarrow 'v) \Rightarrow 'v$   
 ( $\langle \rightarrow \rightarrow \rangle [65,1000] 100$ )

**translations**

$p \rightarrow f \quad \Rightarrow \quad f p$   
 $g \rightarrow (-antiquoteCur f) \Leftarrow$  -antiquoteCur  $f g$

**nonterminal *par* and *pars* and *actuals***

**syntax**

-par  $:: 'a \Rightarrow par$   $\langle \rightarrow \rangle$   
      $:: par \Rightarrow pars$   $\langle \rightarrow \rangle$   
 -pars  $:: [par, pars] \Rightarrow pars$   $\langle \rightarrow, / \rightarrow \rangle$   
 -actuals  $:: pars \Rightarrow actuals$   $\langle '(-)' \rangle$   
 -actuals-empty  $:: actuals$   $\langle '()' \rangle$

**syntax** -Call  $:: 'p \Rightarrow actuals \Rightarrow (('a, string, 'f) com) \langle CALL \rightarrow [1000,1000] 21$   
 -GuardedCall  $:: 'p \Rightarrow actuals \Rightarrow (('a, string, 'f) com) \langle CALL_g \rightarrow [1000,1000]$   
 21)  
 -CallAss  $:: 'a \Rightarrow 'p \Rightarrow actuals \Rightarrow (('a, string, 'f) com)$

$(\langle \cdot := \text{CALL} \rightarrow [30,1000,1000] \ 21)$   
 $\text{-Call-exn} :: 'p \Rightarrow \text{actuals} \Rightarrow (('a, \text{string}, 'f) \text{ com}) (\langle \text{CALL}_e \rightarrow [1000,1000] \ 21)$   
 $\text{-CallAss-exn} :: 'a \Rightarrow 'p \Rightarrow \text{actuals} \Rightarrow (('a, \text{string}, 'f) \text{ com})$   
 $(\langle \cdot := \text{CALL}_e \rightarrow [30,1000,1000] \ 21)$   
 $\text{-Proc} :: 'p \Rightarrow \text{actuals} \Rightarrow (('a, \text{string}, 'f) \text{ com}) (\langle \text{PROC} \rightarrow 21)$   
 $\text{-ProcAss} :: 'a \Rightarrow 'p \Rightarrow \text{actuals} \Rightarrow (('a, \text{string}, 'f) \text{ com})$   
 $(\langle \cdot := \text{PROC} \rightarrow [30,1000,1000] \ 21)$   
 $\text{-GuardedCallAss} :: 'a \Rightarrow 'p \Rightarrow \text{actuals} \Rightarrow (('a, \text{string}, 'f) \text{ com})$   
 $(\langle \cdot := \text{CALL}_g \rightarrow [30,1000,1000] \ 21)$   
 $\text{-DynCall} :: 'p \Rightarrow \text{actuals} \Rightarrow (('a, \text{string}, 'f) \text{ com}) (\langle \text{DYNCALL} \rightarrow [1000,1000]$   
 $21)$   
 $\text{-GuardedDynCall} :: 'p \Rightarrow \text{actuals} \Rightarrow (('a, \text{string}, 'f) \text{ com}) (\langle \text{DYNCALL}_g \rightarrow$   
 $[1000,1000] \ 21)$   
 $\text{-DynCallAss} :: 'a \Rightarrow 'p \Rightarrow \text{actuals} \Rightarrow (('a, \text{string}, 'f) \text{ com})$   
 $(\langle \cdot := \text{DYNCALL} \rightarrow [30,1000,1000] \ 21)$   
 $\text{-DynCall-exn} :: 'p \Rightarrow \text{actuals} \Rightarrow (('a, \text{string}, 'f) \text{ com}) (\langle \text{DYNCALL}_e \rightarrow$   
 $[1000,1000] \ 21)$   
 $\text{-DynCallAss-exn} :: 'a \Rightarrow 'p \Rightarrow \text{actuals} \Rightarrow (('a, \text{string}, 'f) \text{ com})$   
 $(\langle \cdot := \text{DYNCALL}_e \rightarrow [30,1000,1000] \ 21)$   
 $\text{-GuardedDynCallAss} :: 'a \Rightarrow 'p \Rightarrow \text{actuals} \Rightarrow (('a, \text{string}, 'f) \text{ com})$   
 $(\langle \cdot := \text{DYNCALL}_g \rightarrow [30,1000,1000] \ 21)$   
  
 $\text{-Bind} :: ['s \Rightarrow 'v, \text{idt}, 'v \Rightarrow ('s, 'p, 'f) \text{ com}] \Rightarrow ('s, 'p, 'f) \text{ com}$   
 $(\langle \cdot \gg \cdot / \rightarrow [22,1000,21] \ 21)$   
 $\text{-bseq} :: ('s, 'p, 'f) \text{ com} \Rightarrow ('s, 'p, 'f) \text{ com} \Rightarrow ('s, 'p, 'f) \text{ com}$   
 $(\langle \cdot \gg / \rightarrow [22, 21] \ 21)$   
 $\text{-FCall} :: ['p, \text{actuals}, \text{idt}, (('a, \text{string}, 'f) \text{ com})] \Rightarrow (('a, \text{string}, 'f) \text{ com})$   
 $(\langle \text{CALL} \gg \cdot / \rightarrow [1000,1000,1000,21] \ 21)$

### translations

$\text{-Bind } e \ i \ c == \text{CONST bind } (-\text{quote } e) (\lambda i. c)$   
 $\text{-FCall } p \ \text{acts } i \ c == \text{-FCall } p \ \text{acts } (\lambda i. c)$   
 $\text{-bseq } c \ d == \text{CONST bseq } c \ d$

### nonterminal modifyargs

#### syntax

$\text{-may-modify} :: ['a, 'a, \text{modifyargs}] \Rightarrow \text{bool}$   
 $(\langle \cdot \text{ may'-only'-modify'-globals - in } [-] \rangle [100,100,0] \ 100)$   
 $\text{-may-not-modify} :: ['a, 'a] \Rightarrow \text{bool}$   
 $(\langle \cdot \text{ may'-not'-modify'-globals } \rightarrow [100,100] \ 100)$   
 $\text{-may-modify-empty} :: ['a, 'a] \Rightarrow \text{bool}$   
 $(\langle \cdot \text{ may'-only'-modify'-globals - in } [] \rangle [100,100] \ 100)$   
 $\text{-modifyargs} :: [\text{id}, \text{modifyargs}] \Rightarrow \text{modifyargs } (\langle \cdot / \rightarrow)$   
 $:: \text{id} \Rightarrow \text{modifyargs} \quad (\langle \cdot \rightarrow)$

### translations

$s$  may-only-modify-globals  $Z$  in  $\square \Rightarrow s$  may-not-modify-globals  $Z$

**definition**  $Let'$ :: [ $'a$ ,  $'a \Rightarrow 'b$ ]  $\Rightarrow 'b$   
where  $Let' = Let$

$\langle ML \rangle$

### syntax

- $Measure$ :: ( $'a \Rightarrow nat$ )  $\Rightarrow ('a \times 'a)$  set  
    ( $\langle MEASURE \rightarrow [22] 1$ )  
- $Mlex$ :: ( $'a \Rightarrow nat$ )  $\Rightarrow ('a \times 'a)$  set  $\Rightarrow ('a \times 'a)$  set  
    (**infixr**  $\langle *MLEX* \rangle 30$ )

### syntax-consts

- $Measure == measure$  and  
- $Mlex == mlex-prod$

### translations

$MEASURE f \quad \Rightarrow (CONST measure) (-quote f)$   
 $f \langle *MLEX* \rangle r \quad \Rightarrow (-quote f) \langle *mlex* \rangle r$

$\langle ML \rangle$

end

## 17 Examples using the Verification Environment

**theory**  $VcgEx$  imports  $../HeapList ../Vcg$  begin

Some examples, especially the single-step Isar proofs are taken from `HOL/Isar_examples/HoareEx.th`

### 17.1 State Spaces

First of all we provide a store of program variables that occur in the programs considered later. Slightly unexpected things may happen when attempting to work with undeclared variables.

**record**  $'g vars = 'g state +$   
     $A-' :: nat$   
     $I-' :: nat$   
     $M-' :: nat$   
     $N-' :: nat$

$R-' :: nat$   
 $S-' :: nat$   
 $B-' :: bool$   
 $Arr-' :: nat\ list$   
 $Abr-' :: string$

We decorate the state components in the record with the suffix  $-'$ , to avoid cluttering the namespace with the simple names that could no longer be used for logical variables otherwise.

We will first consider programs without procedures, later on we will regard procedures without global variables and finally we will get the full pictures: mutually recursive procedures with global variables (including heap).

## 17.2 Basic Examples

We look at few trivialities involving assignment and sequential composition, in order to get an idea of how to work with our formulation of Hoare Logic.

Using the basic rule directly is a bit cumbersome.

**lemma**  $\Gamma \vdash \{ |'N = 5| \} 'N ::= 2 * 'N \{ |'N = 10| \}$   
*<proof>*

If we refer to components (variables) of the state-space of the program we always mark these with  $'$ . It is the acute-symbol and is present on most keyboards. So all program variables are marked with the acute and all logical variables are not. The assertions of the Hoare tuple are ordinary Isabelle sets. As we usually want to refer to the state space in the assertions, we provide special brackets for them. They can be written as  $\{ | \ | \}$  in ASCII or  $\{ | \}$  with symbols. Internally marking variables has two effects. First of all we refer to the implicit state and secondary we get rid of the suffix  $-'$ . So the assertion  $\{ |'N = 5| \}$  internally gets expanded to  $\{ s. N-' s = 5 \}$  written in ordinary set comprehension notation of Isabelle. It describes the set of states where the  $N-'$  component is equal to  $5$ .

Certainly we want the state modification already done, e.g. by simplification. The *vcs* method performs the basic state update for us; we may apply the Simplifier afterwards to achieve “obvious” consequences as well.

**lemma**  $\Gamma \vdash \{ True \} 'N ::= 10 \{ |'N = 10| \}$   
*<proof>*

**lemma**  $\Gamma \vdash \{ 2 * 'N = 10 \} 'N ::= 2 * 'N \{ |'N = 10| \}$   
*<proof>*

**lemma**  $\Gamma \vdash \{ |'N = 5| \} 'N ::= 2 * 'N \{ |'N = 10| \}$   
*<proof>*

**lemma**  $\Gamma \vdash \{\! \{ 'N + 1 = a + 1 \} \! \} 'N ::= 'N + 1 \{\! \{ 'N = a + 1 \} \! \}$   
 $\langle proof \rangle$

**lemma**  $\Gamma \vdash \{\! \{ 'N = a \} \! \} 'N ::= 'N + 1 \{\! \{ 'N = a + 1 \} \! \}$   
 $\langle proof \rangle$

**lemma**  $\Gamma \vdash \{\! \{ a = a \wedge b = b \} \! \} 'M ::= a;; 'N ::= b \{\! \{ 'M = a \wedge 'N = b \} \! \}$   
 $\langle proof \rangle$

**lemma**  $\Gamma \vdash \{\! \{ True \} \! \} 'M ::= a;; 'N ::= b \{\! \{ 'M = a \wedge 'N = b \} \! \}$   
 $\langle proof \rangle$

**lemma**  $\Gamma \vdash \{\! \{ 'M = a \wedge 'N = b \} \! \}$   
 $\quad 'I ::= 'M;; 'M ::= 'N;; 'N ::= 'I$   
 $\quad \{\! \{ 'M = b \wedge 'N = a \} \! \}$   
 $\langle proof \rangle$

We can also perform verification conditions generation step by step by using the *vcg-step* method.

**lemma**  $\Gamma \vdash \{\! \{ 'M = a \wedge 'N = b \} \! \}$   
 $\quad 'I ::= 'M;; 'M ::= 'N;; 'N ::= 'I$   
 $\quad \{\! \{ 'M = b \wedge 'N = a \} \! \}$   
 $\langle proof \rangle$

It is important to note that statements like the following one can only be proven for each individual program variable. Due to the extra-logical nature of record fields, we cannot formulate a theorem relating record selectors and updates schematically.

**lemma**  $\Gamma \vdash \{\! \{ 'N = a \} \! \} 'N ::= 'N \{\! \{ 'N = a \} \! \}$   
 $\langle proof \rangle$

**lemma**  $\Gamma \vdash \{s. x-'s = a\} (Basic (\lambda s. x-'update (x-'s) s)) \{s. x-'s = a\}$   
 $\langle proof \rangle$

In the following assignments we make use of the consequence rule in order to achieve the intended precondition. Certainly, the *vcg* method is able to handle this case, too.

**lemma**  $\Gamma \vdash \{\! \{ 'M = 'N \} \! \} 'M ::= 'M + 1 \{\! \{ 'M \neq 'N \} \! \}$   
 $\langle proof \rangle$

**lemma**  $\Gamma \vdash \{\! \{ 'M = 'N \} \! \} 'M ::= 'M + 1 \{\! \{ 'M \neq 'N \} \! \}$   
 $\langle proof \rangle$

**lemma**  $\Gamma \vdash \{\! \{ 'M = 'N \} \! \} 'M ::= 'M + 1 \{\! \{ 'M \neq 'N \} \! \}$

*<proof>*

### 17.3 Multiplication by Addition

We now do some basic examples of actual WHILE programs. This one is a loop for calculating the product of two natural numbers, by iterated addition. We first give detailed structured proof based on single-step Hoare rules.

**lemma**  $\Gamma \vdash \{ \prime M = 0 \wedge \prime S = 0 \}$   
    WHILE  $\prime M \neq a$   
    DO  $\prime S ::= \prime S + b;; \prime M ::= \prime M + 1$  OD  
     $\{ \prime S = a * b \}$   
*<proof>*

The subsequent version of the proof applies the *vcg* method to reduce the Hoare statement to a purely logical problem that can be solved fully automatically. Note that we have to specify the WHILE loop invariant in the original statement.

**lemma**  $\Gamma \vdash \{ \prime M = 0 \wedge \prime S = 0 \}$   
    WHILE  $\prime M \neq a$   
    INV  $\{ \prime S = \prime M * b \}$   
    DO  $\prime S ::= \prime S + b;; \prime M ::= \prime M + 1$  OD  
     $\{ \prime S = a * b \}$   
*<proof>*

Here some examples of “breaking” out of a loop

**lemma**  $\Gamma \vdash \{ \prime M = 0 \wedge \prime S = 0 \}$   
    TRY  
    WHILE True  
    INV  $\{ \prime S = \prime M * b \}$   
    DO IF  $\prime M = a$  THEN THROW ELSE  $\prime S ::= \prime S + b;; \prime M ::= \prime M + 1$   
    FI OD  
    CATCH  
    SKIP  
    END  
     $\{ \prime S = a * b \}$   
*<proof>*

**lemma**  $\Gamma \vdash \{ \prime M = 0 \wedge \prime S = 0 \}$   
    TRY  
    WHILE True  
    INV  $\{ \prime S = \prime M * b \}$   
    DO IF  $\prime M = a$  THEN  $\prime Abr ::= \text{"Break"};;$  THROW  
    ELSE  $\prime S ::= \prime S + b;; \prime M ::= \prime M + 1$   
    FI  
    OD  
    CATCH  
    IF  $\prime Abr = \text{"Break"}$  THEN SKIP ELSE Throw FI  
    END

$\{\{ 'S = a * b \}$   
 $\langle proof \rangle$

Some more syntactic sugar, the label statement  $\dots \cdot \dots$  as shorthand for the *TRY-CATCH* above, and the *RAISE* for an state-update followed by a *THROW*.

**lemma**  $\Gamma \vdash \{\{ 'M = 0 \wedge 'S = 0 \}$   
 $\{\{ 'Abr = "Break" \}\} \cdot \text{WHILE True INV } \{\{ 'S = 'M * b \}$   
 $\text{DO IF } 'M = a \text{ THEN RAISE } 'Abr ::= "Break"$   
 $\text{ELSE } 'S ::= 'S + b;; 'M ::= 'M + 1$   
 $\text{FI}$   
 $\text{OD}$   
 $\{\{ 'S = a * b \}$   
 $\langle proof \rangle$

**lemma**  $\Gamma \vdash \{\{ 'M = 0 \wedge 'S = 0 \}$   
 $\text{TRY}$   
 $\text{WHILE True}$   
 $\text{INV } \{\{ 'S = 'M * b \}$   
 $\text{DO IF } 'M = a \text{ THEN RAISE } 'Abr ::= "Break"$   
 $\text{ELSE } 'S ::= 'S + b;; 'M ::= 'M + 1$   
 $\text{FI}$   
 $\text{OD}$   
 $\text{CATCH}$   
 $\text{IF } 'Abr = "Break" \text{ THEN SKIP ELSE Throw FI}$   
 $\text{END}$   
 $\{\{ 'S = a * b \}$   
 $\langle proof \rangle$

**lemma**  $\Gamma \vdash \{\{ 'M = 0 \wedge 'S = 0 \}$   
 $\{\{ 'Abr = "Break" \}\} \cdot \text{WHILE True}$   
 $\text{INV } \{\{ 'S = 'M * b \}$   
 $\text{DO IF } 'M = a \text{ THEN RAISE } 'Abr ::= "Break"$   
 $\text{ELSE } 'S ::= 'S + b;; 'M ::= 'M + 1$   
 $\text{FI}$   
 $\text{OD}$   
 $\{\{ 'S = a * b \}$   
 $\langle proof \rangle$

Blocks

**lemma**  $\Gamma \vdash \{\{ 'T = i \} \text{ LOC } 'T;; 'T ::= 2 \text{ COL } \{\{ 'T \leq i \}$   
 $\langle proof \rangle$

**lemma**  $\Gamma \vdash \{\{ 'N = n \} \text{ LOC } 'N ::= 10;; 'N ::= 'N + 2 \text{ COL } \{\{ 'N = n \}$   
 $\langle proof \rangle$

**lemma**  $\Gamma \vdash \{\{ 'N = n \} \text{ LOC } 'N ::= 10, 'M;; 'N ::= 'N + 2 \text{ COL } \{\{ 'N = n \}$   
 $\langle proof \rangle$

## 17.4 Summing Natural Numbers

We verify an imperative program to sum natural numbers up to a given limit. First some functional definition for proper specification of the problem.

**primrec**

$sum :: (nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat$

**where**

$sum\ f\ 0 = 0$

|  $sum\ f\ (Suc\ n) = f\ n + sum\ f\ n$

**syntax**

$-sum :: idt \Rightarrow nat \Rightarrow nat \Rightarrow nat$

$(\langle SUMM\ -<-. \rightarrow [0, 0, 10]\ 10)$

**syntax-consts**

$-sum == sum$

**translations**

$SUMM\ j<k.\ b == CONST\ sum\ (\lambda j.\ b)\ k$

The following proof is quite explicit in the individual steps taken, with the *vcg* method only applied locally to take care of assignment and sequential composition. Note that we express intermediate proof obligation in pure logic, without referring to the state space.

**theorem**  $\Gamma \vdash \{True\}$

$\ 'S := 0;;\ 'I := 1;;$

$\ WHILE\ 'I \neq n$

$\ DO$

$\ 'S := 'S + 'I;;$

$\ 'I := 'I + 1$

$\ OD$

$\ \{ 'S = (SUMM\ j < n.\ j) \}$

$(is\ \Gamma \vdash - (-;;\ ?while)\ -)$

$\langle proof \rangle$

The next version uses the *vcg* method, while still explaining the resulting proof obligations in an abstract, structured manner.

**theorem**  $\Gamma \vdash \{True\}$

$\ 'S := 0;;\ 'I := 1;;$

$\ WHILE\ 'I \neq n$

$\ INV\ \{ 'S = (SUMM\ j < 'I.\ j) \}$

$\ DO$

$\ 'S := 'S + 'I;;$

$\ 'I := 'I + 1$

$\ OD$

$\ \{ 'S = (SUMM\ j < n.\ j) \}$

$\langle proof \rangle$

Certainly, this proof may be done fully automatically as well, provided that the invariant is given beforehand.

**theorem**  $\Gamma \vdash \{\{True\}\}$   
 $\quad 'S ::= 0;; 'I ::= 1;;$   
 $\quad WHILE 'I \neq n$   
 $\quad INV \{\{ 'S = (SUMM j < 'I. j) \}\}$   
 $\quad DO$   
 $\quad \quad 'S ::= 'S + 'I;;$   
 $\quad \quad 'I ::= 'I + 1$   
 $\quad OD$   
 $\quad \{\{ 'S = (SUMM j < n. j) \}\}$   
 $\langle proof \rangle$

## 17.5 SWITCH

**lemma**  $\Gamma \vdash \{\{ 'N = 5 \}\} SWITCH 'B$   
 $\quad \{True\} \Rightarrow 'N ::= 6$   
 $\quad | \{False\} \Rightarrow 'N ::= 7$   
 $\quad END$   
 $\{\{ 'N > 5 \}\}$   
 $\langle proof \rangle$

**lemma**  $\Gamma \vdash \{\{ 'N = 5 \}\} SWITCH 'N$   
 $\quad \{v. v < 5\} \Rightarrow 'N ::= 6$   
 $\quad | \{v. v \geq 5\} \Rightarrow 'N ::= 7$   
 $\quad END$   
 $\{\{ 'N > 5 \}\}$   
 $\langle proof \rangle$

## 17.6 (Mutually) Recursive Procedures

### 17.6.1 Factorial

We want to define a procedure for the factorial. We first define a HOL functions that calculates it to specify the procedure later on.

**primrec**  $fac:: nat \Rightarrow nat$   
**where**  
 $fac\ 0 = 1 \mid$   
 $fac\ (Suc\ n) = (Suc\ n) * fac\ n$

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

Now we define the procedure

**procedures**  
 $Fac\ (N|R) = IF\ 'N = 0\ THEN\ 'R ::= 1$   
 $\quad ELSE\ 'R ::= CALL\ Fac('N - 1);;$   
 $\quad 'R ::= 'N * 'R$   
 $FI$

A procedure is given by the signature of the procedure followed by the

procedure body. The signature consists of the name of the procedure and a list of parameters. The parameters in front of the pipe `|` are value parameters and behind the pipe are the result parameters. Value parameters model call by value semantics. The value of a result parameter at the end of the procedure is passed back to the caller.

Behind the scenes the `procedures` command provides us convenient syntax for procedure calls, defines a constant for the procedure body (named `Fac-body`) and creates some locales. The purpose of locales is to set up logical contexts to support modular reasoning. A locale is named `Fac-impl` and extends the `hoare` locale with a theorem  $\Gamma \text{ "Fac" } = \text{Fac-body}$  that simply states how the procedure is defined in the procedure context. Check out the locales. The purpose of the locales is to give us easy means to setup the context in which we will prove programs correct. In these locales the procedure context  $\Gamma$  is fixed. So always use this letter in procedure specifications. This is crucial, if we later on prove some tuples under the assumption of some procedure specifications.

```
thm Fac-body.Fac-body-def
print-locale Fac-impl
```

To see how a call is syntactically translated you can switch off the printing translation via the configuration option `hoare-use-call-tr'`

```
context Fac-impl
begin
```

`'M ::= CALL Fac('N)` is internally:

```
declare [[hoare-use-call-tr' = false]]
```

```
call ( $\lambda s. s(\!N-' := N-' s)$ ) Fac-'proc ( $\lambda s t. s(\!globals := globals t)$ ) ( $\lambda i t. 'M ::= R-' t$ )
```

```
term CALL Fac('N, 'M)
declare [[hoare-use-call-tr' = true]]
end
```

Now let us prove that `Fac` meets its specification.

Procedure specifications are ordinary Hoare tuples. We use the parameter-less call for the specification; `'R ::= PROC Fac('N)` is syntactic sugar for `Call "Fac"`. This emphasises that the specification describes the internal behaviour of the procedure, whereas parameter passing corresponds to the procedure call.

```
lemma (in Fac-impl)
shows  $\forall n. \Gamma, \Theta \vdash \{ 'N = n \} \text{ PROC Fac('N, 'R) } \{ 'R = fac\ n \}$ 
 $\langle \textit{proof} \rangle$ 
```

Since the factorial was implemented recursively, the main ingredient of this proof is, to assume that the specification holds for the recursive call of `Fac`

and prove the body correct. The assumption for recursive calls is added to the context by the rule *HoarePartial.ProcRec1* (also derived from general rule for mutually recursive procedures):

$$\begin{aligned} & \llbracket \forall Z. \Gamma, \Theta \cup (\bigcup_Z \{(P\ Z, p, Q\ Z, A\ Z)\}) \vdash_{/F} (P\ Z)\ \text{the } (\Gamma\ p)\ (Q\ Z), (A\ Z); \\ & \quad p \in \text{dom } \Gamma \rrbracket \\ \implies & \forall Z. \Gamma, \Theta \vdash_{/F} (P\ Z)\ \text{Call } p\ (Q\ Z), (A\ Z) \end{aligned}$$

The verification condition generator will infer the specification out of the context when it encounters a recursive call of the factorial.

We can also step through verification condition generation. When the verification condition generator encounters a procedure call it tries to use the rule *ProcSpec*. To be successful there must be a specification of the procedure in the context.

**lemma** (in *Fac-impl*)  
**shows**  $\forall n. \Gamma \vdash \{N=n\} \ 'R := PROC\ Fac\ (N) \ \{R = fac\ n\}$   
*<proof>*

Here some Isar style version of the proof

**lemma** (in *Fac-impl*)  
**shows**  $\forall n. \Gamma \vdash \{N=n\} \ 'R := PROC\ Fac\ (N) \ \{R = fac\ n\}$   
*<proof>*

To avoid retyping of potentially large pre and postconditions in the previous proof we can use the casual term abbreviations of the Isar language.

**lemma** (in *Fac-impl*)  
**shows**  $\forall n. \Gamma \vdash \{N=n\} \ 'R := PROC\ Fac\ (N) \ \{R = fac\ n\}$   
 (**is**  $\forall n. \Gamma \vdash (?Pre\ n)\ ?Fac\ (?Post\ n)$ )  
*<proof>*

The previous proof pattern has still some kind of inconvenience. The augmented context is always printed in the proof state. That can mess up the state, especially if we have large specifications. This may be annoying if we want to develop single step or structured proofs. In this case it can be a good idea to introduce a new variable for the augmented context.

**lemma** (in *Fac-impl*) *Fac-spec*:  
**shows**  $\forall n. \Gamma \vdash \{N=n\} \ 'R := PROC\ Fac\ (N) \ \{R = fac\ n\}$   
 (**is**  $\forall n. \Gamma \vdash (?Pre\ n)\ ?Fac\ (?Post\ n)$ )  
*<proof>*

There are different rules available to prove procedure calls, depending on the kind of postcondition and whether or not the procedure is recursive or even mutually recursive. See for example *HoarePartial.ProcRec1*, *HoarePartial.ProcNoRec1*. They are all derived from the most general rule *HoarePartial.ProcRec*. All of them have some side-condition concerning definedness

of the procedure. They can be solved in a uniform fashion. That's why we have created the method *hoare-rule*, which behaves like the method *rule* but automatically tries to solve the side-conditions.

### 17.6.2 Odd and Even

Odd and even are defined mutually recursive here. In the *procedures* command we conjoin both definitions with *and*.

**procedures**

```

odd(N | A) = IF 'N=0 THEN 'A:=0
            ELSE IF 'N=1 THEN CALL even ('N - 1, 'A)
            ELSE CALL odd ('N - 2, 'A)
            FI
            FI

```

**and**

```

even(N | A) = IF 'N=0 THEN 'A:=1
             ELSE IF 'N=1 THEN CALL odd ('N - 1, 'A)
             ELSE CALL even ('N - 2, 'A)
             FI
             FI

```

**print-theorems**

```

thm odd-body.odd-body-def
thm even-body.even-body-def
print-locale odd-even-clique

```

To prove the procedure calls to *odd* respectively *even* correct we first derive a rule to justify that we can assume both specifications to verify the bodies. This rule can be derived from the general *HoarePartial.ProcRec* rule. An ML function does this work:

$\langle ML \rangle$

**lemma** (in *odd-even-clique*)

```

shows odd-spec:  $\forall n. \Gamma \vdash \{ 'N=n \} 'A ::= PROC\ odd('N)$ 
               $\{ (\exists b. n = 2 * b + 'A) \wedge 'A < 2 \}$  (is ?P1)
and even-spec:  $\forall n. \Gamma \vdash \{ 'N=n \} 'A ::= PROC\ even('N)$ 
               $\{ (\exists b. n + 1 = 2 * b + 'A) \wedge 'A < 2 \}$  (is ?P2)

```

$\langle proof \rangle$

## 17.7 Expressions With Side Effects

R := N++ + M++

**lemma**  $\Gamma \vdash \{ True \}$

$'N \gg n. 'N ::= 'N + 1 \gg$

$'M \gg m. 'M ::= 'M + 1 \gg$   
 $'R ::= n + m$   
 $\{\{ 'R = 'N + 'M - 2 \}\}$   
 $\langle proof \rangle$

$R := \text{Fac } (N) + \text{Fac } (M)$

**lemma (in *Fac-impl*) shows**

$\Gamma \vdash \{\{ True \}\}$   
 $CALL \text{Fac}'(N) \gg n. CALL \text{Fac}'(M) \gg m.$   
 $'R ::= n + m$   
 $\{\{ 'R = fac 'N + fac 'M \}\}$   
 $\langle proof \rangle$

$R := (\text{Fac}(\text{Fac } (N)))$

**lemma (in *Fac-impl*) shows**

$\Gamma \vdash \{\{ True \}\}$   
 $CALL \text{Fac}'(N) \gg n. CALL \text{Fac}(n) \gg m.$   
 $'R ::= m$   
 $\{\{ 'R = fac (fac 'N) \}\}$   
 $\langle proof \rangle$

## 17.8 Global Variables and Heap

Now we define and verify some procedures on heap-lists. We consider list structures consisting of two fields, a content element *cont* and a reference to the next list element *next*. We model this by the following state space where every field has its own heap.

**record *globals-list* =**  
 $next-' :: ref \Rightarrow ref$   
 $cont-' :: ref \Rightarrow nat$

**record *'g list-vars* = 'g state +**  
 $p-' :: ref$   
 $q-' :: ref$   
 $r-' :: ref$   
 $root-' :: ref$   
 $tmp-' :: ref$

Updates to global components inside a procedure will always be propagated to the caller. This is implicitly done by the parameter passing syntax translations. The record containing the global variables must begin with the prefix "globals".

We first define an append function on lists. It takes two references as parameters. It appends the list referred to by the first parameter with the list referred to by the second parameter, and returns the result right into the first parameter.

**procedures**

```

append(p,q|p) =
  IF 'p=NULL THEN 'p ::= 'q ELSE 'p → 'next ::= CALL append('p→'next,'q)
FI

```

**context** *append-impl***begin****declare**  $[[\text{hoare-use-call-tr}' = \text{false}]]$ **term** *CALL* *append*('p,'q,'p→'next)**declare**  $[[\text{hoare-use-call-tr}' = \text{true}]]$ **end**

Below we give two specifications this time. One captures the functional behaviour and focuses on the entities that are potentially modified by the procedure, the other one is a pure frame condition. The list in the modifies clause has to list all global state components that may be changed by the procedure. Note that we know from the modifies clause that the *cont* parts of the lists will not be changed. Also a small side note on the syntax. We use ordinary brackets in the postcondition of the modifies clause, and also the state components do not carry the acute, because we explicitly note the state  $t$  here.

The functional specification now introduces two logical variables besides the state space variable  $\sigma$ , namely  $P_s$  and  $Q_s$ . They are universally quantified and range over both the pre and the postcondition, so that we are able to properly instantiate the specification during the proofs. The syntax  $\{\sigma. \dots\}$  is a shorthand to fix the current state:  $\{s. \sigma = s \dots\}$ .

**lemma** (in *append-impl*) *append-spec*:**shows**  $\forall \sigma P_s Q_s. \Gamma \vdash$ 

$$\{\sigma. \text{List } 'p \text{ } 'next P_s \wedge \text{List } 'q \text{ } 'next Q_s \wedge \text{set } P_s \cap \text{set } Q_s = \{\}\}$$

$$'p ::= \text{PROC } \text{append}('p, 'q)$$

$$\{\text{List } 'p \text{ } 'next (P_s @ Q_s) \wedge (\forall x. x \notin \text{set } P_s \longrightarrow 'next x = \sigma_{next} x)\}$$
*<proof>*

The modifies clause is equal to a proper record update specification of the following form.

**lemma**  $\{t. t \text{ may-only-modify-globals } Z \text{ in } [next]\}$ 

=

$$\{t. \exists next. \text{globals } t = \text{next}'\text{-update } (\lambda-. next) (\text{globals } Z)\}$$
*<proof>*

If the verification condition generator works on a procedure call it checks whether it can find a modified clause in the context. If one is present the procedure call is simplified before the Hoare rule *HoarePartial.ProcSpec* is applied. Simplification of the procedure call means, that the “copy back” of the global components is simplified. Only those components that occur

in the modifies clause will actually be copied back. This simplification is justified by the rule *HoarePartial.ProcModifyReturn*. So after this simplification all global components that do not appear in the modifies clause will be treated as local variables.

You can study the effect of the modifies clause on the following two examples, where we want to prove that (@) does not change the *cont* part of the heap.

**lemma** (in *append-impl*)

**shows**  $\Gamma \vdash \{\!| p = \text{Null} \wedge 'cont = c \!|\} 'p ::= \text{CALL } \text{append}( 'p, \text{Null} ) \{\!| 'cont = c \!|\}$   
*<proof>*

To prove the frame condition, we have to tell the verification condition generator to use only the modifies clauses and not to search for functional specifications by the parameter *spec=modifies*. It will also try to solve the verification conditions automatically.

**lemma** (in *append-impl*) *append-modifies*:

**shows**

$\forall \sigma. \Gamma \vdash \{\sigma\} 'p ::= \text{PROC } \text{append}( 'p, 'q ) \{t. t \text{ may-only-modify-globals } \sigma \text{ in } [next]\}$   
*<proof>*

**lemma** (in *append-impl*)

**shows**  $\Gamma \vdash \{\!| p = \text{Null} \wedge 'cont = c \!|\} 'p \rightarrow 'next ::= \text{CALL } \text{append}( 'p, \text{Null} ) \{\!| 'cont = c \!|\}$   
*<proof>*

Of course we could add the modifies clause to the functional specification as well. But separating both has the advantage that we split up the verification work. We can make use of the modifies clause before we apply the functional specification in a fully automatic fashion.

To verify the body of (@) we do not need the modifies clause, since the specification does not talk about *cont* at all, and we don't access *cont* inside the body. This may be different for more complex procedures.

To prove that a procedure respects the modifies clause, we only need the modifies clauses of the procedures called in the body. We do not need the functional specifications. So we can always prove the modifies clause without functional specifications, but we may need the modifies clause to prove the functional specifications.

### 17.8.1 Insertion Sort

**primrec** *sorted*:: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  bool

**where**

*sorted le* [] = True |

*sorted le* (x#xs) = (( $\forall y \in \text{set } xs. le\ x\ y$ )  $\wedge$  *sorted le* xs)

**procedures**

```

insert( $r, p \mid p$ ) =
  IF  $r = \text{Null}$  THEN SKIP
  ELSE IF  $p = \text{Null}$  THEN  $p := r;; p \rightarrow \text{next} := \text{Null}$ 
  ELSE IF  $r \rightarrow \text{cont} \leq p \rightarrow \text{cont}$ 
    THEN  $r \rightarrow \text{next} := p;; p := r$ 
    ELSE  $p \rightarrow \text{next} := \text{CALL insert}(r, p \rightarrow \text{next})$ 
  FI
FI

```

In the postcondition of the functional specification there is a small but important subtlety. Whenever we talk about the *cont* part we refer to the one of the pre-state, even in the conclusion of the implication. The reason is, that we have separated out, that *cont* is not modified by the procedure, to the modifies clause. So whenever we talk about unmodified parts in the postcondition we have to use the pre-state part, or explicitly state an equality in the postcondition. The reason is simple. If the postcondition would talk about *'cont* instead of  $\sigma \text{cont}$ , we get a new instance of *cont* during verification and the postcondition would only state something about this new instance. But as the verification condition generator uses the modifies clause the caller of *insert* instead still has the old *cont* after the call. That's the very reason for the modifies clause. So the caller and the specification will simply talk about two different things, without being able to relate them (unless an explicit equality is added to the specification).

**lemma** (in *insert-impl*) *insert-modifies*:

$\forall \sigma. \Gamma \vdash \{\sigma\} p := \text{PROC insert}(r, p) \{t. t \text{ may-only-modify-globals } \sigma \text{ in } [\text{next}]\}$   
 $\langle \text{proof} \rangle$

**lemma** (in *insert-impl*) *insert-spec*:

$\forall \sigma Ps. \Gamma \vdash \{\sigma. \text{List } p \text{ 'next } Ps \wedge \text{sorted } (\leq) (\text{map } 'cont Ps) \wedge$   
 $\quad r \neq \text{Null} \wedge r \notin \text{set } Ps\}$   
 $\quad p := \text{PROC insert}(r, p)$   
 $\{\exists Qs. \text{List } p \text{ 'next } Qs \wedge \text{sorted } (\leq) (\text{map } \sigma cont Qs) \wedge$   
 $\quad \text{set } Qs = \text{insert } \sigma r (\text{set } Ps) \wedge$   
 $\quad (\forall x. x \notin \text{set } Qs \longrightarrow 'next x = \sigma next x)\}$

$\langle \text{proof} \rangle$

**procedures**

```

insertSort( $p \mid p$ ) =
   $r := \text{Null};;$ 
  WHILE ( $p \neq \text{Null}$ ) DO
     $q := p;;$ 
     $p := p \rightarrow \text{next};;$ 

```

$$\begin{aligned} & 'r ::= CALL\ insert('q, 'r) \\ & OD;; \\ & 'p ::= 'r \end{aligned}$$

**lemma** (in *insertSort-impl*) *insertSort-modifies*:

**shows**

$$\forall \sigma. \Gamma \vdash \{ \sigma \} 'p ::= PROC\ insertSort('p) \\ \{ t. t\ may\ only\ modify\ globals\ \sigma\ in\ [next] \}$$

*<proof>*

Insertion sort is not implemented recursively here but with a while loop. Note that the while loop is not annotated with an invariant in the procedure definition. The invariant only comes into play during verification. Therefore we will annotate the body during the proof with the rule *HoarePartial.annotateI*.

**lemma** (in *insertSort-impl*) *insertSort-body-spec*:

**shows**  $\forall \sigma\ Ps. \Gamma, \Theta \vdash \{ \sigma. List\ 'p\ 'next\ Ps \}$

$$\begin{aligned} & 'p ::= PROC\ insertSort('p) \\ & \{ \exists Qs. List\ 'p\ 'next\ Qs \wedge sorted\ (\leq)\ (map\ \sigma\ cont\ Qs) \wedge \\ & \quad set\ Qs = set\ Ps \} \end{aligned}$$

*<proof>*

## 17.8.2 Memory Allocation and Deallocation

The basic idea of memory management is to keep a list of allocated references in the state space. Allocation of a new reference adds a new reference to the list deallocation removes a reference. Moreover we keep a counter "free" for the free memory.

**record** *globals-list-alloc* = *globals-list* +  
*alloc-'::ref list*  
*free-'::nat*

**record** *'g list-vars'* = *'g list-vars* +  
*i-'::nat*  
*first-'::ref*

**definition** *sz* = (2::nat)

Restrict locale *hoare* to the required type.

**locale** *hoare-ex* =

*hoare*  $\Gamma$  **for**  $\Gamma :: 'c \rightarrow (( 'a\ globals\ list\ alloc\ scheme, 'b)\ list\ vars'\ scheme, 'c, 'd)$   
*com*

**lemma** (in *hoare-ex*)

$$\begin{aligned} &\Gamma \vdash \{\{i = 0 \wedge \text{'first} = \text{Null} \wedge n * sz \leq \text{'free}\}\} \\ &\quad \text{WHILE } i < n \\ &\quad \text{INV } \{\{\exists Ps. \text{List } \text{'first } \text{'next } Ps \wedge \text{length } Ps = i \wedge i \leq n \wedge \\ &\quad \quad \text{set } Ps \subseteq \text{set } \text{'alloc} \wedge (n - i) * sz \leq \text{'free}\}\} \\ &\quad \text{DO} \\ &\quad \quad \text{'p} ::= \text{NEW } sz [\text{'cont} ::= 0, \text{'next} ::= \text{Null}];; \\ &\quad \quad \text{'p} \rightarrow \text{'next} ::= \text{'first};; \\ &\quad \quad \text{'first} ::= \text{'p};; \\ &\quad \quad i ::= i + 1 \\ &\quad \text{OD} \\ &\quad \{\{\exists Ps. \text{List } \text{'first } \text{'next } Ps \wedge \text{length } Ps = n \wedge \text{set } Ps \subseteq \text{set } \text{'alloc}\}\} \end{aligned}$$

*<proof>*

**lemma** (in *hoare-ex*)

$$\begin{aligned} &\Gamma \vdash \{\{i = 0 \wedge \text{'first} = \text{Null} \wedge n * sz \leq \text{'free}\}\} \\ &\quad \text{WHILE } i < n \\ &\quad \text{INV } \{\{\exists Ps. \text{List } \text{'first } \text{'next } Ps \wedge \text{length } Ps = i \wedge i \leq n \wedge \\ &\quad \quad \text{set } Ps \subseteq \text{set } \text{'alloc} \wedge (n - i) * sz \leq \text{'free}\}\} \\ &\quad \text{DO} \\ &\quad \quad \text{'p} ::= \text{NNEW } sz [\text{'cont} ::= 0, \text{'next} ::= \text{Null}];; \\ &\quad \quad \text{'p} \rightarrow \text{'next} ::= \text{'first};; \\ &\quad \quad \text{'first} ::= \text{'p};; \\ &\quad \quad i ::= i + 1 \\ &\quad \text{OD} \\ &\quad \{\{\exists Ps. \text{List } \text{'first } \text{'next } Ps \wedge \text{length } Ps = n \wedge \text{set } Ps \subseteq \text{set } \text{'alloc}\}\} \end{aligned}$$

*<proof>*

## 17.9 Fault Avoiding Semantics

If we want to ensure that no runtime errors occur we can insert guards into the code. We will not be able to prove any nontrivial Hoare triple about code with guards, if we cannot show that the guards will never fail. A trivial hoare triple is one with an empty precondition.

**lemma**  $\Gamma \vdash \{\{True\}\} \{\{p \neq \text{Null}\}\} \mapsto \text{'p} \rightarrow \text{'next} ::= \text{'p} \{\{True\}\}$   
*<proof>*

**lemma**  $\Gamma \vdash \{\{\}\} \{\{p \neq \text{Null}\}\} \mapsto \text{'p} \rightarrow \text{'next} ::= \text{'p} \{\{True\}\}$   
*<proof>*

Let us consider this small program that reverts a list. At first without guards.

**lemma** (in *hoare-ex*) *rev-strip*:

$$\begin{aligned} &\Gamma \vdash \{\{\text{List } \text{'p } \text{'next } Ps \wedge \text{List } \text{'q } \text{'next } Qs \wedge \text{set } Ps \cap \text{set } Qs = \{\}\} \wedge \\ &\quad \text{set } Ps \subseteq \text{set } \text{'alloc} \wedge \text{set } Qs \subseteq \text{set } \text{'alloc}\}\} \end{aligned}$$

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

If we want to ensure that we do not dereference *Null* or access unallocated memory, we have to add some guards.

**locale** *hoare-ex-guard* =  
 $hoare \ \Gamma \ \mathbf{for} \ \Gamma :: 'c \rightarrow (( 'a \ \mathit{globals-list-alloc-scheme}, 'b) \ \mathit{list-vars'-scheme}, 'c, \mathit{bool})$   
 $com$

**lemma**

$(\mathbf{in} \ \mathit{hoare-ex-guard})$   
 $\Gamma \vdash \ \{List \ 'p \ 'next \ Ps \wedge List \ 'q \ 'next \ Qs \wedge set \ Ps \cap set \ Qs = \{\} \wedge$   
 $\quad set \ Ps \subseteq set \ 'alloc \wedge set \ Qs \subseteq set \ 'alloc\}$   
 $WHILE \ 'p \neq Null$   
 $INV \ \{\exists ps \ qs. List \ 'p \ 'next \ ps \wedge List \ 'q \ 'next \ qs \wedge set \ ps \cap set \ qs = \{\} \wedge$   
 $\quad rev \ ps \ @ \ qs = rev \ Ps \ @ \ Qs \wedge$   
 $\quad set \ ps \subseteq set \ 'alloc \wedge set \ qs \subseteq set \ 'alloc\}$   
 $DO \ 'r ::= 'p;;$   
 $\quad \{p \neq Null \wedge 'p \in set \ 'alloc\} \mapsto 'p ::= 'p \rightarrow 'next;;$   
 $\quad \{r \neq Null \wedge 'r \in set \ 'alloc\} \mapsto 'r \rightarrow 'next ::= 'q;;$   
 $\quad 'q ::= 'r \ OD$   
 $\ \{List \ 'q \ 'next \ (rev \ Ps \ @ \ Qs) \wedge set \ Ps \subseteq set \ 'alloc \wedge set \ Qs \subseteq set \ 'alloc\}$   
 $\langle proof \rangle$

We can also just prove that no faults will occur, by giving the trivial post-condition.

**lemma**  $(\mathbf{in} \ \mathit{hoare-ex-guard}) \ \mathit{rev-noFault}$ :

$\Gamma \vdash \ \{List \ 'p \ 'next \ Ps \wedge List \ 'q \ 'next \ Qs \wedge set \ Ps \cap set \ Qs = \{\} \wedge$   
 $\quad set \ Ps \subseteq set \ 'alloc \wedge set \ Qs \subseteq set \ 'alloc\}$   
 $WHILE \ 'p \neq Null$   
 $INV \ \{\exists ps \ qs. List \ 'p \ 'next \ ps \wedge List \ 'q \ 'next \ qs \wedge set \ ps \cap set \ qs = \{\} \wedge$   
 $\quad rev \ ps \ @ \ qs = rev \ Ps \ @ \ Qs \wedge$   
 $\quad set \ ps \subseteq set \ 'alloc \wedge set \ qs \subseteq set \ 'alloc\}$   
 $DO \ 'r ::= 'p;;$   
 $\quad \{p \neq Null \wedge 'p \in set \ 'alloc\} \mapsto 'p ::= 'p \rightarrow 'next;;$   
 $\quad \{r \neq Null \wedge 'r \in set \ 'alloc\} \mapsto 'r \rightarrow 'next ::= 'q;;$   
 $\quad 'q ::= 'r \ OD$   
 $UNIV, UNIV$   
 $\langle proof \rangle$

**lemma**  $(\mathbf{in} \ \mathit{hoare-ex-guard}) \ \mathit{rev-moduloGuards}$ :

$$\begin{array}{l}
\Gamma \vdash / \{True\} \{ \{List \ 'p \ 'next \ Ps \wedge List \ 'q \ 'next \ Qs \wedge set \ Ps \cap set \ Qs = \{\} \wedge \\
\quad set \ Ps \subseteq set \ 'alloc \wedge set \ Qs \subseteq set \ 'alloc\} \\
WHILE \ 'p \neq Null \\
INV \ \{ \exists ps \ qs. List \ 'p \ 'next \ ps \wedge List \ 'q \ 'next \ qs \wedge set \ ps \cap set \ qs = \{\} \wedge \\
\quad rev \ ps \ @ \ qs = rev \ Ps \ @ \ Qs \wedge \\
\quad set \ ps \subseteq set \ 'alloc \wedge set \ qs \subseteq set \ 'alloc\} \\
DO \ 'r \ := \ 'p;; \\
\quad \{ \{p \neq Null \wedge 'p \in set \ 'alloc\} \checkmark \mapsto 'p \ := \ 'p \rightarrow 'next;; \\
\quad \{ \{r \neq Null \wedge 'r \in set \ 'alloc\} \checkmark \mapsto 'r \rightarrow 'next \ := \ 'q;; \\
\quad \quad 'q \ := \ 'r \ OD \\
\{List \ 'q \ 'next \ (rev \ Ps \ @ \ Qs) \wedge set \ Ps \subseteq set \ 'alloc \wedge set \ Qs \subseteq set \ 'alloc\} \\
\langle proof \rangle
\end{array}$$

**lemma** *CombineStrip'*:  
**assumes** *deriv*:  $\Gamma, \Theta \vdash /_F P \ c' \ Q, A$   
**assumes** *deriv-strip*:  $\Gamma, \Theta \vdash / \{\} P \ c'' \ UNIV, UNIV$   
**assumes** *c''*:  $c'' = mark\text{-}guards \ False \ (strip\text{-}guards \ (-F) \ c')$   
**assumes** *c*:  $c = mark\text{-}guards \ False \ c'$   
**shows**  $\Gamma, \Theta \vdash / \{\} P \ c \ Q, A$   
 $\langle proof \rangle$

We can then combine the prove that no fault will occur with the functional proof of the programme without guards to get the full prove by the rule  $\llbracket ?\Gamma, ?\Theta \vdash / ?_F ?P \ ?c \ ?Q, ?A; ?\Gamma, ?\Theta \vdash ?P \ strip\text{-}guards \ (- \ ?F) \ ?c \ UNIV, UNIV \rrbracket \implies ?\Gamma, ?\Theta \vdash ?P \ ?c \ ?Q, ?A$

**lemma**  
*(in hoare-ex-guard)*  
 $\Gamma \vdash \{ \{List \ 'p \ 'next \ Ps \wedge List \ 'q \ 'next \ Qs \wedge set \ Ps \cap set \ Qs = \{\} \wedge \\
\quad set \ Ps \subseteq set \ 'alloc \wedge set \ Qs \subseteq set \ 'alloc\} \\
WHILE \ 'p \neq Null \\
INV \ \{ \exists ps \ qs. List \ 'p \ 'next \ ps \wedge List \ 'q \ 'next \ qs \wedge set \ ps \cap set \ qs = \{\} \wedge \\
\quad rev \ ps \ @ \ qs = rev \ Ps \ @ \ Qs \wedge \\
\quad set \ ps \subseteq set \ 'alloc \wedge set \ qs \subseteq set \ 'alloc\} \\
DO \ 'r \ := \ 'p;; \\
\quad \{ \{p \neq Null \wedge 'p \in set \ 'alloc\} \mapsto 'p \ := \ 'p \rightarrow 'next;; \\
\quad \{ \{r \neq Null \wedge 'r \in set \ 'alloc\} \mapsto 'r \rightarrow 'next \ := \ 'q;; \\
\quad \quad 'q \ := \ 'r \ OD \\
\{List \ 'q \ 'next \ (rev \ Ps \ @ \ Qs) \wedge set \ Ps \subseteq set \ 'alloc \wedge set \ Qs \subseteq set \ 'alloc\} \\
\langle proof \rangle$

In the previous example the effort to split up the prove did not really pay off. But when we think of programs with a lot of guards and complicated specifications it may be better to first focus on a prove without the messy guards. Maybe it is possible to automate the no fault proofs so that it

suffices to focus on the stripped program.

The purpose of guards is to watch for faults that can occur during evaluation of expressions. In the example before we watched for null pointer dereferencing or memory faults. We can also look for array index bounds or division by zero. As the condition of a while loop is evaluated in each iteration we cannot just add a guard before the while loop. Instead we need a special guard for the condition. Example:  $WHILE (False, \{\!| p \neq Null \!\}) \mapsto p \rightarrow next \neq Null \text{ DO SKIP OD}$

## 17.10 Circular Lists

**definition**

$$\begin{aligned} distPath &:: ref \Rightarrow (ref \Rightarrow ref) \Rightarrow ref \Rightarrow ref \text{ list} \Rightarrow bool \text{ where} \\ distPath \ x \ next \ y \ as &= (Path \ x \ next \ y \ as \ \wedge \ distinct \ as) \end{aligned}$$

**lemma** *neg-dP*:  $\llbracket p \neq q; Path \ p \ h \ q \ Ps; distinct \ Ps \rrbracket \implies \exists Qs. p \neq Null \ \wedge \ Ps = p \# Qs \ \wedge \ p \notin set \ Qs$   
 $\langle proof \rangle$

**lemma** *circular-list-rev-I*:

$$\begin{aligned} \Gamma \vdash & \{\!| root = r \ \wedge \ distPath \ root \ next \ root \ (r \# Ps) \!\} \\ & \ 'p := root;; \ 'q := root \rightarrow next;; \\ & \ WHILE \ 'q \neq root \\ INV & \ \{\!| \exists ps \ qs. distPath \ 'p \ next \ root \ ps \ \wedge \ distPath \ 'q \ next \ root \ qs \ \wedge \\ & \ \ \ \ \ \ root = r \ \wedge \ r \neq Null \ \wedge \ r \notin set \ Ps \ \wedge \ set \ ps \ \cap \ set \ qs = \{\} \ \wedge \\ & \ \ \ \ \ \ Ps = (rev \ ps) \ @ \ qs \!\} \\ DO & \ 'tmp := 'q;; \ 'q := 'q \rightarrow next;; \ 'tmp \rightarrow next := 'p;; \ 'p := 'tmp \text{ OD}; \\ & \ root \rightarrow next := 'p \\ & \ \{\!| root = r \ \wedge \ distPath \ root \ next \ root \ (r \# rev \ Ps) \!\} \\ & \langle proof \rangle \end{aligned}$$

**lemma** *path-is-list*:  $\wedge a \ next \ b. \llbracket Path \ b \ next \ a \ Ps; a \notin set \ Ps; a \neq Null \rrbracket \implies List \ b \ (next(a := Null)) \ (Ps \ @ \ [a])$   
 $\langle proof \rangle$

The simple algorithm for acyclic list reversal, with modified annotations, works for cyclic lists as well.:

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

$$\begin{aligned} \Gamma \vdash & \{\!| p = r \ \wedge \ distPath \ 'p \ next \ 'p \ (r \# Ps) \!\} \\ & \ 'q := Null;; \\ & \ WHILE \ 'p \neq Null \\ INV & \ \{\!| ((q = Null) \longrightarrow (\exists ps. distPath \ 'p \ next \ r \ ps \ \wedge \ ps = r \# Ps)) \ \wedge \\ & \ \ \ \ \ \ ((q \neq Null) \longrightarrow (\exists ps \ qs. distPath \ 'q \ next \ r \ qs \ \wedge \ List \ 'p \ next \ ps \ \wedge \end{aligned}$$

$$\neg (set\ ps \cap set\ qs = \{\} \wedge rev\ qs @ ps = Ps@[r]) \wedge$$

$$\neg ('p = Null \wedge 'q = Null \wedge r = Null)$$

$$\Downarrow$$

*DO*

$$'tmp ::= 'p;; 'p ::= 'p \rightarrow 'next;; 'tmp \rightarrow 'next ::= 'q;; 'q ::= 'tmp$$

*OD*

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

*<proof>*

Although the above algorithm is more succinct, its invariant looks more involved. The reason for the case distinction on  $q$  is due to the fact that during execution, the pointer variables can point to either cyclic or acyclic structures.

When working on lists, its sometimes better to remove *fun-upd-apply* from the simpset, and instead include *fun-upd-same* and *fun-upd-other* to the simpset

**lemma**  $\Gamma \vdash \{\sigma\}$

$$'I ::= 'M;;$$

$$ANNO\ \tau.\ \{\{\tau.\ 'I = {}^\sigma M\}\}$$

$$'M ::= 'N;; 'N ::= 'I$$

$$\{\{ 'M = {}^\tau N \wedge 'N = {}^\tau I \}\}$$

$$\{\{ 'M = {}^\sigma N \wedge 'N = {}^\sigma M \}\}$$

*<proof>*

**lemma**  $\Gamma \vdash (\{\sigma\} \cap \{\{ 'M = 0 \wedge 'S = 0 \}\})$

$$(ANNO\ \tau.\ (\{\{\tau\} \cap \{\{ 'A = {}^\sigma A \wedge 'I = {}^\sigma I \wedge 'M = 0 \wedge 'S = 0 \}\}\})$$

$$WHILE\ 'M \neq 'A$$

$$INV\ \{\{ 'S = 'M * 'I \wedge 'A = {}^\tau A \wedge 'I = {}^\tau I \}\}$$

$$DO\ 'S ::= 'S + 'I;; 'M ::= 'M + 1\ OD$$

$$\{\{ 'S = {}^\tau A * {}^\tau I \}\}$$

$$\{\{ 'S = {}^\sigma A * {}^\sigma I \}\}$$

*<proof>*

Instead of annotations one can also directly use previously proven lemmas.

**lemma** *foo-lemma*:  $\forall n\ m.\ \Gamma \vdash \{\{ 'N = n \wedge 'M = m \}\} 'N ::= 'N + 1;; 'M ::= 'M + 1$

$$\{\{ 'N = n + 1 \wedge 'M = m + 1 \}\}$$

*<proof>*

**lemma**  $\Gamma \vdash \{\{ 'N = n \wedge 'M = m \}\} LEMMA\ foo-lemma$

$$'N ::= 'N + 1;; 'M ::= 'M + 1$$

$$END;;$$

$$'N ::= 'N + 1$$

$$\{\{ 'N = n + 2 \wedge 'M = m + 1 \}\}$$

*<proof>*

**lemma**  $\Gamma \vdash \{ 'N = n \wedge 'M = m \}$   
*LEMMA foo-lemma*  
 $'N ::= 'N + 1;; 'M ::= 'M + 1$   
*END;;*  
*LEMMA foo-lemma*  
 $'N ::= 'N + 1;; 'M ::= 'M + 1$   
*END*  
 $\{ 'N = n + 2 \wedge 'M = m + 2 \}$   
*<proof>*

**lemma**  $\Gamma \vdash \{ 'N = n \wedge 'M = m \}$   
 $'N ::= 'N + 1;; 'M ::= 'M + 1;;$   
 $'N ::= 'N + 1;; 'M ::= 'M + 1$   
 $\{ 'N = n + 2 \wedge 'M = m + 2 \}$   
*<proof>*

Just some test on marked, guards

**lemma**  $\Gamma \vdash \{ True \} \text{ WHILE } \{ P 'N \} \surd, \{ Q 'M \} \#, \{ R 'N \} \dashv \dashrightarrow 'N < 'M$   
*INV*  $\{ 'N < 2 \} \text{ DO}$   
 $'N ::= 'M$   
*OD*  
 $\{ hard \}$   
*<proof>*

**lemma**  $\Gamma \vdash / \{ True \} \{ True \} \text{ WHILE } \{ P 'N \} \surd, \{ Q 'M \} \#, \{ R 'N \} \dashv \dashrightarrow 'N < 'M$   
*INV*  $\{ 'N < 2 \} \text{ DO}$   
 $'N ::= 'M$   
*OD*  
 $\{ hard \}$   
*<proof>*

**term**  $\Gamma \vdash / \{ True \} \{ True \} \text{ WHILE}_g 'N < 'Arr!i$   
*FIX*  $Z.$   
*INV*  $\{ 'N < 2 \}$   
  
*DO*  
 $'N ::= 'M$   
*OD*  
 $\{ hard \}$

**lemma**  $\Gamma \vdash / \{ True \} \{ True \} \text{ WHILE}_g 'N < 'Arr!i$   
*FIX*  $Z.$   
*INV*  $\{ 'N < 2 \}$   
*VAR*  $arbitrary$   
*DO*  
 $'N ::= 'M$

```

      OD
    {hard}
  <proof>

lemma  $\Gamma \vdash / \{True\} \{True\} \text{ WHILE } \{P \ 'N \} \checkmark, \{Q \ 'M \} \#, \{R \ 'N \} \mapsto 'N < 'M$ 
      FIX Z.
      INV  $\{ 'N < 2 \}$ 
      VAR arbitrary
      DO
      'N ::= 'M
      OD
    {hard}
  <proof>

end

```

## 18 Examples using Statespaces

```
theory VcgExSP imports ../HeapList ../Vcg begin
```

### 18.1 State Spaces

First of all we provide a store of program variables that occur in the programs considered later. Slightly unexpected things may happen when attempting to work with undeclared variables.

```
hoarestate state-space =
```

```

  A :: nat
  I :: nat
  M :: nat
  N :: nat
  R :: nat
  S :: nat
  B :: bool
  Abr :: string

```

```
lemma (in state-space)  $\Gamma \vdash \{ 'N = n \} \text{ LOC } 'N ::= 10;; 'N ::= 'N + 2 \text{ COL } \{ 'N = n \}$ 
  <proof>
```

Internally we decorate the state components in the statespace with the suffix *-'*, to avoid cluttering the namespace with the simple names that could no longer be used for logical variables otherwise.

We will first consider programs without procedures, later on we will regard procedures without global variables and finally we will get the full pictures: mutually recursive procedures with global variables (including heap).

## 18.2 Basic Examples

We look at few trivialities involving assignment and sequential composition, in order to get an idea of how to work with our formulation of Hoare Logic.

Using the basic rule directly is a bit cumbersome.

**lemma** (in *state-space*)  $\Gamma \vdash \{|N = 5|\} \ 'N ::= 2 * 'N \{|N = 10|\}$   
 $\langle proof \rangle$

**lemma** (in *state-space*)  $\Gamma \vdash \{True\} \ 'N ::= 10 \{|N = 10|\}$   
 $\langle proof \rangle$

**lemma** (in *state-space*)  $\Gamma \vdash \{2 * 'N = 10\} \ 'N ::= 2 * 'N \{|N = 10|\}$   
 $\langle proof \rangle$

**lemma** (in *state-space*)  $\Gamma \vdash \{'N = 5\} \ 'N ::= 2 * 'N \{|N = 10|\}$   
 $\langle proof \rangle$

**lemma** (in *state-space*)  $\Gamma \vdash \{'N + 1 = a + 1\} \ 'N ::= 'N + 1 \{|N = a + 1|\}$   
 $\langle proof \rangle$

**lemma** (in *state-space*)  $\Gamma \vdash \{'N = a\} \ 'N ::= 'N + 1 \{|N = a + 1|\}$   
 $\langle proof \rangle$

**lemma** (in *state-space*)  
**shows**  $\Gamma \vdash \{a = a \wedge b = b\} \ 'M ::= a;; 'N ::= b \{|M = a \wedge 'N = b|\}$   
 $\langle proof \rangle$

**lemma** (in *state-space*)  
**shows**  $\Gamma \vdash \{True\} \ 'M ::= a;; 'N ::= b \{|M = a \wedge 'N = b|\}$   
 $\langle proof \rangle$

**lemma** (in *state-space*)  
**shows**  $\Gamma \vdash \{|M = a \wedge 'N = b\}$   
 $\quad \quad \quad 'I ::= 'M;; 'M ::= 'N;; 'N ::= 'I$   
 $\quad \quad \quad \{|M = b \wedge 'N = a\}$   
 $\langle proof \rangle$

We can also perform verification conditions generation step by step by using the *vcg-step* method.

**lemma** (in *state-space*)  
**shows**  $\Gamma \vdash \{|M = a \wedge 'N = b\}$   
 $\quad \quad \quad 'I ::= 'M;; 'M ::= 'N;; 'N ::= 'I$   
 $\quad \quad \quad \{|M = b \wedge 'N = a\}$   
 $\langle proof \rangle$

In the following assignments we make use of the consequence rule in order to achieve the intended precondition. Certainly, the *vcg* method is able to

handle this case, too.

**lemma** (in *state-space*)  
**shows**  $\Gamma \vdash \{\! \{ M = N \} \! \}$   $M ::= M + 1 \{\! \{ M \neq N \} \! \}$   
*<proof>*

**lemma** (in *state-space*)  
**shows**  $\Gamma \vdash \{\! \{ M = N \} \! \}$   $M ::= M + 1 \{\! \{ M \neq N \} \! \}$   
*<proof>*

**lemma** (in *state-space*)  
**shows**  $\Gamma \vdash \{\! \{ M = N \} \! \}$   $M ::= M + 1 \{\! \{ M \neq N \} \! \}$   
*<proof>*

### 18.3 Multiplication by Addition

We now do some basic examples of actual WHILE programs. This one is a loop for calculating the product of two natural numbers, by iterated addition. We first give detailed structured proof based on single-step Hoare rules.

**lemma** (in *state-space*)  
**shows**  $\Gamma \vdash \{\! \{ M = 0 \wedge S = 0 \} \! \}$   
     WHILE  $M \neq a$   
     DO  $S ::= S + b;; M ::= M + 1$  OD  
      $\{\! \{ S = a * b \} \! \}$   
*<proof>*

The subsequent version of the proof applies the *vcg* method to reduce the Hoare statement to a purely logical problem that can be solved fully automatically. Note that we have to specify the WHILE loop invariant in the original statement.

**lemma** (in *state-space*)  
**shows**  $\Gamma \vdash \{\! \{ M = 0 \wedge S = 0 \} \! \}$   
     WHILE  $M \neq a$   
     INV  $\{\! \{ S = M * b \} \! \}$   
     DO  $S ::= S + b;; M ::= M + 1$  OD  
      $\{\! \{ S = a * b \} \! \}$   
*<proof>*

Here some examples of “breaking” out of a loop

**lemma** (in *state-space*)  
**shows**  $\Gamma \vdash \{\! \{ M = 0 \wedge S = 0 \} \! \}$   
     TRY  
     WHILE *True*  
     INV  $\{\! \{ S = M * b \} \! \}$   
     DO IF  $M = a$  THEN THROW ELSE  $S ::= S + b;; M ::= M + 1$   
 FI OD  
     CATCH  
     SKIP

$END$   
 $\{\{ 'S = a * b \}$   
 $\langle proof \rangle$

**lemma** (*in state-space*)  
**shows**  $\Gamma \vdash \{\{ 'M = 0 \wedge 'S = 0 \}$   
 $TRY$   
 $WHILE True$   
 $INV \{\{ 'S = 'M * b \}$   
 $DO IF 'M = a THEN 'Abr ::= "Break";; THROW$   
 $ELSE 'S ::= 'S + b;; 'M ::= 'M + 1$   
 $FI$   
 $OD$   
 $CATCH$   
 $IF 'Abr = "Break" THEN SKIP ELSE Throw FI$   
 $END$   
 $\{\{ 'S = a * b \}$   
 $\langle proof \rangle$

Some more syntactic sugar, the label statement  $\dots \cdot \dots$  as shorthand for the *TRY-CATCH* above, and the *RAISE* for an state-update followed by a *THROW*.

**lemma** (*in state-space*)  
**shows**  $\Gamma \vdash \{\{ 'M = 0 \wedge 'S = 0 \}$   
 $\{\{ 'Abr = "Break" \} \cdot WHILE True INV \{\{ 'S = 'M * b \}$   
 $DO IF 'M = a THEN RAISE 'Abr ::= "Break"$   
 $ELSE 'S ::= 'S + b;; 'M ::= 'M + 1$   
 $FI$   
 $OD$   
 $\{\{ 'S = a * b \}$   
 $\langle proof \rangle$

**lemma** (*in state-space*)  
**shows**  $\Gamma \vdash \{\{ 'M = 0 \wedge 'S = 0 \}$   
 $TRY$   
 $WHILE True$   
 $INV \{\{ 'S = 'M * b \}$   
 $DO IF 'M = a THEN RAISE 'Abr ::= "Break"$   
 $ELSE 'S ::= 'S + b;; 'M ::= 'M + 1$   
 $FI$   
 $OD$   
 $CATCH$   
 $IF 'Abr = "Break" THEN SKIP ELSE Throw FI$   
 $END$   
 $\{\{ 'S = a * b \}$   
 $\langle proof \rangle$

**lemma** (*in state-space*)  
**shows**  $\Gamma \vdash \{\{ 'M = 0 \wedge 'S = 0 \}$

```

    {Abr = "Break"} · WHILE True
    INV {S = M * b}
    DO IF M = a THEN RAISE Abr ::= "Break"
        ELSE S ::= S + b;; M ::= M + 1
        FI
    OD
    {S = a * b}
⟨proof⟩

```

Blocks

**lemma** (in *state-space*)  
**shows**  $\Gamma \vdash \{T = i\} \text{LOC } T;; T ::= 2 \text{COL } \{T \leq i\}$   
 ⟨proof⟩

## 18.4 Summing Natural Numbers

We verify an imperative program to sum natural numbers up to a given limit. First some functional definition for proper specification of the problem.

**primrec**  
 $sum :: (nat \Rightarrow nat) \Rightarrow nat \Rightarrow nat$   
**where**  
 $sum\ f\ 0 = 0$   
 $| sum\ f\ (Suc\ n) = f\ n + sum\ f\ n$

**syntax**  
 $-sum :: idt \Rightarrow nat \Rightarrow nat \Rightarrow nat$   
 $(\langle SUMM\ -<- . \rightarrow [0, 0, 10]\ 10)$

**syntax-consts**  
 $-sum == sum$

**translations**  
 $SUMM\ j < k.\ b == CONST\ sum\ (\lambda j.\ b)\ k$

The following proof is quite explicit in the individual steps taken, with the *vcg* method only applied locally to take care of assignment and sequential composition. Note that we express intermediate proof obligation in pure logic, without referring to the state space.

**theorem** (in *state-space*)  
**shows**  $\Gamma \vdash \{True\}$   
 $S ::= 0;; T ::= 1;;$   
 $WHILE\ T \neq n$   
 $DO$   
 $S ::= S + T;;$   
 $T ::= T + 1$   
 $OD$   
 $\{S = (SUMM\ j < n.\ j)\}$   
 (is  $\Gamma \vdash - (-;; ?while) -$ )  
 ⟨proof⟩

The next version uses the *vcg* method, while still explaining the resulting proof obligations in an abstract, structured manner.

**theorem** (in *state-space*)  
**shows**  $\Gamma \vdash \{True\}$   
 $\quad 'S ::= 0;; 'I ::= 1;;$   
 $\quad WHILE 'I \neq n$   
 $\quad INV \{ 'S = (SUMM j < 'I. j) \}$   
 $\quad DO$   
 $\quad \quad 'S ::= 'S + 'I;;$   
 $\quad \quad 'I ::= 'I + 1$   
 $\quad OD$   
 $\quad \{ 'S = (SUMM j < n. j) \}$   
 $\langle proof \rangle$

Certainly, this proof may be done fully automatically as well, provided that the invariant is given beforehand.

**theorem** (in *state-space*)  
**shows**  $\Gamma \vdash \{True\}$   
 $\quad 'S ::= 0;; 'I ::= 1;;$   
 $\quad WHILE 'I \neq n$   
 $\quad INV \{ 'S = (SUMM j < 'I. j) \}$   
 $\quad DO$   
 $\quad \quad 'S ::= 'S + 'I;;$   
 $\quad \quad 'I ::= 'I + 1$   
 $\quad OD$   
 $\quad \{ 'S = (SUMM j < n. j) \}$   
 $\langle proof \rangle$

## 18.5 SWITCH

**lemma** (in *state-space*)  
**shows**  $\Gamma \vdash \{ 'N = 5 \} SWITCH 'B$   
 $\quad \quad \{ True \} \Rightarrow 'N ::= 6$   
 $\quad \quad | \{ False \} \Rightarrow 'N ::= 7$   
 $\quad \quad END$   
 $\quad \{ 'N > 5 \}$   
 $\langle proof \rangle$

**lemma** (in *state-space*)  
**shows**  $\Gamma \vdash \{ 'N = 5 \} SWITCH 'N$   
 $\quad \quad \{ v. v < 5 \} \Rightarrow 'N ::= 6$   
 $\quad \quad | \{ v. v \geq 5 \} \Rightarrow 'N ::= 7$   
 $\quad \quad END$   
 $\quad \{ 'N > 5 \}$   
 $\langle proof \rangle$

## 18.6 (Mutually) Recursive Procedures

### 18.6.1 Factorial

We want to define a procedure for the factorial. We first define a HOL functions that calculates it to specify the procedure later on.

```
primrec fac:: nat  $\Rightarrow$  nat
where
fac 0 = 1 |
fac (Suc n) = (Suc n) * fac n
```

```
lemma fac-simp [simp]: 0 < i  $\implies$  fac i = i * fac (i - 1)
  <proof>
```

Now we define the procedure

```
procedures
  Fac (N::nat|R::nat)
  IF 'N = 0 THEN 'R ::= 1
  ELSE 'R ::= CALL Fac('N - 1);
  'R ::= 'N * 'R
  FI
```

```
print-locale Fac-impl
```

To see how a call is syntactically translated you can switch off the printing translation via the configuration option *hoare-use-call-tr'*

```
context Fac-impl
begin
```

'R ::= CALL *Fac*() is internally:

```
declare [[hoare-use-call-tr' = false]]
```

```
call ( $\lambda s. s(\text{locals} := \text{update project-Nat-nat inject-Nat-nat } N\text{'Fac-'} (K\text{-statefun}
(\text{lookup project-Nat-nat } N\text{'Fac-'} (\text{locals } s))) (\text{locals } s))) \text{Fac-'}\text{proc } (\lambda s t.
s(\text{globals} := \text{globals } t)) (\lambda i t. 'R ::= \text{lookup project-Nat-nat } R\text{'Fac-'} (\text{locals }
t))$ )
```

```
term CALL Fac('N,'R)
```

```
declare [[hoare-use-call-tr' = true]]
```

Now let us prove that *Fac* meets its specification.

```
end
```

```
lemma (in Fac-impl) Fac-spec':
  shows  $\forall \sigma. \Gamma, \Theta \vdash \{\sigma\} \text{PROC } \text{Fac}('N, 'R) \{\{ 'R = \text{fac } \sigma N \}\}$ 
  <proof>
```

Since the factorial was implemented recursively, the main ingredient of this proof is, to assume that the specification holds for the recursive call of *Fac* and prove the body correct. The assumption for recursive calls is added to the context by the rule *HoarePartial.ProcRec1* (also derived from general rule for mutually recursive procedures):

$$\begin{aligned} & \llbracket \forall Z. \Gamma, \Theta \cup (\bigcup_Z \{(P\ Z, p, Q\ Z, A\ Z)\}) \vdash_{/F} (P\ Z)\ \text{the } (\Gamma\ p)\ (Q\ Z), (A\ Z); \\ & \quad p \in \text{dom } \Gamma \rrbracket \\ \implies & \forall Z. \Gamma, \Theta \vdash_{/F} (P\ Z)\ \text{Call } p\ (Q\ Z), (A\ Z) \end{aligned}$$

The verification condition generator will infer the specification out of the context when it encounters a recursive call of the factorial.

We can also step through verification condition generation. When the verification condition generator encounters a procedure call it tries to use the rule *ProcSpec*. To be successful there must be a specification of the procedure in the context.

**lemma** (in *Fac-impl*) *Fac-spec1*:  
**shows**  $\forall \sigma. \Gamma, \Theta \vdash \{\sigma\} \ 'R := PROC\ Fac('N) \ \{\!\{R = fac\ \sigma N\}\!\}$   
 $\langle proof \rangle$

Here some Isar style version of the proof

**lemma** (in *Fac-impl*) *Fac-spec2*:  
**shows**  $\forall \sigma. \Gamma, \Theta \vdash \{\sigma\} \ 'R := PROC\ Fac('N) \ \{\!\{R = fac\ \sigma N\}\!\}$   
 $\langle proof \rangle$

To avoid retyping of potentially large pre and postconditions in the previous proof we can use the casual term abbreviations of the Isar language.

**lemma** (in *Fac-impl*) *Fac-spec3*:  
**shows**  $\forall \sigma. \Gamma, \Theta \vdash \{\sigma\} \ 'R := PROC\ Fac('N) \ \{\!\{R = fac\ \sigma N\}\!\}$   
**(is**  $\forall \sigma. \Gamma, \Theta \vdash (?Pre\ \sigma) \ ?Fac\ (?Post\ \sigma)$   
 $\langle proof \rangle$

The previous proof pattern has still some kind of inconvenience. The augmented context is always printed in the proof state. That can mess up the state, especially if we have large specifications. This may be annoying if we want to develop single step or structured proofs. In this case it can be a good idea to introduce a new variable for the augmented context.

**lemma** (in *Fac-impl*) *Fac-spec4*:  
**shows**  $\forall \sigma. \Gamma, \Theta \vdash \{\sigma\} \ 'R := PROC\ Fac('N) \ \{\!\{R = fac\ \sigma N\}\!\}$   
**(is**  $\forall \sigma. \Gamma, \Theta \vdash (?Pre\ \sigma) \ ?Fac\ (?Post\ \sigma)$   
 $\langle proof \rangle$

There are different rules available to prove procedure calls, depending on the kind of postcondition and whether or not the procedure is recursive or

even mutually recursive. See for example *HoareTotal.ProcRec1*, *HoareTotal.ProcNoRec1*. They are all derived from the most general rule *HoareTotal.ProcRec*. All of them have some side-conditions concerning the parameter passing protocol and its relation to the pre and postcondition. They can be solved in a uniform fashion. That's why we have created the method *hoare-rule*, which behaves like the method *rule* but automatically tries to solve the side-conditions.

### 18.6.2 Odd and Even

Odd and even are defined mutually recursive here. In the *procedures* command we conjoin both definitions with *and*.

**procedures**

```

odd(N::nat | A::nat) IF 'N=0 THEN 'A==0
                    ELSE IF 'N=1 THEN CALL even ('N - 1, 'A)
                    ELSE CALL odd ('N - 2, 'A)
                    FI
FI

```

**and**

```

even(N::nat | A::nat) IF 'N=0 THEN 'A==1
                     ELSE IF 'N=1 THEN CALL odd ('N - 1, 'A)
                     ELSE CALL even ('N - 2, 'A)
                     FI
FI

```

**print-theorems**

**print-locale!** *odd-even-clique*

To prove the procedure calls to *odd* respectively *even* correct we first derive a rule to justify that we can assume both specifications to verify the bodies. This rule can be derived from the general *HoareTotal.ProcRec* rule. An ML function will do this work:

$\langle ML \rangle$

**lemma** (in *odd-even-clique*)

```

shows odd-spec:  $\forall \sigma. \Gamma \vdash \{\sigma\} 'A ::= PROC\ odd('N)
                \{\{\exists b. \sigma N = 2 * b + 'A\} \wedge 'A < 2\} \text{ (is ?P1)}
and even-spec:  $\forall \sigma. \Gamma \vdash \{\sigma\} 'A ::= PROC\ even('N)
                \{\{\exists b. \sigma N + 1 = 2 * b + 'A\} \wedge 'A < 2\} \text{ (is ?P2)}$$ 
```

$\langle proof \rangle$

## 18.7 Expressions With Side Effects

**lemma** (in *state-space*) shows  $\Gamma \vdash \{\ True \}$

```

'N  $\gg$  n. 'N ::= 'N + 1  $\gg$ 
'M  $\gg$  m. 'M ::= 'M + 1  $\gg$ 

```

```

'R ::= n + m
{ 'R = 'N + 'M - 2 }
⟨proof⟩

```

**lemma** (in *Fac-impl*) **shows**

```

Γ ⊢ { True }
CALL Fac('N) ≫ n. CALL Fac('N) ≫ m.
'R ::= n + m
{ 'R = fac 'N + fac 'N }
⟨proof⟩

```

**lemma** (in *Fac-impl*) **shows**

```

Γ ⊢ { True }
CALL Fac('N) ≫ n. CALL Fac(n) ≫ m.
'R ::= m
{ 'R = fac (fac 'N) }
⟨proof⟩

```

## 18.8 Global Variables and Heap

Now we will define and verify some procedures on heap-lists. We consider list structures consisting of two fields, a content element *cont* and a reference to the next list element *next*. We model this by the following state space where every field has its own heap.

```

hoarestate globals-list =
  next :: ref ⇒ ref
  cont :: ref ⇒ nat

```

Updates to global components inside a procedure will always be propagated to the caller. This is implicitly done by the parameter passing syntax translations. The record containing the global variables must begin with the prefix "globals".

We will first define an append function on lists. It takes two references as parameters. It appends the list referred to by the first parameter with the list referred to by the second parameter, and returns the result right into the first parameter.

**procedures** (**imports** *globals-list*)

```

append(p::ref,q::ref|p::ref)
  IF 'p=NULL THEN 'p ::= 'q ELSE 'p → 'next ::= CALL append('p→'next,'q)
FI

```

**declare** [[*hoare-use-call-tr'* = *false*]]

**context** *append-impl*

```

begin
term CALL append('p, 'q, 'p→'next)
end
declare [[hoare-use-call-tr' = true]]

```

Below we give two specifications this time.. The first one captures the functional behaviour and focuses on the entities that are potentially modified by the procedure, the second one is a pure frame condition. The list in the modifies clause has to list all global state components that may be changed by the procedure. Note that we know from the modifies clause that the *cont* parts of the lists will not be changed. Also a small side note on the syntax. We use ordinary brackets in the postcondition of the modifies clause, and also the state components do not carry the acute, because we explicitly note the state *t* here.

The functional specification now introduces two logical variables besides the state space variable  $\sigma$ , namely *Ps* and *Qs*. They are universally quantified and range over both the pre and the postcondition, so that we are able to properly instantiate the specification during the proofs. The syntax  $\{\sigma. \dots\}$  is a shorthand to fix the current state:  $\{s. \sigma = s \dots\}$ .

**lemma** (**in** *append-impl*) *append-spec*:

```

shows  $\forall \sigma \ Ps \ Qs. \Gamma \vdash$ 
   $\{\sigma. \text{List } 'p \ 'next \ Ps \wedge \text{List } 'q \ 'next \ Qs \wedge \text{set } Ps \cap \text{set } Qs = \{\}\}$ 
   $\ 'p \ := \ PROC \ append('p, 'q)$ 
   $\ \{\text{List } 'p \ 'next \ (Ps @ Qs) \wedge (\forall x. x \notin \text{set } Ps \longrightarrow 'next \ x = \sigma \ next \ x)\}$ 
  <proof>

```

The modifies clause is equal to a proper record update specification of the following form.

```

lemma (in append-impl) shows  $\{t. t \text{ may-only-modify-globals } Z \text{ in } [next]\}$ 
  =
   $\{t. \exists next. \text{globals } t = \text{update } id \ id \ next \text{' } (K\text{-statefun } next) (\text{globals } Z)\}$ 
  <proof>

```

If the verification condition generator works on a procedure call it checks whether it can find a modifies clause in the context. If one is present the procedure call is simplified before the Hoare rule *HoareTotal.ProcSpec* is applied. Simplification of the procedure call means, that the “copy back” of the global components is simplified. Only those components that occur in the modifies clause will actually be copied back. This simplification is justified by the rule *HoareTotal.ProcModifyReturn*. So after this simplification all global components that do not appear in the modifies clause will be treated as local variables.

You can study the effect of the modifies clause on the following two examples, where we want to prove that (@) does not change the *cont* part of the heap.

**lemma** (**in** *append-impl*)

**shows**  $\Gamma \vdash \{\! \{ 'p = \text{Null} \wedge 'cont = c \} \! \} 'p ::= \text{CALL } \text{append}('p, \text{Null}) \{\! \{ 'cont = c \} \! \}$   
 $\langle \text{proof} \rangle$

To prove the frame condition, we have to tell the verification condition generator to use only the modifies clauses and not to search for functional specifications by the parameter *spec=modifies*. It will also try to solve the verification conditions automatically.

**lemma** (in *append-impl*) *append-modifies*:

**shows**  
 $\forall \sigma. \Gamma \vdash \{ \sigma \} 'p ::= \text{PROC } \text{append}('p, 'q) \{ t. t \text{ may-only-modify-globals } \sigma \text{ in } [next] \}$   
 $\langle \text{proof} \rangle$

**lemma** (in *append-impl*)

**shows**  $\Gamma \vdash \{\! \{ 'p = \text{Null} \wedge 'cont = c \} \! \} 'p \rightarrow 'next ::= \text{CALL } \text{append}('p, \text{Null}) \{\! \{ 'cont = c \} \! \}$   
 $\langle \text{proof} \rangle$

Of course we could add the modifies clause to the functional specification as well. But separating both has the advantage that we split up the verification work. We can make use of the modifies clause before we apply the functional specification in a fully automatic fashion.

To verify the body of (@) we do not need the modifies clause, since the specification does not talk about *cont* at all, and we don't access *cont* inside the body. This may be different for more complex procedures.

To prove that a procedure respects the modifies clause, we only need the modifies clauses of the procedures called in the body. We do not need the functional specifications. So we can always prove the modifies clause without functional specifications, but we may need the modifies clause to prove the functional specifications.

### 18.8.1 Insertion Sort

**primrec** *sorted*:: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  bool

**where**

*sorted le* [] = True |

*sorted le* (x#xs) = (( $\forall y \in \text{set } xs. \text{le } x y$ )  $\wedge$  *sorted le* xs)

**procedures** (imports *globals-list*)

*insert*(*r*::ref, *p*::ref | *p*::ref)

IF *r*=Null THEN SKIP

ELSE IF '*p*=Null THEN '*p* ::= '*r*; '*p*  $\rightarrow$  '*next* ::= Null

ELSE IF '*r*  $\rightarrow$  '*cont*  $\leq$  '*p*  $\rightarrow$  '*cont*

THEN '*r*  $\rightarrow$  '*next* ::= '*p*; '*p* ::= '*r*

ELSE '*p*  $\rightarrow$  '*next* ::= CALL *insert*('r, '*p*  $\rightarrow$  '*next*)

FI

*FI*  
*FI*

In the postcondition of the functional specification there is a small but important subtlety. Whenever we talk about the *cont* part we refer to the one of the pre-state, even in the conclusion of the implication. The reason is, that we have separated out, that *cont* is not modified by the procedure, to the modifies clause. So whenever we talk about unmodified parts in the postcondition we have to use the pre-state part, or explicitly state an equality in the postcondition. The reason is simple. If the postcondition would talk about  $\prime cont$  instead of  $\sigma cont$ , we will get a new instance of *cont* during verification and the postcondition would only state something about this new instance. But as the verification condition generator will use the modifies clause the caller of *insert* instead will still have the old *cont* after the call. That's the sense of the modifies clause. So the caller and the specification will simply talk about two different things, without being able to relate them (unless an explicit equality is added to the specification).

**lemma** (in *insert-impl*) *insert-modifies*:  
 $\forall \sigma. \Gamma \vdash \{ \sigma \} \prime p ::= PROC \text{insert}(\prime r, \prime p) \{ t. t \text{ may-only-modify-globals } \sigma \text{ in } [next] \}$   
*<proof>*

**lemma** (in *insert-impl*) *insert-spec*:  
 $\forall \sigma \text{ Ps} . \Gamma \vdash \{ \sigma. List \prime p \prime next \text{ Ps} \wedge sorted (\leq) (map \prime cont \text{ Ps}) \wedge$   
 $\prime r \neq Null \wedge \prime r \notin set \text{ Ps} \}$   
 $\prime p ::= PROC \text{insert}(\prime r, \prime p)$   
 $\{ \exists Qs. List \prime p \prime next Qs \wedge sorted (\leq) (map \sigma cont Qs) \wedge$   
 $set Qs = insert \sigma r (set \text{ Ps}) \wedge$   
 $(\forall x. x \notin set Qs \longrightarrow \prime next x = \sigma next x) \}$

*<proof>*

**procedures** (**imports** *globals-list*)  
*insertSort*(*p::ref* | *p::ref*)  
**where** *r::ref* *q::ref*  
**in**  
 $\prime r ::= Null;$   
**WHILE** ( $\prime p \neq Null$ ) **DO**  
 $\prime q ::= \prime p;$   
 $\prime p ::= \prime p \rightarrow \prime next;$   
 $\prime r ::= CALL \text{insert}(\prime q, \prime r)$   
**OD**;  
 $\prime p ::= \prime r$

**print-locale** *insertSort-impl*

**lemma** (in *insertSort-impl*) *insertSort-modifies*:

**shows**  
 $\forall \sigma. \Gamma \vdash \{\sigma\} \text{'p} ::= \text{PROC insertSort}(\text{'p})$   
 $\{t. t \text{ may-only-modify-globals } \sigma \text{ in } [\text{next}]\}$   
 $\langle \text{proof} \rangle$

Insertion sort is not implemented recursively here but with a while loop. Note that the while loop is not annotated with an invariant in the procedure definition. The invariant only comes into play during verification. Therefore we will annotate the body during the proof with the rule *Hoare-Total.annotateI*.

**lemma** (in *insertSort-impl*) *insertSort-body-spec*:  
**shows**  $\forall \sigma \text{ Ps. } \Gamma, \Theta \vdash \{\sigma. \text{List 'p 'next Ps}\}$   
 $\text{'p} ::= \text{PROC insertSort}(\text{'p})$   
 $\{\exists Qs. \text{List 'p 'next Qs} \wedge \text{sorted } (\leq) (\text{map } \sigma \text{cont } Qs) \wedge$   
 $\text{set } Qs = \text{set } Ps\}$   
 $\langle \text{proof} \rangle$

## 18.8.2 Memory Allocation and Deallocation

The basic idea of memory management is to keep a list of allocated references in the state space. Allocation of a new reference adds a new reference to the list deallocation removes a reference. Moreover we keep a counter "free" for the free memory.

**hoarestate** *globals-list-alloc* =  
 $\text{alloc}::\text{ref list}$   
 $\text{free}::\text{nat}$   
 $\text{next}::\text{ref} \Rightarrow \text{ref}$   
 $\text{cont}::\text{ref} \Rightarrow \text{nat}$

**hoarestate** *locals-list-alloc* =  
 $i::\text{nat}$   
 $\text{first}::\text{ref}$   
 $p::\text{ref}$   
 $q::\text{ref}$   
 $r::\text{ref}$   
 $\text{root}::\text{ref}$   
 $\text{tmp}::\text{ref}$

**locale** *list-alloc* = *globals-list-alloc* + *locals-list-alloc*

**definition**  $\text{sz} = (2::\text{nat})$

**lemma** (in *list-alloc*)  
**shows**  
 $\Gamma, \Theta \vdash \{i = 0 \wedge \text{'first} = \text{Null} \wedge n * \text{sz} \leq \text{'free}\}$   
 $\text{WHILE } i < n$   
 $\text{INV } \{\exists Ps. \text{List 'first 'next Ps} \wedge \text{length Ps} = i \wedge i \leq n \wedge$   
 $\text{set Ps} \subseteq \text{set 'alloc} \wedge (n - i) * \text{sz} \leq \text{'free}\}$   
 $\text{DO}$   
 $\text{'p} ::= \text{NEW } \text{sz} [\text{'cont}::=0, \text{'next}::= \text{Null}];;$

$$\begin{array}{l}
\quad 'p \rightarrow 'next ::= 'first;; \\
\quad 'first ::= 'p;; \\
\quad 'i ::= 'i + 1 \\
\text{OD} \\
\{\exists Ps. \text{List } 'first \ 'next \ Ps \wedge \text{length } Ps = n \wedge \text{set } Ps \subseteq \text{set } 'alloc\} \\
\langle \text{proof} \rangle
\end{array}$$

**lemma** (in *list-alloc*)

**shows**

$$\begin{array}{l}
\Gamma \vdash \{\ 'i = 0 \wedge 'first = \text{Null} \wedge n * sz \leq 'free \} \\
\quad \text{WHILE } 'i < n \\
\quad \text{INV } \{\exists Ps. \text{List } 'first \ 'next \ Ps \wedge \text{length } Ps = 'i \wedge 'i \leq n \wedge \\
\quad \quad \text{set } Ps \subseteq \text{set } 'alloc \wedge (n - 'i) * sz \leq 'free \} \\
\quad \text{DO} \\
\quad \quad 'p ::= \text{NNEW } sz \ [ 'cont ::= 0, 'next ::= \text{Null} ];; \\
\quad \quad 'p \rightarrow 'next ::= 'first;; \\
\quad \quad 'first ::= 'p;; \\
\quad \quad 'i ::= 'i + 1 \\
\quad \text{OD} \\
\{\exists Ps. \text{List } 'first \ 'next \ Ps \wedge \text{length } Ps = n \wedge \text{set } Ps \subseteq \text{set } 'alloc\}
\end{array}$$

$\langle \text{proof} \rangle$

## 18.9 Fault Avoiding Semantics

If we want to ensure that no runtime errors occur we can insert guards into the code. We will not be able to prove any nontrivial Hoare triple about code with guards, if we cannot show that the guards will never fail. A trivial Hoare triple is one with an empty precondition.

**lemma** (in *list-alloc*)  $\Gamma, \Theta \vdash \{\ \text{True} \} \ \{\ 'p \neq \text{Null} \} \mapsto 'p \rightarrow 'next ::= 'p \ \{\ \text{True} \}$   
 $\langle \text{proof} \rangle$

**lemma** (in *list-alloc*)  $\Gamma, \Theta \vdash \{\ \} \ \{\ 'p \neq \text{Null} \} \mapsto 'p \rightarrow 'next ::= 'p \ \{\ \text{True} \}$   
 $\langle \text{proof} \rangle$

Let us consider this small program that reverts a list. At first without guards.

**lemma** (in *list-alloc*)

**shows**

$$\begin{array}{l}
\Gamma, \Theta \vdash \{\ \text{List } 'p \ 'next \ Ps \wedge \text{List } 'q \ 'next \ Qs \wedge \text{set } Ps \cap \text{set } Qs = \{\} \wedge \\
\quad \text{set } Ps \subseteq \text{set } 'alloc \wedge \text{set } Qs \subseteq \text{set } 'alloc \} \\
\quad \text{WHILE } 'p \neq \text{Null} \\
\quad \text{INV } \{\exists ps \ qs. \text{List } 'p \ 'next \ ps \wedge \text{List } 'q \ 'next \ qs \wedge \text{set } ps \cap \text{set } qs = \{\} \wedge \\
\quad \quad \text{rev } ps \ @ \ qs = \text{rev } Ps \ @ \ Qs \wedge \\
\quad \quad \text{set } ps \subseteq \text{set } 'alloc \wedge \text{set } qs \subseteq \text{set } 'alloc \} \\
\quad \text{DO } 'r ::= 'p;; \\
\quad \quad 'p ::= 'p \rightarrow 'next;;
\end{array}$$

$$\begin{array}{l}
r \rightarrow 'next ::= 'q;; \\
'q ::= 'r \text{ OD} \\
\{\{List\ 'q\ 'next\ (rev\ Ps\ @\ Qs)\ \wedge\ set\ Ps \subseteq set\ 'alloc\ \wedge\ set\ Qs \subseteq set\ 'alloc\}\} \\
\langle proof \rangle
\end{array}$$

If we want to ensure that we do not dereference *Null* or access unallocated memory, we have to add some guards.

**lemma** (in *list-alloc*)

**shows**

$$\begin{array}{l}
\Gamma, \Theta \vdash \{\{List\ 'p\ 'next\ Ps \wedge List\ 'q\ 'next\ Qs \wedge set\ Ps \cap set\ Qs = \{\}\ \wedge \\
\quad set\ Ps \subseteq set\ 'alloc \wedge set\ Qs \subseteq set\ 'alloc\}\} \\
WHILE\ 'p \neq Null \\
INV\ \{\{\exists\ ps\ qs.\ List\ 'p\ 'next\ ps \wedge List\ 'q\ 'next\ qs \wedge set\ ps \cap set\ qs = \{\}\ \wedge \\
\quad rev\ ps\ @\ qs = rev\ Ps\ @\ Qs \wedge \\
\quad set\ ps \subseteq set\ 'alloc \wedge set\ qs \subseteq set\ 'alloc\}\} \\
DO\ 'r ::= 'p;; \\
\quad \{\{p \neq Null \wedge p \in set\ 'alloc\}\} \mapsto 'p ::= 'p \rightarrow 'next;; \\
\quad \{\{r \neq Null \wedge r \in set\ 'alloc\}\} \mapsto 'r \rightarrow 'next ::= 'q;; \\
\quad 'q ::= 'r \text{ OD} \\
\{\{List\ 'q\ 'next\ (rev\ Ps\ @\ Qs)\ \wedge set\ Ps \subseteq set\ 'alloc \wedge set\ Qs \subseteq set\ 'alloc\}\} \\
\langle proof \rangle
\end{array}$$

We can also just prove that no faults will occur, by giving the trivial post-condition.

**lemma** (in *list-alloc*) *rev-noFault*:

**shows**

$$\begin{array}{l}
\Gamma, \Theta \vdash \{\{List\ 'p\ 'next\ Ps \wedge List\ 'q\ 'next\ Qs \wedge set\ Ps \cap set\ Qs = \{\}\ \wedge \\
\quad set\ Ps \subseteq set\ 'alloc \wedge set\ Qs \subseteq set\ 'alloc\}\} \\
WHILE\ 'p \neq Null \\
INV\ \{\{\exists\ ps\ qs.\ List\ 'p\ 'next\ ps \wedge List\ 'q\ 'next\ qs \wedge set\ ps \cap set\ qs = \{\}\ \wedge \\
\quad rev\ ps\ @\ qs = rev\ Ps\ @\ Qs \wedge \\
\quad set\ ps \subseteq set\ 'alloc \wedge set\ qs \subseteq set\ 'alloc\}\} \\
DO\ 'r ::= 'p;; \\
\quad \{\{p \neq Null \wedge p \in set\ 'alloc\}\} \mapsto 'p ::= 'p \rightarrow 'next;; \\
\quad \{\{r \neq Null \wedge r \in set\ 'alloc\}\} \mapsto 'r \rightarrow 'next ::= 'q;; \\
\quad 'q ::= 'r \text{ OD} \\
UNIV, UNIV \\
\langle proof \rangle
\end{array}$$

**lemma** (in *list-alloc*) *rev-moduloGuards*:

**shows**

$$\begin{array}{l}
\Gamma, \Theta \vdash / \{\{True\}\} \{\{List\ 'p\ 'next\ Ps \wedge List\ 'q\ 'next\ Qs \wedge set\ Ps \cap set\ Qs = \{\}\ \wedge \\
\quad set\ Ps \subseteq set\ 'alloc \wedge set\ Qs \subseteq set\ 'alloc\}\} \\
WHILE\ 'p \neq Null \\
INV\ \{\{\exists\ ps\ qs.\ List\ 'p\ 'next\ ps \wedge List\ 'q\ 'next\ qs \wedge set\ ps \cap set\ qs = \{\}\ \wedge \\
\quad rev\ ps\ @\ qs = rev\ Ps\ @\ Qs \wedge \\
\quad set\ ps \subseteq set\ 'alloc \wedge set\ qs \subseteq set\ 'alloc\}\} \\
DO\ 'r ::= 'p;;
\end{array}$$

$$\begin{aligned} & \{\{p \neq \text{Null} \wedge p \in \text{set } 'alloc\}\} \checkmark \mapsto p := p \rightarrow 'next;; \\ & \{\{r \neq \text{Null} \wedge r \in \text{set } 'alloc\}\} \checkmark \mapsto r \rightarrow 'next := q;; \\ & q := r \text{ OD} \\ & \{List\ 'q\ 'next\ (rev\ Ps\ @\ Qs) \wedge \text{set } Ps \subseteq \text{set } 'alloc \wedge \text{set } Qs \subseteq \text{set } 'alloc\} \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *CombineStrip'*:  
**assumes** *deriv*:  $\Gamma, \Theta \vdash /_F P\ c'\ Q, A$   
**assumes** *deriv-strip*:  $\Gamma, \Theta \vdash /_{\{\}} P\ c''\ UNIV, UNIV$   
**assumes** *c''*:  $c'' = \text{mark-guards False (strip-guards } (-F) c')$   
**assumes** *c*:  $c = \text{mark-guards False } c'$   
**shows**  $\Gamma, \Theta \vdash /_{\{\}} P\ c\ Q, A$   
 $\langle \text{proof} \rangle$

We can then combine the prove that no fault will occur with the functional prove of the programm without guards to get the full prove by the rule  

$$\begin{aligned} & \llbracket ?\Gamma, ?\Theta \vdash /_{?F} ?P\ ?c\ ?Q, ?A; ?\Gamma, ?\Theta \vdash ?P\ \text{strip-guards } (-\ ?F)\ ?c\ UNIV, UNIV \rrbracket \\ & \implies ?\Gamma, ?\Theta \vdash ?P\ ?c\ ?Q, ?A \end{aligned}$$

**lemma** (in *list-alloc*)  
**shows**  
 $\Gamma, \Theta \vdash \{List\ 'p\ 'next\ Ps \wedge List\ 'q\ 'next\ Qs \wedge \text{set } Ps \cap \text{set } Qs = \{\} \wedge$   
 $\text{set } Ps \subseteq \text{set } 'alloc \wedge \text{set } Qs \subseteq \text{set } 'alloc\}$   
*WHILE*  $'p \neq \text{Null}$   
*INV*  $\{\{\exists ps\ qs. List\ 'p\ 'next\ ps \wedge List\ 'q\ 'next\ qs \wedge \text{set } ps \cap \text{set } qs = \{\} \wedge$   
 $rev\ ps\ @\ qs = rev\ Ps\ @\ Qs \wedge$   
 $\text{set } ps \subseteq \text{set } 'alloc \wedge \text{set } qs \subseteq \text{set } 'alloc\}$   
*DO*  $'r := 'p;$   
 $\{\{p \neq \text{Null} \wedge p \in \text{set } 'alloc\}\} \mapsto p := p \rightarrow 'next;;$   
 $\{\{r \neq \text{Null} \wedge r \in \text{set } 'alloc\}\} \mapsto r \rightarrow 'next := q;;$   
 $q := r \text{ OD}$   
 $\{List\ 'q\ 'next\ (rev\ Ps\ @\ Qs) \wedge \text{set } Ps \subseteq \text{set } 'alloc \wedge \text{set } Qs \subseteq \text{set } 'alloc\}$   
 $\langle \text{proof} \rangle$

In the previous example the effort to split up the prove did not really pay off. But when we think of programs with a lot of guards and complicated specifications it may be better to first focus on a prove without the messy guards. Maybe it is possible to automate the no fault proofs so that it suffices to focus on the stripped program.

**context** *list-alloc*  
**begin**

The purpose of guards is to watch for faults that can occur during evaluation of expressions. In the example before we watched for null pointer dereferencing or memory faults. We can also look for array index bounds or

division by zero. As the condition of a while loop is evaluated in each iteration we cannot just add a guard before the while loop. Instead we need a special guard for the condition. Example:  $WHILE (False, \{\!| p \neq Null \!\}) \vdash \rightarrow p \rightarrow next \neq Null DO SKIP OD$

end

## 18.10 Cicular Lists

**definition**

$distPath :: ref \Rightarrow (ref \Rightarrow ref) \Rightarrow ref \Rightarrow ref list \Rightarrow bool$  **where**  
 $distPath x next y as = (Path x next y as \wedge distinct as)$

**lemma** *neq-dP*:  $\llbracket p \neq q; Path p h q Ps; distinct Ps \rrbracket \implies$   
 $\exists Qs. p \neq Null \wedge Ps = p \# Qs \wedge p \notin set Qs$   
*<proof>*

**lemma** (*in list-alloc*) *circular-list-rev-I*:

$\Gamma, \Theta \vdash \{\!| root = r \wedge distPath root next root (r \# Ps) \!\}$   
 $p := root;; q := root \rightarrow next;;$   
 $WHILE q \neq root$   
 $INV \{\!| \exists ps qs. distPath p next root ps \wedge distPath q next root qs \wedge$   
 $root = r \wedge r \neq Null \wedge r \notin set Ps \wedge set ps \cap set qs = \{\} \wedge$   
 $Ps = (rev ps) @ qs \!\}$   
 $DO tmp := q;; q := q \rightarrow next;; tmp \rightarrow next := p;; p := tmp OD;;$   
 $root \rightarrow next := p$   
 $\{\!| root = r \wedge distPath root next root (r \# rev Ps) \!\}$   
*<proof>*

**lemma** *path-is-list*:  $\llbracket a next b. Path b next a Ps ; a \notin set Ps; a \neq Null \rrbracket$   
 $\implies List b (next(a := Null)) (Ps @ [a])$   
*<proof>*

The simple algorithm for acyclic list reversal, with modified annotations, works for cyclic lists as well.:

**lemma** (*in list-alloc*) *circular-list-rev-II*:

$\Gamma, \Theta \vdash$   
 $\{\!| p = r \wedge distPath p next p (r \# Ps) \!\}$   
 $q := Null;;$   
 $WHILE p \neq Null$   
 $INV$   
 $\{\!| ((q = Null) \longrightarrow (\exists ps. distPath p next r ps \wedge ps = r \# Ps)) \wedge$   
 $((q \neq Null) \longrightarrow (\exists ps qs. distPath q next r qs \wedge List p next ps \wedge$   
 $set ps \cap set qs = \{\} \wedge rev qs @ ps = Ps @ [r])) \wedge$   
 $\neg (p = Null \wedge q = Null \wedge r = Null)$   
 $\!\}$

$DO$   
 $\quad 'tmp := 'p;; 'p := 'p \rightarrow 'next;; 'tmp \rightarrow 'next := 'q;; 'q := 'tmp$   
 $OD$   
 $\quad \{\{ 'q = r \wedge distPath 'q 'next 'q (r \# rev Ps) \}\}$

$\langle proof \rangle$

Although the above algorithm is more succinct, its invariant looks more involved. The reason for the case distinction on  $q$  is due to the fact that during execution, the pointer variables can point to either cyclic or acyclic structures.

When working on lists, its sometimes better to remove *fun-upd-apply* from the simpset, and instead include *fun-upd-same* and *fun-upd-other* to the simpset

**lemma** (in *state-space*)  $\Gamma \vdash \{\sigma\}$   
 $\quad 'I := 'M;;$   
 $\quad ANNO \tau. \{\{\tau. 'I = {}^\sigma M\}\}$   
 $\quad \quad 'M := 'N;; 'N := 'I$   
 $\quad \quad \{\{ 'M = {}^\tau N \wedge 'N = {}^\tau I \}\}$   
 $\quad \quad \{\{ 'M = {}^\sigma N \wedge 'N = {}^\sigma M \}\}$

$\langle proof \rangle$

**context** *state-space*

**begin**

**term**  $ANNO (\tau, m, k). (\{\{\tau. 'M = m\}\}) 'M := 'N;; 'N := 'I \{\{ 'M = {}^\tau N \ \& \ 'N = {}^\tau I \}, \{\}$

**end**

**lemma** (in *state-space*)  $\Gamma \vdash (\{\sigma\} \cap \{\{ 'M = 0 \wedge 'S = 0 \}\})$   
 $\quad (ANNO \tau. (\{\{\tau\} \cap \{\{ 'A = {}^\sigma A \wedge 'I = {}^\sigma I \wedge 'M = 0 \wedge 'S = 0 \}\}\})$   
 $\quad \quad WHILE 'M \neq 'A$   
 $\quad \quad INV \{\{ 'S = 'M * 'I \wedge 'A = {}^\tau A \wedge 'I = {}^\tau I \}\}$   
 $\quad \quad DO 'S := 'S + 'I;; 'M := 'M + 1 OD$   
 $\quad \quad \{\{ 'S = {}^\tau A * {}^\tau I \}\}$   
 $\quad \quad \{\{ 'S = {}^\sigma A * {}^\sigma I \}\}$

$\langle proof \rangle$

Just some test on marked, guards

**lemma** (in *state-space*)  $\Gamma \vdash \{\{ True \}\} WHILE \{\{ P 'N \} \}_\vee, \{\{ Q 'M \} \}_\#, \{\{ R 'N \} \} \mapsto 'N < 'M$

$\quad \quad INV \{\{ 'N < 2 \}\} DO$   
 $\quad \quad 'N := 'M$   
 $\quad \quad OD$

$\{\{ hard \}\}$

$\langle proof \rangle$

**lemma** (in *state-space*)  $\Gamma \vdash / \{\{ True \}\} \{\{ True \}\} WHILE \{\{ P 'N \} \}_\vee, \{\{ Q 'M \} \}_\#, \{\{ R 'N \} \} \mapsto 'N < 'M$

```

          INV { 'N < 2 } DO
            'N ::= 'M
          OD
    {hard}
<proof>

```

**end**

## 19 Examples for Total Correctness

**theory** *VcgExTotal* **imports** *../HeapList ../Vcg* **begin**

**record** 'g vars = 'g state +

```

  A-' :: nat
  I-' :: nat
  M-' :: nat
  N-' :: nat
  R-' :: nat
  S-' :: nat
  Abr-': string

```

**lemma**  $\Gamma \vdash_t \{ 'M = 0 \wedge 'S = 0 \}$   
 $WHILE\ 'M \neq a$   
 $INV\ \{ 'S = 'M * b \wedge 'M \leq a \}$   
 $VAR\ MEASURE\ a - 'M$   
 $DO\ 'S ::= 'S + b;; 'M ::= 'M + 1\ OD$   
 $\{ 'S = a * b \}$   
<proof>

**lemma**  $\Gamma \vdash_t \{ 'I \leq 3 \}$   
 $WHILE\ 'I < 10\ INV\ \{ 'I \leq 10 \}\ VAR\ MEASURE\ 10 - 'I$   
 $DO$   
 $'I ::= 'I + 1$   
 $OD$   
 $\{ 'I = 10 \}$   
<proof>

Total correctness of a nested loop. In the inner loop we have to express that the loop variable of the outer loop is not changed. We use *FIX* to introduce a new logical variable

**lemma**  $\Gamma \vdash_t \{ 'M=0 \wedge 'N=0 \}$   
 $WHILE\ ('M < i)$   
 $INV\ \{ 'M \leq i \wedge ('M \neq 0 \longrightarrow 'N = j) \wedge 'N \leq j \}$   
 $VAR\ MEASURE\ (i - 'M)$   
 $DO$   
 $'N ::= 0;;$   
 $WHILE\ ('N < j)$

FIX  $m$ .  
 INV  $\{\! \{ 'M=m \wedge 'N \leq j \} \!\}$   
 VAR MEASURE  $(j - 'N)$   
 DO  
    $'N ::= 'N + 1$   
 OD;;  
 $'M ::= 'M + 1$   
 OD  
 $\{\! \{ 'M=i \wedge ('M \neq 0 \longrightarrow 'N=j) \} \!\}$   
 $\langle \text{proof} \rangle$

**primrec**  $\text{fac} :: \text{nat} \Rightarrow \text{nat}$   
**where**  
 $\text{fac } 0 = 1$  |  
 $\text{fac } (\text{Suc } n) = (\text{Suc } n) * \text{fac } n$

**lemma**  $\text{fac-simp}$  [ $\text{simp}$ ]:  $0 < i \Longrightarrow \text{fac } i = i * \text{fac } (i - 1)$   
 $\langle \text{proof} \rangle$

**procedures**  
 $\text{Fac } (N \mid R) = \text{IF } 'N = 0 \text{ THEN } 'R ::= 1$   
                    $\text{ELSE CALL } \text{Fac}('N - 1, 'R);;$   
                    $'R ::= 'N * 'R$   
 FI

**lemma** (**in**  $\text{Fac-impl}$ )  $\text{Fac-spec}$ :  
**shows**  $\forall n. \Gamma \vdash_t \{\! \{ 'N=n \} \!\} 'R ::= \text{PROC } \text{Fac}('N) \{\! \{ 'R = \text{fac } n \} \!\}$   
 $\langle \text{proof} \rangle$

**procedures**  
 $\text{p91}(R, N \mid R) = \text{IF } 100 < 'N \text{ THEN } 'R ::= 'N - 10$   
                    $\text{ELSE } 'R ::= \text{CALL } \text{p91}('R, 'N+11);;$   
                    $'R ::= \text{CALL } \text{p91}('R, 'R) \text{ FI}$

$\text{p91-spec}: \forall n. \Gamma \vdash_t \{\! \{ 'N=n \} \!\} 'R ::= \text{PROC } \text{p91}('R, 'N)$   
                    $\{\! \{ \text{if } 100 < n \text{ then } 'R = n - 10 \text{ else } 'R = 91 \} \!\}, \{\}$

**lemma** (**in**  $\text{p91-impl}$ )  $\text{p91-spec}$ :  
**shows**  $\forall \sigma. \Gamma \vdash_t \{\sigma\} 'R ::= \text{PROC } \text{p91}('R, 'N)$   
                    $\{\! \{ \text{if } 100 < {}^\sigma N \text{ then } 'R = {}^\sigma N - 10 \text{ else } 'R = 91 \} \!\}, \{\}$   
 $\langle \text{proof} \rangle$

**record**  $\text{globals-list} =$   
    $\text{next-}' :: \text{ref} \Rightarrow \text{ref}$   
    $\text{cont-}' :: \text{ref} \Rightarrow \text{nat}$

**record** 'g list-vars = 'g state +

p-' :: ref  
q-' :: ref  
r-' :: ref  
root-' :: ref  
tmp-' :: ref

**procedures**

append(p,q|p) =  
IF 'p=Null THEN 'p ::= 'q ELSE 'p→'next ::= CALL append('p→'next,'q)  
FI

**lemma** (in append-impl)

shows

$\forall \sigma Ps Qs. \Gamma \vdash_t$   
 $\{\sigma. List\ 'p\ 'next\ Ps \wedge List\ 'q\ 'next\ Qs \wedge set\ Ps \cap set\ Qs = \{\}\}$   
 $'p ::= PROC\ append('p, 'q)$   
 $\{List\ 'p\ 'next\ (Ps @ Qs) \wedge (\forall x. x \notin set\ Ps \longrightarrow 'next\ x = \sigma_{next}\ x)\}$   
 $\langle proof \rangle$

**lemma** (in append-impl)

shows

$\forall \sigma Ps Qs. \Gamma \vdash_t$   
 $\{\sigma. List\ 'p\ 'next\ Ps \wedge List\ 'q\ 'next\ Qs \wedge set\ Ps \cap set\ Qs = \{\}\}$   
 $'p ::= PROC\ append('p, 'q)$   
 $\{List\ 'p\ 'next\ (Ps @ Qs) \wedge (\forall x. x \notin set\ Ps \longrightarrow 'next\ x = \sigma_{next}\ x)\}$   
 $\langle proof \rangle$

**lemma** (in append-impl)

shows

append-spec:  
 $\forall \sigma. \Gamma \vdash_t (\{\sigma\} \cap \{islist\ 'p\ 'next\})\ 'p ::= PROC\ append('p, 'q)$   
 $\{\forall Ps Qs. List\ \sigma_p\ \sigma_{next}\ Ps \wedge List\ \sigma_q\ \sigma_{next}\ Qs \wedge set\ Ps \cap set\ Qs = \{\}$   
 $\longrightarrow$   
 $List\ 'p\ 'next\ (Ps @ Qs) \wedge (\forall x. x \notin set\ Ps \longrightarrow 'next\ x = \sigma_{next}\ x)\}$   
 $\langle proof \rangle$

**lemma**  $\Gamma \vdash \{List\ 'p\ 'next\ Ps\}$

'q ::= Null;;  
WHILE 'p ≠ Null INV  $\{\exists Ps' Qs'. List\ 'p\ 'next\ Ps' \wedge List\ 'q\ 'next\ Qs' \wedge$   
 $set\ Ps' \cap set\ Qs' = \{\} \wedge$   
 $rev\ Ps' @ Qs' = rev\ Ps\}$

DO

'r ::= 'p;; 'p ::= 'p→'next;;  
'r→'next ::= 'q;; 'q ::= 'r

OD;;

'p ::= 'q  
 $\{List\ 'p\ 'next\ (rev\ Ps)\}$

$\langle proof \rangle$

**lemma** *conjI2*:  $\llbracket Q; Q \implies P \rrbracket \implies P \wedge Q$   
 $\langle proof \rangle$

**procedures** *Rev*( $p|p$ ) =

```
'q ::= Null;;  
WHILE 'p ≠ Null  
DO  
  'r ::= 'p;; {'p ≠ Null'} → 'p ::= 'p → 'next;;  
  {'r ≠ Null'} → 'r → 'next ::= 'q;; 'q ::= 'r  
OD;;  
'p ::= 'q
```

*Rev-spec*:

$\forall Ps. \Gamma \vdash_t \{ \text{List } 'p \text{ 'next } Ps \} \text{ 'p} ::= \text{PROC } \text{Rev}('p) \{ \text{List } 'p \text{ 'next } (\text{rev } Ps) \}$

*Rev-modifies*:

$\forall \sigma. \Gamma \vdash_{/UNIV} \{ \sigma \} \text{ 'p} ::= \text{PROC } \text{Rev}('p) \{ t. t \text{ may-only-modify-globals } \sigma \text{ in } [next] \}$

We only need partial correctness of modifies clause!

**lemma** *upd-hd-next*:

**assumes**  $p\text{-ps}$ :  $\text{List } p \text{ next } (p \# ps)$

**shows**  $\text{List } (\text{next } p) (\text{next}(p := q)) ps$

$\langle proof \rangle$

**lemma** (in *Rev-impl*) **shows**

*Rev-spec*:

$\forall Ps. \Gamma \vdash_t \{ \text{List } 'p \text{ 'next } Ps \} \text{ 'p} ::= \text{PROC } \text{Rev}('p) \{ \text{List } 'p \text{ 'next } (\text{rev } Ps) \}$

$\langle proof \rangle$

**lemma** (in *Rev-impl*) **shows**

*Rev-modifies*:

$\forall \sigma. \Gamma \vdash_{/UNIV} \{ \sigma \} \text{ 'p} ::= \text{PROC } \text{Rev}('p) \{ t. t \text{ may-only-modify-globals } \sigma \text{ in } [next] \}$

$\langle proof \rangle$

**lemma**  $\Gamma \vdash_t \{ \text{List } 'p \text{ 'next } Ps \}$

```
'q ::= Null;;  
WHILE 'p ≠ Null INV {  $\exists Ps' Qs'. \text{List } 'p \text{ 'next } Ps' \wedge \text{List } 'q \text{ 'next } Qs' \wedge$   
   $\text{set } Ps' \cap \text{set } Qs' = \{ \} \wedge$   
   $\text{rev } Ps' @ Qs' = \text{rev } Ps \}$   
VAR MEASURE (length (list 'p 'next) )  
DO  
  'r ::= 'p;; 'p ::= 'p → 'next;;  
  'r → 'next ::= 'q;; 'q ::= 'r  
OD;;  
'p ::= 'q  
{List 'p 'next (rev Ps)}
```

*<proof>*

**procedures**

$pedal(N,M) = IF\ 0 < 'N\ THEN$   
     $IF\ 0 < 'M\ THEN\ CALL\ coast('N-1, 'M-1)\ FI;;$   
     $CALL\ pedal('N-1, 'M)$   
 $FI$

**and**

$coast(N,M) = CALL\ pedal('N, 'M);;$   
     $IF\ 0 < 'M\ THEN\ CALL\ coast('N, 'M-1)\ FI$

*<ML>*

**lemma (in pedal-coast-clique)**

**shows**  $(\Gamma \vdash_t \{\!\{ True \}\!\})\ PROC\ pedal('N, 'M)\ \{\!\{ True \}\!\} \wedge$   
     $(\Gamma \vdash_t \{\!\{ True \}\!\})\ PROC\ coast('N, 'M)\ \{\!\{ True \}\!\}$   
*<proof>*

**lemma (in pedal-coast-clique)**

**shows**  $(\Gamma \vdash_t \{\!\{ True \}\!\})\ PROC\ pedal('N, 'M)\ \{\!\{ True \}\!\} \wedge$   
     $(\Gamma \vdash_t \{\!\{ True \}\!\})\ PROC\ coast('N, 'M)\ \{\!\{ True \}\!\}$   
*<proof>*

**lemma (in pedal-coast-clique)**

**shows**  $(\Gamma \vdash_t \{\!\{ True \}\!\})\ PROC\ pedal('N, 'M)\ \{\!\{ True \}\!\} \wedge$   
     $(\Gamma \vdash_t \{\!\{ True \}\!\})\ PROC\ coast('N, 'M)\ \{\!\{ True \}\!\}$   
*<proof>*

**lemma (in pedal-coast-clique)**

**shows**  $(\Gamma \vdash_t \{\!\{ True \}\!\})\ PROC\ pedal('N, 'M)\ \{\!\{ True \}\!\} \wedge$   
     $(\Gamma \vdash_t \{\!\{ True \}\!\})\ PROC\ coast('N, 'M)\ \{\!\{ True \}\!\}$   
*<proof>*

**lemma (in pedal-coast-clique)**

**shows**  $(\Gamma \vdash_t \{\!\{ True \}\!\})\ PROC\ pedal('N, 'M)\ \{\!\{ True \}\!\} \wedge$   
     $(\Gamma \vdash_t \{\!\{ True \}\!\})\ PROC\ coast('N, 'M)\ \{\!\{ True \}\!\}$   
*<proof>*

end

## 20 Example: Quicksort on Heap Lists

**theory** Quicksort

**imports** ../Vcg ../HeapList HOL-Library.Multiset

**begin**

**record** *globals-heap* =

*next*' :: *ref*  $\Rightarrow$  *ref*

*cont*' :: *ref*  $\Rightarrow$  *nat*

**record** *'g vars* = *'g state* +

*p*' :: *ref*

*q*' :: *ref*

*le*' :: *ref*

*gt*' :: *ref*

*hd*' :: *ref*

*tl*' :: *ref*

**procedures**

*append*(*p*,*q*|*p*) =

IF *'p*=Null THEN *'p* ::= *'q* ELSE *'p* $\rightarrow$ *'next* ::= CALL *append*(*'p* $\rightarrow$ *'next*,*'q*)

FI

*append-spec*:

$\forall \sigma$  *Ps Qs*.

$\Gamma \vdash \{\sigma. \text{List } 'p \text{ 'next } Ps \wedge \text{List } 'q \text{ 'next } Qs \wedge \text{set } Ps \cap \text{set } Qs = \{\}\}$

$'p ::= \text{PROC } \text{append}('p, 'q)$

$\{\text{List } 'p \text{ 'next } (Ps @ Qs) \wedge (\forall x. x \notin \text{set } Ps \longrightarrow 'next \ x = {}^\sigma \text{next } x)\}$

*append-modifies*:

$\forall \sigma. \Gamma \vdash \{\sigma\} 'p ::= \text{PROC } \text{append}('p, 'q) \{t. t \text{ may-only-modify-globals } \sigma \text{ in } [next]\}$

**lemma** (in *append-impl*) *append-modifies*:

**shows**

$\forall \sigma. \Gamma \vdash \{\sigma\} 'p ::= \text{PROC } \text{append}('p, 'q) \{t. t \text{ may-only-modify-globals } \sigma \text{ in } [next]\}$

*<proof>*

**lemma** (in *append-impl*) *append-spec*:

**shows**  $\forall \sigma$  *Ps Qs*.  $\Gamma \vdash$

$\{\sigma. \text{List } 'p \text{ 'next } Ps \wedge \text{List } 'q \text{ 'next } Qs \wedge \text{set } Ps \cap \text{set } Qs = \{\}\}$

$'p ::= \text{PROC } \text{append}('p, 'q)$

$\{\text{List } 'p \text{ 'next } (Ps @ Qs) \wedge (\forall x. x \notin \text{set } Ps \longrightarrow 'next \ x = {}^\sigma \text{next } x)\}$

*<proof>*

**primrec** *sorted*:: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  bool

**where**

*sorted le* [] = True |

*sorted le* (x#xs) = (( $\forall y \in \text{set } xs. \text{le } x y$ )  $\wedge$  *sorted le* xs)

**lemma** *sorted-append[simp]*:

*sorted le* (xs@ys) = (*sorted le* xs  $\wedge$  *sorted le* ys  $\wedge$   
 $(\forall x \in \text{set } xs. \forall y \in \text{set } ys. \text{le } x y)$ )

*<proof>*

**procedures** *quickSort*(p|p) =

IF 'p=Null THEN SKIP

ELSE 'tl := 'p $\rightarrow$ 'next;;

'le := Null;;

'gt := Null;;

WHILE 'tl $\neq$ Null DO

'hd := 'tl;;

'tl := 'tl $\rightarrow$ 'next;;

IF 'hd $\rightarrow$ 'cont  $\leq$  'p $\rightarrow$ 'cont

THEN 'hd $\rightarrow$ 'next := 'le;;

'le := 'hd

ELSE 'hd $\rightarrow$ 'next := 'gt;;

'gt := 'hd

FI

OD;;

'le := CALL *quickSort*('le);;

'gt := CALL *quickSort*('gt);;

'p $\rightarrow$ 'next := 'gt;;

'le := CALL *append*('le, 'p);;

'p := 'le

FI

*quickSort-spec*:

$\forall \sigma \text{ Ps}. \Gamma \vdash \{\sigma. \text{List } 'p \text{ 'next Ps}\} 'p ::= \text{PROC } \text{quickSort}('p)$

$\{\{(\exists \text{sortedPs}. \text{List } 'p \text{ 'next sortedPs} \wedge$

*sorted* ( $\leq$ ) (*map*  $\sigma_{\text{cont}}$  *sortedPs*)  $\wedge$

*mset Ps* = *mset sortedPs*)  $\wedge$

$(\forall x. x \notin \text{set } Ps \longrightarrow 'next x = \sigma_{\text{next}} x)\}$

*quickSort-modifies*:

$\forall \sigma. \Gamma \vdash \{\sigma\} 'p ::= \text{PROC } \text{quickSort}('p) \{t. t \text{ may-only-modify-globals } \sigma \text{ in } [next]\}$

**lemma** (in *quickSort-impl*) *quickSort-modifies*:

**shows**

$\forall \sigma. \Gamma \vdash \{\sigma\} 'p ::= \text{PROC } \text{quickSort}('p) \{t. t \text{ may-only-modify-globals } \sigma \text{ in } [next]\}$

*<proof>*

**lemma** (in *quickSort-impl*) *quickSort-spec*:  
**shows**  
 $\forall \sigma Ps. \Gamma \vdash \{ \sigma. List \ 'p \ 'next Ps \}$   
 $\ 'p ::= PROC \ quickSort(\ 'p)$   
 $\{ (\exists \ sortedPs. List \ 'p \ 'next \ sortedPs \wedge$   
 $\ sorted \ (\le) \ (map \ \sigma_{cont} \ sortedPs) \wedge$   
 $\ mset \ Ps = mset \ sortedPs) \wedge$   
 $\ (\forall x. x \notin set \ Ps \longrightarrow \ 'next \ x = \sigma_{next} \ x) \}$   
 $\langle proof \rangle$

**end**

**theory** *XVcg*  
**imports** *Vcg*

**begin**

We introduce a syntactic variant of the let-expression so that we can safely unfold it during verification condition generation. With the new theorem attribute *vcg-simp* we can declare equalities to be used by the verification condition generator, while simplifying assertions.

**syntax**  
 $-Let' :: [letbinds, basicblock] \Rightarrow basicblock$   
 $\langle \langle notation = \langle mixfix \ LET \ expression \rangle \rangle LET \ (-) / IN \ (-) \rangle 23$

**syntax-consts**  
 $-Let' == Let'$

**translations**  
 $-Let' \ (-binds \ b \ bs) \ e == -Let' \ b \ (-Let' \ bs \ e)$   
 $-Let' \ (-bind \ x \ a) \ e == CONST \ Let' \ a \ (\%x. \ e)$

**lemma** *Let'-unfold* [*vcg-simp*]:  $Let' \ x \ f = f \ x$   
 $\langle proof \rangle$

**lemma** *Let'-split-conv* [*vcg-simp*]:  
 $(Let' \ x \ (\lambda p. \ (case\ prod \ (f \ p) \ (g \ p)))) =$   
 $(Let' \ x \ (\lambda p. \ (f \ p) \ (fst \ (g \ p)) \ (snd \ (g \ p))))$   
 $\langle proof \rangle$

**end**

## 21 Examples for Parallel Assignments

**theory** *XVcgEx*  
**imports**  $../XVcg$

```

begin

record globals =
  G'::nat
  H'::nat

record 'g vars = 'g state +
  A'::nat
  B'::nat
  C'::nat
  I'::nat
  M'::nat
  N'::nat
  R'::nat
  S'::nat
  Arr'::nat list
  Abr'::string

term BASIC
  'A ::= x,
  'B ::= y
  END

term BASIC
  'G ::= 'H,
  'H ::= 'G
  END

term BASIC
  LET (x,y) = ('A,b);
  z = 'B
  IN 'A ::= x,
  'G ::= 'A + y + z
  END

lemma  $\Gamma \vdash \{ 'A = 0 \}$ 
   $\{ 'A < 0 \} \mapsto \text{BASIC}$ 
  LET (a,b,c) = foo 'A
  IN
  'A ::= a,
  'B ::= b,
  'C ::= c
  END
   $\{ 'A = x \wedge 'B = y \wedge 'C = c \}$ 
  <proof>

lemma  $\Gamma \vdash \{ 'A = 0 \}$ 

```

$\{\ 'A < 0 \} \mapsto \text{BASIC}$   
 $\text{LET } (a,b,c) = \text{foo } 'A$   
 $\text{IN}$   
 $\ 'A ::= a,$   
 $\ 'G ::= b + 'B,$   
 $\ 'H ::= c$   
 $\text{END}$   
 $\{\ 'A = x \wedge 'G = y \wedge 'H = c \}$   
 $\langle \text{proof} \rangle$

**definition**  $\text{foo} :: \text{nat} \Rightarrow (\text{nat} \times \text{nat} \times \text{nat})$   
**where**  $\text{foo } n = (n, n+1, n+2)$

**lemma**  $\Gamma \vdash \{\ 'A = 0 \}$   
 $\{\ 'A < 0 \} \mapsto \text{BASIC}$   
 $\text{LET } (a,b,c) = \text{foo } 'A$   
 $\text{IN}$   
 $\ 'A ::= a,$   
 $\ 'G ::= b + 'B,$   
 $\ 'H ::= c$   
 $\text{END}$   
 $\{\ 'A = x \wedge 'G = y \wedge 'H = c \}$   
 $\langle \text{proof} \rangle$

**end**

## 22 Examples for Procedures as Parameters

**theory** *ProcParEx* **imports** *../Vcg* **begin**

**lemma** *DynProcProcPar'*:  
**assumes** *adapt*:  $P \subseteq \{s. p \ s = q \wedge$   
 $(\exists Z. \text{init } s \in P' \ Z \wedge$   
 $(\forall t \in Q' \ Z. \text{return } s \ t \in R \ s \ t) \wedge$   
 $(\forall t \in A' \ Z. \text{return } s \ t \in A))\}$   
**assumes** *result*:  $\forall s \ t. \Gamma, \Theta \vdash_{/F} (R \ s \ t) \ \text{result } s \ t \ Q, A$   
**assumes** *q*:  $\forall Z. \Gamma, \Theta \vdash_{/F} (P' \ Z) \ \text{Call } q \ (Q' \ Z), (A' \ Z)$   
**shows**  $\Gamma, \Theta \vdash_{/F} P \ \text{dynCall } \text{init } p \ \text{return } \text{result } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *conseq-exploit-pre'*:  
 $\llbracket \forall s \in S. \Gamma, \Theta \vdash (\{s\} \cap P) \ c \ Q, A \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash (P \cap S) \ c \ Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *conseq-exploit-pre''*:  

$$\llbracket \forall Z. \forall s \in S Z. \Gamma, \Theta \vdash (\{s\} \cap P Z) c (Q Z), (A Z) \rrbracket$$

$$\implies$$

$$\forall Z. \Gamma, \Theta \vdash (P Z \cap S Z) c (Q Z), (A Z)$$
*<proof>*

**lemma** *conseq-exploit-pre'''*:  

$$\llbracket \forall s \in S. \forall Z. \Gamma, \Theta \vdash (\{s\} \cap P Z) c (Q Z), (A Z) \rrbracket$$

$$\implies$$

$$\forall Z. \Gamma, \Theta \vdash (P Z \cap S) c (Q Z), (A Z)$$
*<proof>*

**record** *'g vars = 'g state +*  
*compare-' :: string*  
*n-' :: nat*  
*m-' :: nat*  
*b-' :: bool*  
*k-' :: nat*

**procedures** *compare(n,m|b) = NoBody*  
**print-locale!** *compare-signature*

**context** *compare-signature*  
**begin**  
**declare** *[[hoare-use-call-tr' = false]]*  
**term** *'b ::= CALL compare('n, 'm)*  
**term** *'b ::= DYNCALL 'compare('n, 'm)*  
**declare** *[[hoare-use-call-tr' = true]]*  
**term** *'b ::= DYNCALL 'compare('n, 'm)*  
**end**

**procedures**  
*LEQ (n,m | b) = 'b ::= 'n ≤ 'm*  
*LEQ-spec:  $\forall \sigma. \Gamma \vdash \{\sigma\} \text{ PROC } LEQ('n, 'm, 'b) \{ \{ 'b = (\sigma_n \leq \sigma_m) \} \}$*   
*LEQ-modifies:  $\forall \sigma. \Gamma \vdash \{\sigma\} \text{ PROC } LEQ('n, 'm, 'b) \{ t. t \text{ may-only-modify-globals } \sigma \}$*   
*in []}*

**definition** *mx:: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a  $\Rightarrow$  'a  $\Rightarrow$  'a*  
**where** *mx leq a b = (if leq a b then a else b)*

**procedures**

$$\begin{aligned} &Max (compare, n, m \mid k) = \\ &'b ::= DYNCALL 'compare('n, 'm);; \\ &IF 'b THEN 'k ::= 'n ELSE 'k ::= 'm FI \end{aligned}$$

$$\begin{aligned} &Max-spec: \bigwedge leq. \forall \sigma. \Gamma \vdash \\ &(\{\sigma\} \cap \{s. (\forall \tau. \Gamma \vdash \{\tau\} 'b ::= PROC {}^scompare('n, 'm) \{\!\!| 'b = (leq {}^{\tau}n {}^{\tau}m) \!\!\}) \wedge \\ &(\forall \tau. \Gamma \vdash \{\tau\} 'b ::= PROC {}^scompare('n, 'm) \{t. t \text{ may-only-modify-globals } \tau \\ &\tau \text{ in } []\})\}) \\ &PROC Max('compare, 'n, 'm, 'k) \\ &\{\!\!| 'k = mx leq {}^{\sigma}n {}^{\sigma}m \!\!\} \end{aligned}$$
**lemma (in Max-impl ) Max-spec1:****shows**

$$\begin{aligned} &\forall \sigma leq. \Gamma \vdash \\ &(\{\sigma\} \cap \{\!\!| (\forall \tau. \Gamma \vdash \{\tau\} 'b ::= PROC {}'compare('n, 'm) \{\!\!| 'b = (leq {}^{\tau}n {}^{\tau}m) \!\!\}) \wedge \\ &(\forall \tau. \Gamma \vdash \{\tau\} 'b ::= PROC {}'compare('n, 'm) \{t. t \text{ may-only-modify-globals } \tau \\ &\text{in } []\})\!\!\}) \\ &'k ::= PROC Max('compare, 'n, 'm) \\ &\{\!\!| 'k = mx leq {}^{\sigma}n {}^{\sigma}m \!\!\} \\ &\langle proof \rangle \end{aligned}$$
**lemma (in Max-impl ) Max-spec2:****shows**

$$\begin{aligned} &\forall \sigma leq. \Gamma \vdash \\ &(\{\sigma\} \cap \{\!\!| (\forall \tau. \Gamma \vdash \{\tau\} 'b ::= PROC {}'compare('n, 'm) \{\!\!| 'b = (leq {}^{\tau}n {}^{\tau}m) \!\!\}) \wedge \\ &(\forall \tau. \Gamma \vdash \{\tau\} 'b ::= PROC {}'compare('n, 'm) \{t. t \text{ may-only-modify-globals } \tau \\ &\text{in } []\})\!\!\}) \\ &'k ::= PROC Max('compare, 'n, 'm) \\ &\{\!\!| 'k = mx leq {}^{\sigma}n {}^{\sigma}m \!\!\} \\ &\langle proof \rangle \end{aligned}$$
**lemma (in Max-impl ) Max-spec3:****shows**

$$\begin{aligned} &\forall n m leq. \Gamma \vdash \\ &(\{\!\!| 'n=n \wedge 'm=m \!\!\} \cap \\ &\{\!\!| (\forall \tau. \Gamma \vdash \{\tau\} 'b ::= PROC {}'compare('n, 'm) \{\!\!| 'b = (leq {}^{\tau}n {}^{\tau}m) \!\!\}) \wedge \\ &(\forall \tau. \Gamma \vdash \{\tau\} 'b ::= PROC {}'compare('n, 'm) \{t. t \text{ may-only-modify-globals } \tau \text{ in } \\ &[]\})\!\!\}) \\ &'k ::= PROC Max('compare, 'n, 'm) \\ &\{\!\!| 'k = mx leq n m \!\!\} \\ &\langle proof \rangle \end{aligned}$$
**lemma (in Max-impl ) Max-spec4:****shows**

$$\begin{aligned} &\forall n m leq. \Gamma \vdash \\ &(\{\!\!| 'n=n \wedge 'm=m \!\!\} \cap \{\!\!| \forall \tau. \Gamma \vdash \{\tau\} 'b ::= PROC {}'compare('n, 'm) \{\!\!| 'b = (leq {}^{\tau}n \end{aligned}$$

$\tau m\} \} \}$   
 $k := \text{PROC Max}(\text{'compare}, 'n, 'm)$   
 $\{k = \text{mx leq } n \ m\}$   
 $\langle \text{proof} \rangle$

**locale** *Max-test* = *Max-spec* + *LEQ-spec* + *LEQ-modifies*  
**lemma** (in *Max-test*)

**shows**  
 $\Gamma \vdash \{\sigma\} \ k := \text{CALL Max}(\text{LEQ-'proc}, 'n, 'm) \ \{k = \text{mx } (\leq) \ \sigma_n \ \sigma_m\}$   
 $\langle \text{proof} \rangle$

**lemma** (in *Max-impl*) *Max-spec5*:

**shows**  
 $\forall n \ m \ \text{leq}. \ \Gamma \vdash$   
 $(\{n=n \wedge m=m\} \cap \{\forall n' \ m'. \ \Gamma \vdash \{n=n' \wedge m=m'\} \ b := \text{PROC 'compare}('n, 'm) \ \{b = (\text{leq } n' \ m')\}\})$   
 $k := \text{PROC Max}(\text{'compare}, 'n, 'm)$   
 $\{k = \text{mx leq } n \ m\}$   
**term**  $\{\{s. \ s_n = n' \wedge s_m = m'\} = X\}$   
 $\langle \text{proof} \rangle$

**lemma** (in *LEQ-impl*)

*LEQ-spec*:  $\forall n \ m. \ \Gamma \vdash \{n=n \wedge m=m\} \ \text{PROC LEQ}('n, 'm, 'b) \ \{b = (n \leq m)\}$   
 $\langle \text{proof} \rangle$

**locale** *Max-test'* = *Max-impl* + *LEQ-impl*

**lemma** (in *Max-test'*)

**shows**  
 $\forall n \ m. \ \Gamma \vdash \{n=n \wedge m=m\} \ k := \text{CALL Max}(\text{LEQ-'proc}, 'n, 'm) \ \{k = \text{mx } (\leq) \ n \ m\}$   
 $\langle \text{proof} \rangle$

**end**

## 23 Examples for Procedures as Parameters using Statespaces

**theory** *ProcParExSP* **imports** *../Vcg* **begin**

**lemma** *DynProcProcPar'*:

**assumes** *adapt*:  $P \subseteq \{s. \ p \ s = q \wedge$   
 $(\exists Z. \ \text{init } s \in P' \ Z \wedge$   
 $(\forall t \in Q' \ Z. \ \text{return } s \ t \in R \ s \ t) \wedge$   
 $(\forall t \in A' \ Z. \ \text{return } s \ t \in A))\}$

**assumes** *result*:  $\forall s t. \Gamma, \Theta \vdash_F (R s t) \text{ result } s t Q, A$   
**assumes** *q*:  $\forall Z. \Gamma, \Theta \vdash_F (P' Z) \text{ Call } q (Q' Z), (A' Z)$   
**shows**  $\Gamma, \Theta \vdash_F P \text{ dynCall init } p \text{ return result } Q, A$   
 $\langle \text{proof} \rangle$

**lemma** *conseq-exploit-pre'*:  

$$\llbracket \forall s \in S. \Gamma, \Theta \vdash (\{s\} \cap P) c Q, A \rrbracket$$

$$\implies$$

$$\Gamma, \Theta \vdash (P \cap S) c Q, A$$
 $\langle \text{proof} \rangle$

**lemma** *conseq-exploit-pre''*:  

$$\llbracket \forall Z. \forall s \in S Z. \Gamma, \Theta \vdash (\{s\} \cap P Z) c (Q Z), (A Z) \rrbracket$$

$$\implies$$

$$\forall Z. \Gamma, \Theta \vdash (P Z \cap S Z) c (Q Z), (A Z)$$
 $\langle \text{proof} \rangle$

**lemma** *conseq-exploit-pre'''*:  

$$\llbracket \forall s \in S. \forall Z. \Gamma, \Theta \vdash (\{s\} \cap P Z) c (Q Z), (A Z) \rrbracket$$

$$\implies$$

$$\forall Z. \Gamma, \Theta \vdash (P Z \cap S) c (Q Z), (A Z)$$
 $\langle \text{proof} \rangle$

**procedures** *compare*(*i*::nat,*j*::nat|*r*::bool) *NoBody*

**print-locale!** *compare-signature*

**context** *compare-impl*  
**begin**  
**declare**  $\llbracket \text{hoare-use-call-tr}' = \text{false} \rrbracket$   
**term** *'r* ::= *CALL compare*(*'i*, *'j*)  
**declare**  $\llbracket \text{hoare-use-call-tr}' = \text{true} \rrbracket$   
**end**

**procedures**  
*LEQ* (*i*::nat,*j*::nat | *r*::bool) *'r* ::= *'i* ≤ *'j*  
*LEQ-spec*:  $\forall \sigma. \Gamma \vdash \{\sigma\} \text{ PROC } \text{LEQ}(\text{'i}, \text{'j}, \text{'r}) \{ \text{'r} = (\sigma_i \leq \sigma_j) \}$   
  
*LEQ-modifies*:  $\forall \sigma. \Gamma \vdash \{\sigma\} \text{ PROC } \text{LEQ}(\text{'i}, \text{'j}, \text{'r}) \{ t. t \text{ may-only-modify-globals } \sigma \text{ in } [] \}$

**definition**  $mx:: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$   
**where**  $mx \text{ leq } a \ b = (\text{if leq } a \ b \text{ then } a \ \text{else } b)$

**procedures** (**imports** *compare-signature*)  
*Max* (*compare::string*, *n::nat*, *m::nat* | *k::nat*)  
**where**  $b::bool$   
**in**  
 $'b ::= \text{DYNCALL } 'compare('n, 'm);;$   
 $\text{IF } 'b \ \text{THEN } 'k ::= 'n \ \text{ELSE } 'k ::= 'm \ \text{FI}$

*Max-spec*:  $\bigwedge \text{leq}. \forall \sigma. \Gamma \vdash$   
 $(\{\sigma\} \cap \{s. (\forall \tau. \Gamma \vdash \{\tau\} \ 'r ::= \text{PROC } ^scompare('i, 'j) \ \{\!| r = (\text{leq } ^{\tau}i \ ^{\tau}j)\!|\}) \wedge$   
 $(\forall \tau. \Gamma \vdash \{\tau\} \ 'r ::= \text{PROC } ^scompare('i, 'j) \ \{t. t \ \text{may-only-modify-globals}$   
 $\tau \ \text{in } []\})\})$   
 $\text{PROC } Max('compare, 'n, 'm, 'k)$   
 $\{\!| k = mx \ \text{leq } \sigma_n \ \sigma_m\}$

**context** *Max-spec*  
**begin**  
**thm** *Max-spec*  
**end**  
**context** *Max-impl*  
**begin**  
**term**  $'b ::= \text{DYNCALL } 'compare('n, 'm)$   
**declare**  $[[\text{hoare-use-call-tr}' = \text{false}]]$   
**term**  $'b ::= \text{DYNCALL } 'compare('n, 'm)$   
**declare**  $[[\text{hoare-use-call-tr}' = \text{true}]]$   
**end**

**lemma** (**in** *Max-impl*) *Max-spec1*:

**shows**  
 $\forall \sigma \ \text{leq}. \Gamma \vdash$   
 $(\{\sigma\} \cap \{\!| (\forall \tau. \Gamma \vdash \{\tau\} \ 'r ::= \text{PROC } 'compare('i, 'j) \ \{\!| r = (\text{leq } ^{\tau}i \ ^{\tau}j)\!|\}) \wedge$   
 $(\forall \tau. \Gamma \vdash \{\tau\} \ 'r ::= \text{PROC } 'compare('i, 'j) \ \{t. t \ \text{may-only-modify-globals } \tau \ \text{in}$   
 $[]\})\!|\})$   
 $'k ::= \text{PROC } Max('compare, 'n, 'm)$   
 $\{\!| k = mx \ \text{leq } \sigma_n \ \sigma_m\}$   
 $\langle \text{proof} \rangle$

**lemma** (**in** *Max-impl*) *Max-spec2*:

**shows**  
 $\forall \sigma \ \text{leq}. \Gamma \vdash$   
 $(\{\sigma\} \cap \{\!| (\forall \tau. \Gamma \vdash \{\tau\} \ 'r ::= \text{PROC } 'compare('i, 'j) \ \{\!| r = (\text{leq } ^{\tau}i \ ^{\tau}j)\!|\}) \wedge$   
 $(\forall \tau. \Gamma \vdash \{\tau\} \ 'r ::= \text{PROC } 'compare('i, 'j) \ \{t. t \ \text{may-only-modify-globals } \tau \ \text{in}$

$\{\!\{ \}\!\}$ )  
 $k := \text{PROC Max}(\text{'compare}, 'n, 'm)$   
 $\{\!\{ k = \text{mx leq } \sigma_n \sigma_m \}\!\}$   
 $\langle \text{proof} \rangle$

**lemma** (in *Max-impl*) *Max-spec3*:

**shows**

$\forall n m \text{ leq}. \Gamma \vdash$   
 $(\{\!\{ 'n=n \wedge 'm=m \}\!\} \cap$   
 $\{\!\{ (\forall \tau. \Gamma \vdash \{\tau\} 'r := \text{PROC 'compare}(i, j) \{\!\{ r = (\text{leq } \tau_i \tau_j) \}\!\}) \wedge$   
 $(\forall \tau. \Gamma \vdash \{\tau\} 'r := \text{PROC 'compare}(i, j) \{t. t \text{ may-only-modify-globals } \tau \text{ in}$   
 $\{\!\{ \}\!\}) \}\!\})$   
 $k := \text{PROC Max}(\text{'compare}, 'n, 'm)$   
 $\{\!\{ k = \text{mx leq } n m \}\!\}$   
 $\langle \text{proof} \rangle$

**lemma** (in *Max-impl*) *Max-spec4*:

**shows**

$\forall n m \text{ leq}. \Gamma \vdash$   
 $(\{\!\{ 'n=n \wedge 'm=m \}\!\} \cap \{\!\{ \forall \tau. \Gamma \vdash \{\tau\} 'r := \text{PROC 'compare}(i, j) \{\!\{ r = (\text{leq } \tau_i \tau_j) \}\!\} \}\!\})$   
 $k := \text{PROC Max}(\text{'compare}, 'n, 'm)$   
 $\{\!\{ k = \text{mx leq } n m \}\!\}$   
 $\langle \text{proof} \rangle$

**print-locale** *Max-spec*

**locale** *Max-test* = *Max-spec* **where**

$i\text{'compare}' = i\text{'LEQ}'$  **and**  
 $j\text{'compare}' = j\text{'LEQ}'$  **and**  
 $r\text{'compare}' = r\text{'LEQ}'$   
 $+ \text{LEQ-spec} + \text{LEQ-modifies}$

**lemma** (in *Max-test*)

**shows**

$\Gamma \vdash \{\sigma\} k := \text{CALL Max}(\text{LEQ}'\text{proc}, 'n, 'm) \{\!\{ k = \text{mx } (\leq) \sigma_n \sigma_m \}\!\}$   
 $\langle \text{proof} \rangle$

**lemma** (in *Max-impl*) *Max-spec5*:

**shows**

$\forall n m \text{ leq}. \Gamma \vdash$   
 $(\{\!\{ 'n=n \wedge 'm=m \}\!\} \cap \{\!\{ \forall n' m'. \Gamma \vdash \{\!\{ i=n' \wedge j=m' \}\!\} 'r := \text{PROC 'compare}(i, 'j) \{\!\{ r = (\text{leq } n' m') \}\!\} \}\!\})$

$k := PROC\ Max('compare, 'n, 'm)$   
 $\{\{k = mx\ leq\ n\ m\}\}$   
 $\langle proof \rangle$

**lemma** (in *LEQ-impl*)  
*LEQ-spec*:  $\forall n\ m. \Gamma \vdash \{\{i=n \wedge 'j=m\}\} PROC\ LEQ('i, 'j, 'r) \{\{r = (n \leq m)\}\}$   
 $\langle proof \rangle$

**print-locale** *Max-impl*  
**locale** *Max-test'* = *Max-impl* **where**  
 $i\text{'compare-}' = i\text{'LEQ-}'$  **and**  
 $j\text{'compare-}' = j\text{'LEQ-}'$  **and**  
 $r\text{'compare-}' = r\text{'LEQ-}'$   
 $+ LEQ\text{-impl}$

**lemma** (in *Max-test'*)  
**shows**  
 $\forall n\ m. \Gamma \vdash \{\{n=n \wedge 'm=m\}\} k := CALL\ Max(LEQ\text{'proc}, 'n, 'm) \{\{k = mx (\leq) n\ m\}\}$   
 $\langle proof \rangle$

**end**

## 24 Experiments with Closures

**theory** *Closure*  
**imports** *../Hoare*  
**begin**

**definition**  
 $callClosure\ upd\ cl = Seq\ (Basic\ (upd\ (fst\ cl)))\ (Call\ (snd\ cl))$

**definition**  
 $dynCallClosure\ init\ upd\ cl\ return\ c =$   
 $DynCom\ (\lambda s. call\ (upd\ (fst\ (cl\ s))\ \circ\ init)\ (snd\ (cl\ s))\ return\ c)$

**lemma** *dynCallClosure-sound*:  
**assumes** *adapt*:  
 $P \subseteq \{s. \exists P' Q' A'. \forall n. \Gamma, \Theta \models n: /_F P' (callClosure\ upd\ (cl\ s)) Q', A' \wedge$   
 $init\ s \in P' \wedge$   
 $(\forall t \in Q'. return\ s\ t \in R\ s\ t) \wedge$   
 $(\forall t \in A'. return\ s\ t \in A)\}$   
**assumes** *res*:  $\forall s\ t\ n. \Gamma, \Theta \models n: /_F (R\ s\ t) (c\ s\ t) Q, A$

**shows**

$\Gamma, \Theta \models_n: /_F P$  (*dynCallClosure* *init upd cl return c*)  $Q, A$   
*<proof>*

**lemma** *dynCallClosure*:

**assumes** *adapt*:  $P \subseteq \{s. \exists P' Q' A'. \Gamma, \Theta \vdash /_F P' (\text{callClosure } \text{upd } (cl\ s)) Q', A' \wedge$   
 $\text{init } s \in P' \wedge$   
 $(\forall t \in Q'. \text{return } s\ t \in R\ s\ t) \wedge$   
 $(\forall t \in A'. \text{return } s\ t \in A)\}$

**assumes** *res*:  $\forall s\ t. \Gamma, \Theta \vdash /_F (R\ s\ t) (c\ s\ t) Q, A$

**shows**

$\Gamma, \Theta \vdash /_F P$  (*dynCallClosure* *init upd cl return c*)  $Q, A$   
*<proof>*

**lemma** *in-subsetD*:  $\llbracket P \subseteq P'; x \in P \rrbracket \implies x \in P'$

*<proof>*

**lemma** *dynCallClosureFix*:

**assumes** *adapt*:  $P \subseteq \{s. \exists Z. cl' = cl\ s \wedge$   
 $\text{init } s \in P' Z \wedge$   
 $(\forall t \in Q' Z. \text{return } s\ t \in R\ s\ t) \wedge$   
 $(\forall t \in A' Z. \text{return } s\ t \in A)\}$

**assumes** *res*:  $\forall s\ t. \Gamma, \Theta \vdash /_F (R\ s\ t) (c\ s\ t) Q, A$

**assumes** *spec*:  $\forall Z. \Gamma, \Theta \vdash /_F (P' Z) (\text{callClosure } \text{upd } cl') (Q' Z), (A' Z)$

**shows**

$\Gamma, \Theta \vdash /_F P$  (*dynCallClosure* *init upd cl return c*)  $Q, A$   
*<proof>*

**lemma** *conseq-extract-pre*:

$\llbracket \forall s \in P. \Gamma, \Theta \vdash /_F (\{s\})\ c\ Q, A \rrbracket$   
 $\implies$   
 $\Gamma, \Theta \vdash /_F P\ c\ Q, A$

*<proof>*

**lemma** *app-closure-sound*:

**assumes** *adapt*:  $P \subseteq \{s. \exists P' Q' A'. \forall n. \Gamma, \Theta \models_n: /_F P' (\text{callClosure } \text{upd } (e', p))$   
 $Q', A' \wedge$

$\text{upd } x\ s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A\}$

**assumes** *ap*:  $\text{upd } e = \text{upd } e' \circ \text{upd } x$

**shows**  $\Gamma, \Theta \models_n: /_F P$  (*callClosure* *upd*  $(e, p)$ )  $Q, A$

*<proof>*

**lemma** *app-closure*:

**assumes** *adapt*:  $P \subseteq \{s. \exists P' Q' A'. \Gamma, \Theta \vdash /_F P' (\text{callClosure } \text{upd } (e', p)) Q', A' \wedge$

$upd\ x\ s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A$

**assumes**  $ap: upd\ e = upd\ e' \circ upd\ x$   
**shows**  $\Gamma, \Theta \vdash_F P\ (callClosure\ upd\ (e, p))\ Q, A$   
 $\langle proof \rangle$

**lemma** *app-closure-spec*:

**assumes**  $adapt: P \subseteq \{s. \exists Z. upd\ x\ s \in P' Z \wedge Q' Z \subseteq Q \wedge A' Z \subseteq A\}$   
**assumes**  $ap: upd\ e = upd\ e' \circ upd\ x$   
**assumes**  $spec: \forall Z. \Gamma, \Theta \vdash_F (P' Z)\ (callClosure\ upd\ (e', p))\ (Q' Z), (A' Z)$   
**shows**  $\Gamma, \Theta \vdash_F P\ (callClosure\ upd\ (e, p))\ Q, A$   
 $\langle proof \rangle$

Implementation of closures as association lists.

**definition** *gen-upd var es s = foldl* ( $\lambda s\ (x, i). the\ (var\ x)\ i\ s$ )  $s\ es$

**definition**  $ap\ es\ c \equiv (es @ fst\ c, snd\ c)$

**lemma** *gen-upd-app*:  $\bigwedge es'. gen-upd\ var\ (es @ es') = gen-upd\ var\ es' \circ gen-upd\ var\ es$   
 $\langle proof \rangle$

**lemma** *gen-upd-ap*:

$gen-upd\ var\ (fst\ (ap\ es\ (es', p))) = gen-upd\ var\ es' \circ gen-upd\ var\ es$   
 $\langle proof \rangle$

**lemma** *ap-closure*:

**assumes**  $adapt: P \subseteq \{s. \exists P' Q' A'. \Gamma, \Theta \vdash_F P'\ (callClosure\ (gen-upd\ var)\ c)\ Q', A' \wedge$   
 $gen-upd\ var\ es\ s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A\}$   
**shows**  $\Gamma, \Theta \vdash_F P\ (callClosure\ (gen-upd\ var)\ (ap\ es\ c))\ Q, A$   
 $\langle proof \rangle$

**lemma** *ap-closure-spec*:

**assumes**  $adapt: P \subseteq \{s. \exists Z. gen-upd\ var\ es\ s \in P' Z \wedge Q' Z \subseteq Q \wedge A' Z \subseteq A\}$   
**assumes**  $spec: \forall Z. \Gamma, \Theta \vdash_F (P' Z)\ (callClosure\ (gen-upd\ var)\ c)\ (Q' Z), (A' Z)$   
**shows**  $\Gamma, \Theta \vdash_F P\ (callClosure\ (gen-upd\ var)\ (ap\ es\ c))\ Q, A$   
 $\langle proof \rangle$

**end**

**theory** *ClosureEx*

**imports**  $../Vcg\ ../Simpl-Heap\ Closure$

**begin**

**record** *globals* =

$cnt-' :: ref \Rightarrow nat$

$alloc-' :: ref\ list$   
 $free-' :: nat$   
**record**  $'g\ vars = 'g\ state +$   
 $p-' :: ref$   
 $r-' :: nat$   
 $n-' :: nat$   
 $m-' :: nat$   
 $c-' :: (string \times ref)\ list \times string$   
 $d-' :: (string \times ref)\ list \times string$   
 $e-' :: (string \times nat)\ list \times string$

**definition**  $var_n = [''n'' \mapsto (\lambda x. n\text{-}'\text{-update} (\lambda\cdot. x)),$   
 $''m'' \mapsto (\lambda x. m\text{-}'\text{-update} (\lambda\cdot. x))]$

**definition**  $upd_n = gen\text{-}upd\ var_n$

**lemma**  $upd_n\text{-}ap: upd_n (fst (ap\ es (es',p))) = upd_n\ es' \circ upd_n\ es$   
 $\langle proof \rangle$

**lemma**

$\Gamma \vdash \{ 'n = n_0 \wedge (\forall i\ j. \Gamma \vdash \{ 'n = i \wedge 'm = j \} callClosure\ upd_n\ 'e\ \{ 'r = i + j \}) \}$   
 $\quad 'e ::= (ap\ [(''n'', 'n)]\ 'e)$   
 $\quad \{ \forall j. \Gamma \vdash \{ 'm = j \} callClosure\ upd_n\ 'e\ \{ 'r = n_0 + j \} \}$   
 $\langle proof \rangle$

**definition**  $var = [''p'' \mapsto (\lambda x. p\text{-}'\text{-update} (\lambda\cdot. x))]$

**definition**  $upd = gen\text{-}upd\ var$

**procedures**  $Inc(p|r) =$

$\quad 'p \rightarrow 'cnt ::= 'p \rightarrow 'cnt + 1;;$   
 $\quad 'r ::= 'p \rightarrow 'cnt$

**lemma** (**in**  $Inc\text{-}impl$ )

$\forall i\ p. \Gamma \vdash \{ 'p \rightarrow 'cnt = i \} 'r ::= PROC\ Inc('p)\ \{ 'r = i + 1 \wedge 'p \rightarrow 'cnt = i + 1 \}$   
 $\langle proof \rangle$

**procedures** (**imports**  $Inc\text{-}signature$ )  $NewCounter(|c) =$

$\quad 'p ::= NEW\ 1\ [ 'cnt ::= 0];;$   
 $\quad 'c ::= ([(''p'', 'p)], Inc\text{-}'\text{-proc})$

**locale**  $NewCounter\text{-}impl' = NewCounter\text{-}impl + Inc\text{-}impl$

**lemma** (**in**  $NewCounter\text{-}impl'$ )

**shows**

$\forall alloc. \Gamma \vdash \{ 1 \leq 'free \} 'c ::= PROC\ NewCounter()$   
 $\quad \{ \exists p. p \rightarrow 'cnt = 0 \wedge$   
 $\quad (\forall i. \Gamma \vdash \{ p \rightarrow 'cnt = i \} callClosure\ upd\ 'c\ \{ 'r = i + 1 \wedge p \rightarrow 'cnt = i + 1 \}) \}$

$\langle proof \rangle$

**lemma** (in *NewCounter-impl'*)

**shows**

$$\begin{aligned} & \forall alloc. \Gamma \vdash \{1 \leq 'free\} 'c ::= PROC NewCounter() \\ & \{ \exists p. p \rightarrow 'cnt = 0 \wedge \\ & (\forall i. \Gamma \vdash \{p \rightarrow 'cnt = i\} callClosure upd 'c \{ 'r=i+1 \wedge p \rightarrow 'cnt = i+1 \}) \} \end{aligned}$$

$\langle proof \rangle$

**lemma** (in *NewCounter-impl'*)

**shows** *NewCounter-spec*:

$$\begin{aligned} & \forall alloc. \Gamma \vdash \{1 \leq 'free \wedge 'alloc=alloc\} 'c ::= PROC NewCounter() \\ & \{ \exists p. p \notin set alloc \wedge p \in set 'alloc \wedge p \neq Null \wedge p \rightarrow 'cnt = 0 \wedge \\ & (\forall i. \Gamma \vdash \{p \rightarrow 'cnt = i\} callClosure upd 'c \{ 'r=i+1 \wedge p \rightarrow 'cnt = i+1 \}) \} \end{aligned}$$

$\langle proof \rangle$

**lemma**  $\Gamma \vdash \{ \exists p. p \neq Null \wedge p \rightarrow 'cnt = i \wedge$

$$\begin{aligned} & (\forall i. \Gamma \vdash \{p \rightarrow 'cnt = i\} callClosure upd 'c \{ 'r=i+1 \wedge p \rightarrow 'cnt = i+1 \}) \} \\ & dynCallClosure (\lambda s. s) upd c-' (\lambda s t. s(\mathit{globals} := \mathit{globals} t)) \\ & (\lambda s t. Basic (\lambda u. u(\mathit{r}' := \mathit{r}' t))) \end{aligned}$$
$$\{ 'r=i+1 \}$$

$\langle proof \rangle$

**declare**  $[[hoare-trace = 1]]$

$\langle ML \rangle$

**lemma** (in *NewCounter-impl'*)

**shows**  $\Gamma \vdash \{1 \leq 'free\}$

$$\begin{aligned} & 'c ::= CALL NewCounter ();; \\ & dynCallClosure (\lambda s. s) upd c-' (\lambda s t. s(\mathit{globals} := \mathit{globals} t)) \\ & (\lambda s t. Basic (\lambda u. u(\mathit{r}' := \mathit{r}' t))) \end{aligned}$$
$$\{ 'r=1 \}$$

$\langle proof \rangle$

**lemma** (in *NewCounter-impl'*)

**shows**  $\Gamma \vdash \{1 \leq 'free\}$

$$\begin{aligned} & 'c ::= CALL NewCounter ();; \\ & dynCallClosure (\lambda s. s) upd c-' (\lambda s t. s(\mathit{globals} := \mathit{globals} t)) \\ & (\lambda s t. Basic (\lambda u. u(\mathit{r}' := \mathit{r}' t)));; \\ & dynCallClosure (\lambda s. s) upd c-' (\lambda s t. s(\mathit{globals} := \mathit{globals} t)) \\ & (\lambda s t. Basic (\lambda u. u(\mathit{r}' := \mathit{r}' t))) \end{aligned}$$
$$\{ 'r=2 \}$$

$\langle proof \rangle$

**lemma** (in *NewCounter-impl'*)  
**shows**  $\Gamma \vdash \{\!| 1 \leq 'free \|\}$   
 $'c ::= CALL\ NewCounter\ ();;$   
 $'d ::= 'c;;$   
 $dynCallClosure\ (\lambda s.\ s)\ upd\ c\text{-}'\ (\lambda s\ t.\ s(\!|globals := globals\ t\!|))$   
 $\quad (\lambda s\ t.\ Basic\ (\lambda u.\ u(\!|n\text{-}' := r\text{-}'\ t\!|)));;$   
 $dynCallClosure\ (\lambda s.\ s)\ upd\ d\text{-}'\ (\lambda s\ t.\ s(\!|globals := globals\ t\!|))$   
 $\quad (\lambda s\ t.\ Basic\ (\lambda u.\ u(\!|m\text{-}' := r\text{-}'\ t\!|)));;$   
 $'r ::= 'n + 'm$   
 $\{\!| r = 3 \|\}$

*<proof>*

**end**

## 25 Experiments on State Composition

**theory** *Compose* **imports** *../HoareTotalProps* **begin**

We develop some theory to support state-space modular development of programs. These experiments aim at the representation of state-spaces with records. If we use *statespaces* instead we get this kind of compositionality for free.

### 25.1 Changing the State-Space

**definition**  $lift_f:: ('S \Rightarrow 's) \Rightarrow ('S \Rightarrow 's \Rightarrow 'S) \Rightarrow ('s \Rightarrow 's) \Rightarrow ('S \Rightarrow 'S)$   
**where**  $lift_f\ prj\ inject\ f = (\lambda S.\ inject\ S\ (f\ (prj\ S)))$

**definition**  $lift_s:: ('S \Rightarrow 's) \Rightarrow 's\ set \Rightarrow 'S\ set$   
**where**  $lift_s\ prj\ A = \{S.\ prj\ S \in A\}$

**definition**  $lift_r:: ('S \Rightarrow 's) \Rightarrow ('S \Rightarrow 's \Rightarrow 'S) \Rightarrow ('s \times 's)\ set$   
 $\Rightarrow ('S \times 'S)\ set$

**where**

$lift_r\ prj\ inject\ R = \{(S, T).\ (prj\ S, prj\ T) \in R \wedge T = inject\ S\ (prj\ T)\}$

**primrec**  $lift_c:: ('S \Rightarrow 's) \Rightarrow ('S \Rightarrow 's \Rightarrow 'S) \Rightarrow ('s, 'p, 'f)\ com \Rightarrow ('S, 'p, 'f)\ com$

**where**

$lift_c\ prj\ inject\ Skip = Skip\ |$   
 $lift_c\ prj\ inject\ (Basic\ f) = Basic\ (lift_f\ prj\ inject\ f)\ |$   
 $lift_c\ prj\ inject\ (Spec\ r) = Spec\ (lift_r\ prj\ inject\ r)\ |$   
 $lift_c\ prj\ inject\ (Seq\ c_1\ c_2) =$   
 $\quad (Seq\ (lift_c\ prj\ inject\ c_1)\ (lift_c\ prj\ inject\ c_2))\ |$   
 $lift_c\ prj\ inject\ (Cond\ b\ c_1\ c_2) =$   
 $\quad Cond\ (lift_s\ prj\ b)\ (lift_c\ prj\ inject\ c_1)\ (lift_c\ prj\ inject\ c_2)\ |$   
 $lift_c\ prj\ inject\ (While\ b\ c) =$

$While (lift_s\ prj\ b)\ (lift_c\ prj\ inject\ c) \mid$   
 $lift_c\ prj\ inject\ (Call\ p) = Call\ p \mid$   
 $lift_c\ prj\ inject\ (DynCom\ c) = DynCom\ (\lambda s.\ lift_c\ prj\ inject\ (c\ (prj\ s))) \mid$   
 $lift_c\ prj\ inject\ (Guard\ f\ g\ c) = Guard\ f\ (lift_s\ prj\ g)\ (lift_c\ prj\ inject\ c) \mid$   
 $lift_c\ prj\ inject\ Throw = Throw \mid$   
 $lift_c\ prj\ inject\ (Catch\ c_1\ c_2) =$   
 $Catch\ (lift_c\ prj\ inject\ c_1)\ (lift_c\ prj\ inject\ c_2)$

**lemma**  $lift_c$ -Skip:  $(lift_c\ prj\ inject\ c = Skip) = (c = Skip)$   
 $\langle proof \rangle$

**lemma**  $lift_c$ -Basic:  
 $(lift_c\ prj\ inject\ c = Basic\ lf) = (\exists f.\ c = Basic\ f \wedge lf = lift_f\ prj\ inject\ f)$   
 $\langle proof \rangle$

**lemma**  $lift_c$ -Spec:  
 $(lift_c\ prj\ inject\ c = Spec\ lr) = (\exists r.\ c = Spec\ r \wedge lr = lift_r\ prj\ inject\ r)$   
 $\langle proof \rangle$

**lemma**  $lift_c$ -Seq:  
 $(lift_c\ prj\ inject\ c = Seq\ lc_1\ lc_2) =$   
 $(\exists\ c_1\ c_2.\ c = Seq\ c_1\ c_2 \wedge$   
 $lc_1 = lift_c\ prj\ inject\ c_1 \wedge lc_2 = lift_c\ prj\ inject\ c_2)$   
 $\langle proof \rangle$

**lemma**  $lift_c$ -Cond:  
 $(lift_c\ prj\ inject\ c = Cond\ lb\ lc_1\ lc_2) =$   
 $(\exists\ b\ c_1\ c_2.\ c = Cond\ b\ c_1\ c_2 \wedge lb = lift_s\ prj\ b \wedge$   
 $lc_1 = lift_c\ prj\ inject\ c_1 \wedge lc_2 = lift_c\ prj\ inject\ c_2)$   
 $\langle proof \rangle$

**lemma**  $lift_c$ -While:  
 $(lift_c\ prj\ inject\ c = While\ lb\ lc') =$   
 $(\exists\ b\ c'.\ c = While\ b\ c' \wedge lb = lift_s\ prj\ b \wedge$   
 $lc' = lift_c\ prj\ inject\ c')$   
 $\langle proof \rangle$

**lemma**  $lift_c$ -Call:  
 $(lift_c\ prj\ inject\ c = Call\ p) = (c = Call\ p)$   
 $\langle proof \rangle$

**lemma**  $lift_c$ -DynCom:  
 $(lift_c\ prj\ inject\ c = DynCom\ lc) =$   
 $(\exists C.\ c = DynCom\ C \wedge lc = (\lambda s.\ lift_c\ prj\ inject\ (C\ (prj\ s))))$   
 $\langle proof \rangle$

**lemma**  $lift_c$ -Guard:

$(\text{lift}_c \text{ prj inject } c = \text{Guard } f \text{ lg } lc') =$   
 $(\exists g \ c'. \ c = \text{Guard } f \ g \ c' \wedge \text{lg} = \text{lift}_s \text{ prj } g \wedge$   
 $lc' = \text{lift}_c \text{ prj inject } c')$   
 <proof>

**lemma** *lift<sub>c</sub>-Throw*:

$(\text{lift}_c \text{ prj inject } c = \text{Throw}) = (c = \text{Throw})$   
 <proof>

**lemma** *lift<sub>c</sub>-Catch*:

$(\text{lift}_c \text{ prj inject } c = \text{Catch } lc_1 \ lc_2) =$   
 $(\exists \ c_1 \ c_2. \ c = \text{Catch } c_1 \ c_2 \wedge$   
 $lc_1 = \text{lift}_c \text{ prj inject } c_1 \wedge lc_2 = \text{lift}_c \text{ prj inject } c_2)$   
 <proof>

**definition** *xstate-map*:: ( $'S \Rightarrow 's$ )  $\Rightarrow$  ( $'S, 'f$ ) *xstate*  $\Rightarrow$  ( $'s, 'f$ ) *xstate*  
**where**

*xstate-map*  $g \ x = (\text{case } x \text{ of}$   
 $\quad \text{Normal } s \Rightarrow \text{Normal } (g \ s)$   
 $\quad | \text{Abrupt } s \Rightarrow \text{Abrupt } (g \ s)$   
 $\quad | \text{Fault } f \Rightarrow \text{Fault } f$   
 $\quad | \text{Stuck} \Rightarrow \text{Stuck})$

**lemma** *xstate-map-simps* [*simp*]:

$\text{xstate-map } g \ (\text{Normal } s) = \text{Normal } (g \ s)$   
 $\text{xstate-map } g \ (\text{Abrupt } s) = \text{Abrupt } (g \ s)$   
 $\text{xstate-map } g \ (\text{Fault } f) = (\text{Fault } f)$   
 $\text{xstate-map } g \ \text{Stuck} = \text{Stuck}$   
 <proof>

**lemma** *xstate-map-Normal-conv*:

$\text{xstate-map } g \ S = \text{Normal } s = (\exists s'. \ S = \text{Normal } s' \wedge s = g \ s')$   
 <proof>

**lemma** *xstate-map-Abrupt-conv*:

$\text{xstate-map } g \ S = \text{Abrupt } s = (\exists s'. \ S = \text{Abrupt } s' \wedge s = g \ s')$   
 <proof>

**lemma** *xstate-map-Fault-conv*:

$\text{xstate-map } g \ S = \text{Fault } f = (S = \text{Fault } f)$   
 <proof>

**lemma** *xstate-map-Stuck-conv*:

$\text{xstate-map } g \ S = \text{Stuck} = (S = \text{Stuck})$   
 <proof>

**lemmas** *xstate-map-convs* = *xstate-map-Normal-conv* *xstate-map-Abrupt-conv*

*xstate-map-Fault-conv xstate-map-Stuck-conv*

**definition** *state*:: ('s,'f) *xstate*  $\Rightarrow$  's

**where**

*state* *x* = (case *x* of  
     Normal *s*  $\Rightarrow$  *s*  
     | Abrupt *s*  $\Rightarrow$  *s*  
     | Fault *g*  $\Rightarrow$  undefined  
     | Stuck  $\Rightarrow$  undefined)

**lemma** *state-simps* [*simp*]:

*state* (Normal *s*) = *s*

*state* (Abrupt *s*) = *s*

*<proof>*

**locale** *lift-state-space* =

**fixes** *project*::'S  $\Rightarrow$  's

**fixes** *inject*::'S  $\Rightarrow$  's  $\Rightarrow$  'S

**fixes** *project<sub>x</sub>*::('S,'f) *xstate*  $\Rightarrow$  ('s,'f) *xstate*

**fixes** *lift<sub>e</sub>*::('s,'p,'f) *body*  $\Rightarrow$  ('S,'p,'f) *body*

**fixes** *lift<sub>c</sub>*:: ('s,'p,'f) *com*  $\Rightarrow$  ('S,'p,'f) *com*

**fixes** *lift<sub>f</sub>*:: ('s  $\Rightarrow$  's)  $\Rightarrow$  ('S  $\Rightarrow$  'S)

**fixes** *lift<sub>s</sub>*:: 's *set*  $\Rightarrow$  'S *set*

**fixes** *lift<sub>r</sub>*:: ('s  $\times$  's) *set*  $\Rightarrow$  ('S  $\times$  'S) *set*

**assumes** *proj-inj-commute*:  $\bigwedge S s. \text{project } (\text{inject } S s) = s$

**defines** *lift<sub>c</sub>*  $\equiv$  *Compose.lift<sub>c</sub> project inject*

**defines** *project<sub>x</sub>*  $\equiv$  *xstate-map project*

**defines** *lift<sub>e</sub>*  $\equiv$  ( $\lambda \Gamma p. \text{map-option lift}_c (\Gamma p)$ )

**defines** *lift<sub>f</sub>*  $\equiv$  *Compose.lift<sub>f</sub> project inject*

**defines** *lift<sub>s</sub>*  $\equiv$  *Compose.lift<sub>s</sub> project*

**defines** *lift<sub>r</sub>*  $\equiv$  *Compose.lift<sub>r</sub> project inject*

**lemma** (in *lift-state-space*) *lift<sub>f</sub>-simp*:

*lift<sub>f</sub> f*  $\equiv$   $\lambda S. \text{inject } S (f (\text{project } S))$

*<proof>*

**lemma** (in *lift-state-space*) *lift<sub>s</sub>-simp*:

*lift<sub>s</sub> A*  $\equiv$  {*S. project S*  $\in$  *A*}

*<proof>*

**lemma** (in *lift-state-space*) *lift<sub>r</sub>-simp*:

*lift<sub>r</sub> R*  $\equiv$  {(*S,T*). (*project S,project T*)  $\in$  *R*  $\wedge$  *T=inject S (project T)*}

*<proof>*

**lemma** (in *lift-state-space*) *lift<sub>c</sub>-Skip-simp* [*simp*]:

*lift<sub>c</sub> Skip* = *Skip*

$\langle \text{proof} \rangle$   
**lemma** (in *lift-state-space*) *lift<sub>c</sub>-Basic-simp* [*simp*]:  
 $\text{lift}_c (\text{Basic } f) = \text{Basic } (\text{lift}_f f)$   
 $\langle \text{proof} \rangle$   
**lemma** (in *lift-state-space*) *lift<sub>c</sub>-Spec-simp* [*simp*]:  
 $\text{lift}_c (\text{Spec } r) = \text{Spec } (\text{lift}_r r)$   
 $\langle \text{proof} \rangle$   
**lemma** (in *lift-state-space*) *lift<sub>c</sub>-Seq-simp* [*simp*]:  
 $\text{lift}_c (\text{Seq } c_1 \ c_2) =$   
 $(\text{Seq } (\text{lift}_c c_1) (\text{lift}_c c_2))$   
 $\langle \text{proof} \rangle$   
**lemma** (in *lift-state-space*) *lift<sub>c</sub>-Cond-simp* [*simp*]:  
 $\text{lift}_c (\text{Cond } b \ c_1 \ c_2) =$   
 $\text{Cond } (\text{lift}_s b) (\text{lift}_c c_1) (\text{lift}_c c_2)$   
 $\langle \text{proof} \rangle$   
**lemma** (in *lift-state-space*) *lift<sub>c</sub>-While-simp* [*simp*]:  
 $\text{lift}_c (\text{While } b \ c) =$   
 $\text{While } (\text{lift}_s b) (\text{lift}_c c)$   
 $\langle \text{proof} \rangle$   
**lemma** (in *lift-state-space*) *lift<sub>c</sub>-Call-simp* [*simp*]:  
 $\text{lift}_c (\text{Call } p) = \text{Call } p$   
 $\langle \text{proof} \rangle$   
**lemma** (in *lift-state-space*) *lift<sub>c</sub>-DynCom-simp* [*simp*]:  
 $\text{lift}_c (\text{DynCom } c) = \text{DynCom } (\lambda s. \text{lift}_c (c (\text{project } s)))$   
 $\langle \text{proof} \rangle$   
**lemma** (in *lift-state-space*) *lift<sub>c</sub>-Guard-simp* [*simp*]:  
 $\text{lift}_c (\text{Guard } f \ g \ c) = \text{Guard } f (\text{lift}_s g) (\text{lift}_c c)$   
 $\langle \text{proof} \rangle$   
**lemma** (in *lift-state-space*) *lift<sub>c</sub>-Throw-simp* [*simp*]:  
 $\text{lift}_c \text{Throw} = \text{Throw}$   
 $\langle \text{proof} \rangle$   
**lemma** (in *lift-state-space*) *lift<sub>c</sub>-Catch-simp* [*simp*]:  
 $\text{lift}_c (\text{Catch } c_1 \ c_2) =$   
 $\text{Catch } (\text{lift}_c c_1) (\text{lift}_c c_2)$   
 $\langle \text{proof} \rangle$   
  
**lemma** (in *lift-state-space*) *project<sub>x</sub>-def'*:  
 $\text{project}_x s \equiv (\text{case } s \text{ of}$   
 $\quad \text{Normal } s \Rightarrow \text{Normal } (\text{project } s)$   
 $\quad | \text{Abrupt } s \Rightarrow \text{Abrupt } (\text{project } s)$   
 $\quad | \text{Fault } f \Rightarrow \text{Fault } f$   
 $\quad | \text{Stuck} \Rightarrow \text{Stuck})$   
 $\langle \text{proof} \rangle$   
  
**lemma** (in *lift-state-space*) *lift<sub>e</sub>-def'*:  
 $\text{lift}_e \Gamma \ p \equiv (\text{case } \Gamma \ p \text{ of } \text{Some } \text{bdy} \Rightarrow \text{Some } (\text{lift}_c \text{bdy}) \ | \ \text{None} \Rightarrow \text{None})$   
 $\langle \text{proof} \rangle$

The problem is that  $\text{lift}_c \text{project inject} \circ \Gamma$  is quite a strong premise. The

problem is that  $\Gamma$  is a function here. A map would be better. We only have to lift those procedures in the domain of  $\Gamma$ :  $\Gamma p = \text{Some bdy} \longrightarrow \Gamma' p = \text{Some lift}_c \text{ project inject bdy}$ . We then can com up with theorems that allow us to extend the domains of  $\Gamma$  and preserve validity.

**lemma (in lift-state-space)**  
 $\{(S, T). \exists t. (\text{project } S, t) \in r \wedge T = \text{inject } S \ t\}$   
 $\subseteq \{(S, T). (\text{project } S, \text{project } T) \in r \wedge T = \text{inject } S \ (\text{project } T)\}$   
 $\langle \text{proof} \rangle$

**lemma (in lift-state-space)**  
 $\{(S, T). (\text{project } S, \text{project } T) \in r \wedge T = \text{inject } S \ (\text{project } T)\}$   
 $\subseteq \{(S, T). \exists t. (\text{project } S, t) \in r \wedge T = \text{inject } S \ t\}$   
 $\langle \text{proof} \rangle$

**lemma (in lift-state-space) lift-exec:**  
**assumes** *exec-lc*:  $(\text{lift}_e \Gamma) \vdash \langle \text{lc}, s \rangle \Rightarrow t$   
**shows**  $\bigwedge c. \llbracket \text{lift}_c \ c = \text{lc} \rrbracket \Longrightarrow$   
 $\Gamma \vdash \langle c, \text{project}_x \ s \rangle \Rightarrow \text{project}_x \ t$   
 $\langle \text{proof} \rangle$

**lemma (in lift-state-space) lift-exec':**  
**assumes** *exec-lc*:  $(\text{lift}_e \Gamma) \vdash \langle \text{lift}_c \ c, s \rangle \Rightarrow t$   
**shows**  $\Gamma \vdash \langle c, \text{project}_x \ s \rangle \Rightarrow \text{project}_x \ t$   
 $\langle \text{proof} \rangle$

**lemma (in lift-state-space) lift-valid:**  
**assumes** *valid*:  $\Gamma \models_{/F} P \ c \ Q, A$   
**shows**  
 $(\text{lift}_e \Gamma) \models_{/F} (\text{lift}_s \ P) \ (\text{lift}_c \ c) \ (\text{lift}_s \ Q), (\text{lift}_s \ A)$   
 $\langle \text{proof} \rangle$

**lemma (in lift-state-space) lift-hoarep:**  
**assumes** *deriv*:  $\Gamma, \{\} \vdash_{/F} P \ c \ Q, A$   
**shows**  
 $(\text{lift}_e \Gamma), \{\} \vdash_{/F} (\text{lift}_s \ P) \ (\text{lift}_c \ c) \ (\text{lift}_s \ Q), (\text{lift}_s \ A)$   
 $\langle \text{proof} \rangle$

**lemma (in lift-state-space) lift-hoarep':**  
 $\forall Z. \Gamma, \{\} \vdash_{/F} (P \ Z) \ c \ (Q \ Z), (A \ Z) \Longrightarrow$   
 $\forall Z. (\text{lift}_e \Gamma), \{\} \vdash_{/F} (\text{lift}_s \ (P \ Z)) \ (\text{lift}_c \ c)$   
 $(\text{lift}_s \ (Q \ Z)), (\text{lift}_s \ (A \ Z))$   
 $\langle \text{proof} \rangle$

**lemma** (in *lift-state-space*) *lift-termination*:

**assumes** *termi*:  $\Gamma \vdash c \downarrow s$

**shows**  $\bigwedge S. \text{project}_x S = s \implies$

$\text{lift}_e \Gamma \vdash (\text{lift}_c c) \downarrow S$

*<proof>*

**lemma** (in *lift-state-space*) *lift-termination'*:

**assumes** *termi*:  $\Gamma \vdash c \downarrow \text{project}_x S$

**shows**  $\text{lift}_e \Gamma \vdash (\text{lift}_c c) \downarrow S$

*<proof>*

**lemma** (in *lift-state-space*) *lift-validt*:

**assumes** *valid*:  $\Gamma \models_{t/F} P \ c \ Q, A$

**shows**  $(\text{lift}_e \Gamma) \models_{t/F} (\text{lift}_s P) \ (\text{lift}_c c) \ (\text{lift}_s Q), (\text{lift}_s A)$

*<proof>*

**lemma** (in *lift-state-space*) *lift-hoaret*:

**assumes** *deriv*:  $\Gamma, \{\} \vdash_{t/F} P \ c \ Q, A$

**shows**

$(\text{lift}_e \Gamma), \{\} \vdash_{t/F} (\text{lift}_s P) \ (\text{lift}_c c) \ (\text{lift}_s Q), (\text{lift}_s A)$

*<proof>*

**locale** *lift-state-space-ext* = *lift-state-space* +

**assumes** *inj-proj-commute*:  $\bigwedge S. \text{inject } S \ (\text{project } S) = S$

**assumes** *inject-last*:  $\bigwedge S \ s \ t. \text{inject } (\text{inject } S \ s) \ t = \text{inject } S \ t$

**lemma** (in *lift-state-space-ext*) *lift-exec-inject-same*:

**assumes** *exec-lc*:  $(\text{lift}_e \Gamma) \vdash \langle lc, s \rangle \Rightarrow t$

**shows**  $\bigwedge c. \llbracket \text{lift}_c c = lc; t \notin (\text{Fault} \ ' \ \text{UNIV}) \cup \{\text{Stuck}\} \rrbracket \implies$   
 $\text{state } t = \text{inject } (\text{state } s) \ (\text{project } (\text{state } t))$

*<proof>*

**lemma** (in *lift-state-space-ext*) *valid-inject-project*:

**assumes** *noFaultStuck*:

$\Gamma \vdash \langle c, \text{Normal } (\text{project } \sigma) \rangle \Rightarrow \notin (\text{Fault} \ ' \ \text{UNIV} \cup \{\text{Stuck}\})$

**shows**  $\text{lift}_e \Gamma \models_{t/F} \{\sigma\} \ \text{lift}_c c$

$\{t. t = \text{inject } \sigma \ (\text{project } t)\}, \{t. t = \text{inject } \sigma \ (\text{project } t)\}$

*<proof>*

**lemma** (in *lift-state-space-ext*) *lift-exec-inject-same'*:

**assumes** *exec-lc*:  $(\text{lift}_e \Gamma) \vdash \langle \text{lift}_c c, S \rangle \Rightarrow T$

**shows**  $\bigwedge c. \llbracket T \notin (\text{Fault} \ ' \ \text{UNIV}) \cup \{\text{Stuck}\} \rrbracket \implies$

$\text{state } T = \text{inject } (\text{state } S) \ (\text{project } (\text{state } T))$

*<proof>*

**lemma** (in *lift-state-space-ext*) *valid-lift-modifies*:  
**assumes** *valid*:  $\forall s. \Gamma \models_{/F} \{s\} c \text{ (Modif } s), (\text{ModifAbr } s)$   
**shows**  $(\text{lift}_e \Gamma) \models_{/F} \{S\} (\text{lift}_c c)$   
 $\{T. T \in \text{lift}_s (\text{Modif } (\text{project } S)) \wedge T = \text{inject } S (\text{project } T)\},$   
 $\{T. T \in \text{lift}_s (\text{ModifAbr } (\text{project } S)) \wedge T = \text{inject } S (\text{project } T)\}$   
 $\langle \text{proof} \rangle$

**lemma** (in *lift-state-space-ext*) *hoare-lift-modifies*:  
**assumes** *deriv*:  $\forall \sigma. \Gamma, \{\} \vdash_{/F} \{\sigma\} c \text{ (Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\forall \sigma. (\text{lift}_e \Gamma), \{\} \vdash_{/F} \{\sigma\} (\text{lift}_c c)$   
 $\{T. T \in \text{lift}_s (\text{Modif } (\text{project } \sigma)) \wedge T = \text{inject } \sigma (\text{project } T)\},$   
 $\{T. T \in \text{lift}_s (\text{ModifAbr } (\text{project } \sigma)) \wedge T = \text{inject } \sigma (\text{project } T)\}$   
 $\langle \text{proof} \rangle$

**lemma** (in *lift-state-space-ext*) *hoare-lift-modifies'*:  
**assumes** *deriv*:  $\forall \sigma. \Gamma, \{\} \vdash_{/F} \{\sigma\} c \text{ (Modif } \sigma), (\text{ModifAbr } \sigma)$   
**shows**  $\forall \sigma. (\text{lift}_e \Gamma), \{\} \vdash_{/F} \{\sigma\} (\text{lift}_c c)$   
 $\{T. T \in \text{lift}_s (\text{Modif } (\text{project } \sigma)) \wedge$   
 $\quad (\exists T'. T = \text{inject } \sigma T')\},$   
 $\{T. T \in \text{lift}_s (\text{ModifAbr } (\text{project } \sigma)) \wedge$   
 $\quad (\exists T'. T = \text{inject } \sigma T')\}$   
 $\langle \text{proof} \rangle$

## 25.2 Renaming Procedures

**primrec** *rename*::  $(p \Rightarrow q) \Rightarrow (s, p, f) \text{ com} \Rightarrow (s, q, f) \text{ com}$

**where**

$\text{rename } N \text{ Skip} = \text{Skip} \mid$   
 $\text{rename } N (\text{Basic } f) = \text{Basic } f \mid$   
 $\text{rename } N (\text{Spec } r) = \text{Spec } r \mid$   
 $\text{rename } N (\text{Seq } c_1 c_2) = (\text{Seq } (\text{rename } N c_1) (\text{rename } N c_2)) \mid$   
 $\text{rename } N (\text{Cond } b c_1 c_2) = \text{Cond } b (\text{rename } N c_1) (\text{rename } N c_2) \mid$   
 $\text{rename } N (\text{While } b c) = \text{While } b (\text{rename } N c) \mid$   
 $\text{rename } N (\text{Call } p) = \text{Call } (N p) \mid$   
 $\text{rename } N (\text{DynCom } c) = \text{DynCom } (\lambda s. \text{rename } N (c s)) \mid$   
 $\text{rename } N (\text{Guard } f g c) = \text{Guard } f g (\text{rename } N c) \mid$   
 $\text{rename } N \text{ Throw} = \text{Throw} \mid$   
 $\text{rename } N (\text{Catch } c_1 c_2) = \text{Catch } (\text{rename } N c_1) (\text{rename } N c_2)$

**lemma** *rename-Skip*:  $\text{rename } h c = \text{Skip} = (c = \text{Skip})$   
 $\langle \text{proof} \rangle$

**lemma** *rename-Basic*:  
 $(\text{rename } h c = \text{Basic } f) = (c = \text{Basic } f)$   
 $\langle \text{proof} \rangle$

**lemma** *rename-Spec*:

$(\text{rename } h \ c = \text{Spec } r) = (c = \text{Spec } r)$   
 ⟨proof⟩

**lemma** *rename-Seq*:

$(\text{rename } h \ c = \text{Seq } rc_1 \ rc_2) =$   
 $(\exists \ c_1 \ c_2. \ c = \text{Seq } c_1 \ c_2 \wedge$   
 $rc_1 = \text{rename } h \ c_1 \wedge rc_2 = \text{rename } h \ c_2)$   
 ⟨proof⟩

**lemma** *rename-Cond*:

$(\text{rename } h \ c = \text{Cond } b \ rc_1 \ rc_2) =$   
 $(\exists \ c_1 \ c_2. \ c = \text{Cond } b \ c_1 \ c_2 \wedge rc_1 = \text{rename } h \ c_1 \wedge rc_2 = \text{rename } h \ c_2)$   
 ⟨proof⟩

**lemma** *rename-While*:

$(\text{rename } h \ c = \text{While } b \ rc') = (\exists \ c'. \ c = \text{While } b \ c' \wedge rc' = \text{rename } h \ c')$   
 ⟨proof⟩

**lemma** *rename-Call*:

$(\text{rename } h \ c = \text{Call } q) = (\exists \ p. \ c = \text{Call } p \wedge q = h \ p)$   
 ⟨proof⟩

**lemma** *rename-DynCom*:

$(\text{rename } h \ c = \text{DynCom } rc) = (\exists \ C. \ c = \text{DynCom } C \wedge rc = (\lambda s. \text{rename } h \ (C \ s)))$   
 ⟨proof⟩

**lemma** *rename-Guard*:

$(\text{rename } h \ c = \text{Guard } f \ g \ rc') =$   
 $(\exists \ c'. \ c = \text{Guard } f \ g \ c' \wedge rc' = \text{rename } h \ c')$   
 ⟨proof⟩

**lemma** *rename-Throw*:

$(\text{rename } h \ c = \text{Throw}) = (c = \text{Throw})$   
 ⟨proof⟩

**lemma** *rename-Catch*:

$(\text{rename } h \ c = \text{Catch } rc_1 \ rc_2) =$   
 $(\exists \ c_1 \ c_2. \ c = \text{Catch } c_1 \ c_2 \wedge rc_1 = \text{rename } h \ c_1 \wedge rc_2 = \text{rename } h \ c_2)$   
 ⟨proof⟩

**lemma** *exec-rename-to-exec*:

**assumes**  $\Gamma: \forall p \ bdy. \ \Gamma \ p = \text{Some } bdy \longrightarrow \Gamma' (h \ p) = \text{Some } (\text{rename } h \ bdy)$

**assumes** *exec*:  $\Gamma \vdash \langle rc, s \rangle \Rightarrow t$

**shows**  $\bigwedge c. \ \text{rename } h \ c = rc \Longrightarrow \exists t'. \ \Gamma \vdash \langle c, s \rangle \Rightarrow t' \wedge (t' = \text{Stuck} \vee t' = t)$

⟨proof⟩

**lemma** *exec-rename-to-exec'*:

**assumes**  $\Gamma: \forall p \text{ bdy}. \Gamma p = \text{Some bdy} \longrightarrow \Gamma' (N p) = \text{Some} (\text{rename } N \text{ bdy})$   
**assumes** *exec*:  $\Gamma \vdash \langle \text{rename } N \text{ c, s} \rangle \Rightarrow t$   
**shows**  $\exists t'. \Gamma \vdash \langle c, s \rangle \Rightarrow t' \wedge (t' = \text{Stuck} \vee t' = t)$   
*<proof>*

**lemma** *valid-to-valid-rename*:

**assumes**  $\Gamma: \forall p \text{ bdy}. \Gamma p = \text{Some bdy} \longrightarrow \Gamma' (N p) = \text{Some} (\text{rename } N \text{ bdy})$   
**assumes** *valid*:  $\Gamma \models_{/F} P \text{ c } Q, A$   
**shows**  $\Gamma' \models_{/F} P (\text{rename } N \text{ c}) Q, A$   
*<proof>*

**lemma** *hoare-to-hoare-rename*:

**assumes**  $\Gamma: \forall p \text{ bdy}. \Gamma p = \text{Some bdy} \longrightarrow \Gamma' (N p) = \text{Some} (\text{rename } N \text{ bdy})$   
**assumes** *deriv*:  $\Gamma, \{\} \vdash_{/F} P \text{ c } Q, A$   
**shows**  $\Gamma', \{\} \vdash_{/F} P (\text{rename } N \text{ c}) Q, A$   
*<proof>*

**lemma** *hoare-to-hoare-rename'*:

**assumes**  $\Gamma: \forall p \text{ bdy}. \Gamma p = \text{Some bdy} \longrightarrow \Gamma' (N p) = \text{Some} (\text{rename } N \text{ bdy})$   
**assumes** *deriv*:  $\forall Z. \Gamma, \{\} \vdash_{/F} (P Z) \text{ c } (Q Z), (A Z)$   
**shows**  $\forall Z. \Gamma', \{\} \vdash_{/F} (P Z) (\text{rename } N \text{ c}) (Q Z), (A Z)$   
*<proof>*

**lemma** *terminates-to-terminates-rename*:

**assumes**  $\Gamma: \forall p \text{ bdy}. \Gamma p = \text{Some bdy} \longrightarrow \Gamma' (N p) = \text{Some} (\text{rename } N \text{ bdy})$   
**assumes** *termi*:  $\Gamma \vdash c \downarrow s$   
**assumes** *noStuck*:  $\Gamma \vdash \langle c, s \rangle \Rightarrow \notin \{\text{Stuck}\}$   
**shows**  $\Gamma \vdash \text{rename } N \text{ c } \downarrow s$   
*<proof>*

**lemma** *validt-to-validt-rename*:

**assumes**  $\Gamma: \forall p \text{ bdy}. \Gamma p = \text{Some bdy} \longrightarrow \Gamma' (N p) = \text{Some} (\text{rename } N \text{ bdy})$   
**assumes** *valid*:  $\Gamma \models_{t/F} P \text{ c } Q, A$   
**shows**  $\Gamma' \models_{t/F} P (\text{rename } N \text{ c}) Q, A$   
*<proof>*

**lemma** *hoaret-to-hoaret-rename*:

**assumes**  $\Gamma: \forall p \text{ bdy}. \Gamma p = \text{Some bdy} \longrightarrow \Gamma' (N p) = \text{Some} (\text{rename } N \text{ bdy})$   
**assumes** *deriv*:  $\Gamma, \{\} \vdash_{t/F} P \text{ c } Q, A$   
**shows**  $\Gamma', \{\} \vdash_{t/F} P (\text{rename } N \text{ c}) Q, A$   
*<proof>*

**lemma** *hoaret-to-hoaret-rename'*:

**assumes**  $\Gamma: \forall p \text{ bdy}. \Gamma p = \text{Some bdy} \longrightarrow \Gamma' (N p) = \text{Some} (\text{rename } N \text{ bdy})$   
**assumes** *deriv*:  $\forall Z. \Gamma, \{\} \vdash_{t/F} (P Z) \text{ c } (Q Z), (A Z)$

**shows**  $\forall Z. \Gamma', \{\} \vdash_{t/F} (P Z) (\text{rename } N c) (Q Z), (A Z)$   
 <proof>

**lemma** *lift<sub>c</sub>-whileAnno* [simp]: *lift<sub>c</sub> prj inject (whileAnno b I V c) =*  
*whileAnno (lift<sub>s</sub> prj b)*  
*(lift<sub>s</sub> prj I) (lift<sub>r</sub> prj inject V) (lift<sub>c</sub> prj inject c)*  
 <proof>

**lemma** *lift<sub>c</sub>-block* [simp]: *lift<sub>c</sub> prj inject (block init bdy return c) =*  
*block (lift<sub>f</sub> prj inject init) (lift<sub>c</sub> prj inject bdy)*  
*(λs. (lift<sub>f</sub> prj inject (return (prj s))))*  
*(λs t. lift<sub>c</sub> prj inject (c (prj s) (prj t)))*  
 <proof>

**lemma** *lift<sub>c</sub>-call* [simp]: *lift<sub>c</sub> prj inject (call init p return c) =*  
*call (lift<sub>f</sub> prj inject init) p*  
*(λs. (lift<sub>f</sub> prj inject (return (prj s))))*  
*(λs t. lift<sub>c</sub> prj inject (c (prj s) (prj t)))*  
 <proof>

**lemma** *rename-whileAnno* [simp]: *rename h (whileAnno b I V c) =*  
*whileAnno b I V (rename h c)*  
 <proof>

**lemma** *rename-block* [simp]: *rename h (block init bdy return c) =*  
*block init (rename h bdy) return (λs t. rename h (c s t))*  
 <proof>

**lemma** *rename-call* [simp]: *rename h (call init p return c) =*  
*call init (h p) return (λs t. rename h (c s t))*  
 <proof>

**end**

**theory** *ComposeEx* **imports** *Compose ../Vcg ../HeapList* **begin**

**record** *globals-list* =  
*next-'* :: *ref* ⇒ *ref*

**record** *state-list* = *globals-list state* +  
*p-'* :: *ref*  
*sl-q-'* :: *ref*  
*r-'* :: *ref*

**procedures** *Rev(p|sl-q)* =

```

    'sl-q ::= Null;;
    WHILE 'p ≠ Null
    DO
      'r ::= 'p;; {'p ≠ Null} → 'p ::= 'p → 'next;;
      {'r ≠ Null} → 'r → 'next ::= 'sl-q;; 'sl-q ::= 'r
    OD

```

**print-theorems**

**lemma** (in *Rev-impl*)

*Rev-modifies*:  
 $\forall \sigma. \Gamma \vdash /UNIV \{ \sigma \} \text{'sl-q} ::= PROC \text{Rev}('p) \{ t. t \text{ may-only-modify-globals } \sigma \text{ in } [next] \}$   
 ⟨proof⟩

**lemma** (in *Rev-impl*) **shows**

*Rev-spec*:  
 $\forall Ps. \Gamma \vdash \{ List 'p 'next Ps \} \text{'sl-q} ::= PROC \text{Rev}('p) \{ List 'sl-q 'next (rev Ps) \}$   
 ⟨proof⟩

**declare** [[*names-unique* = false]]

**record** *globals* =

*strnext-'* :: ref ⇒ ref  
*chr-'* :: ref ⇒ char

*qnext-'* :: ref ⇒ ref  
*cont-'* :: ref ⇒ int

**record** *state* = *globals state* +

*str-'* :: ref  
*queue-'* :: ref  
*q-'* :: ref  
*r-'* :: ref

**definition** *project-globals-str*:: *globals* ⇒ *globals-list*

**where** *project-globals-str* *g* = (*next-'* = *strnext-'* *g*)

**definition** *project-str*:: *state* ⇒ *state-list*

**where**

*project-str* *s* =  
 (*globals* = *project-globals-str* (*globals* *s*),  
*state-list.p-'* = *str-'* *s*, *sl-q-'* = *q-'* *s*, *state-list.r-'* = *r-'* *s*)

**definition** *inject-globals-str*::

*globals* ⇒ *globals-list* ⇒ *globals*

**where**

*inject-globals-str* *G* *g* =

$G(\text{strnext}' := \text{next}' g)$

**definition**  $\text{inject-str}::\text{state} \Rightarrow \text{state-list} \Rightarrow \text{state}$  **where**  
 $\text{inject-str } S s = S(\text{globals} := \text{inject-globals-str } (\text{globals } S) (\text{globals } s),$   
 $\text{str}' := \text{state-list.p}' s, q' := \text{sl-q}' s,$   
 $r' := \text{state-list.r}' s)$

**lemma**  $\text{globals-inject-project-str-commutes}$ :  
 $\text{inject-globals-str } G (\text{project-globals-str } G) = G$   
(proof)

**lemma**  $\text{inject-project-str-commutes}$ :  $\text{inject-str } S (\text{project-str } S) = S$   
(proof)

**lemma**  $\text{globals-project-inject-str-commutes}$ :  
 $\text{project-globals-str } (\text{inject-globals-str } G g) = g$   
(proof)

**lemma**  $\text{project-inject-str-commutes}$ :  $\text{project-str } (\text{inject-str } S s) = s$   
(proof)

**lemma**  $\text{globals-inject-str-last}$ :  
 $\text{inject-globals-str } (\text{inject-globals-str } G g) g' = \text{inject-globals-str } G g'$   
(proof)

**lemma**  $\text{inject-str-last}$ :  
 $\text{inject-str } (\text{inject-str } S s) s' = \text{inject-str } S s'$   
(proof)

**definition**  
 $\text{lift}_e = (\lambda \Gamma p. \text{map-option } (\text{lift}_c \text{ project-str inject-str}) (\Gamma p))$

**print-locale**  $\text{lift-state-space}$

**interpretation**  $\text{ex: lift-state-space project-str inject-str}$   
 $\text{xstate-map project-str lift}_e \text{ lift}_c \text{ project-str inject-str}$   
 $\text{lift}_f \text{ project-str inject-str lift}_s \text{ project-str}$   
 $\text{lift}_r \text{ project-str inject-str}$   
(proof)

**interpretation**  $\text{ex: lift-state-space-ext project-str inject-str}$   
 $\text{xstate-map project-str lift}_e \text{ lift}_c \text{ project-str inject-str}$   
 $\text{lift}_f \text{ project-str inject-str lift}_s \text{ project-str}$   
 $\text{lift}_r \text{ project-str inject-str}$

(proof)

**lemmas** *Rev-lift-spec* = *ex.lift-hoarep'* [*OF Rev-impl.Rev-spec,simplified lift<sub>s</sub>-def project-str-def project-globals-str-def,simplified, of - "Rev"*]

**print-theorems**

**definition**  $\mathcal{N} \ p' \ p = (\text{if } p = \text{"Rev"} \text{ then } p' \text{ else ""})$

**procedures** *RevStr(str|q)* = *rename* ( $\mathcal{N}$  *RevStr-'proc*)  
*(lift<sub>c</sub> project-str inject-str (Rev-body.Rev-body))*

**lemmas** *Rev-lift-spec'* =  
*Rev-lift-spec* [*of ["Rev" ↦ Rev-body.Rev-body]* ,  
*simplified Rev-impl-def Rev-clique-def,simplified*]

**thm** *Rev-lift-spec'*

**lemma** *Rev-lift-spec''*:  
 $\forall Ps. \text{lift}_e [\text{"Rev"} \mapsto \text{Rev-body.Rev-body}]$   
 $\vdash \{\{List \ 'str \ 'strnext \ Ps\} \text{ Call "Rev"} \{\{List \ 'q \ 'strnext \ (rev \ Ps)\}\}$   
 $\langle \text{proof} \rangle$

**lemma** (**in** *RevStr-impl*) *N-ok*:  
 $\forall p \text{ bdy. } (\text{lift}_e [\text{"Rev"} \mapsto \text{Rev-body.Rev-body}]) \ p = \text{Some bdy} \longrightarrow$   
 $\Gamma (\mathcal{N} \ \text{RevStr-'proc} \ p) = \text{Some } (\text{rename } (\mathcal{N} \ \text{RevStr-'proc}) \ \text{bdy})$   
 $\langle \text{proof} \rangle$

**context** *RevStr-impl*

**begin**

**thm** *hoare-to-hoare-rename'* [*OF - Rev-lift-spec''*, *OF N-ok*,  
*simplified N-def, simplified* ]

**end**

**lemmas** (**in** *RevStr-impl*) *RevStr-spec* =  
*hoare-to-hoare-rename'* [*OF - Rev-lift-spec''*, *OF N-ok*,  
*simplified N-def, simplified* ]

**lemma** (**in** *RevStr-impl*) *RevStr-spec'*:  
 $\forall Ps. \Gamma \vdash \{\{List \ 'str \ 'strnext \ Ps\} \ 'q := \text{PROC } \text{RevStr}(\ 'str)$   
 $\{\{List \ 'q \ 'strnext \ (rev \ Ps)\}\}$   
 $\langle \text{proof} \rangle$

**lemmas** *Rev-modifies'* =

*Rev-impl.Rev-modifies* [*of ["Rev" ↦ Rev-body.Rev-body]*, *simplified Rev-impl-def*,  
*simplified*]

**thm** *Rev-modifies'*

```

context RevStr-impl
begin
lemmas RevStr-modifies' =
  hoare-to-hoare-rename' [OF - ex.hoare-lift-modifies' [OF Rev-modifies'],
    OF N-ok, of "Rev", simplified N-def Rev-clique-def,simplified]
end

```

```

lemma (in RevStr-impl) RevStr-modifies:
 $\forall \sigma. \Gamma \vdash_{UNIV} \{\sigma\} \text{'str} ::= PROC \text{RevStr}(\text{'str})$ 
  {t. t may-only-modify-globals  $\sigma$  in [strnext]}
  <proof>

```

```

end

```

## 26 User Guide

We introduce the verification environment with a couple of examples that illustrate how to use the different bits and pieces to verify programs.

### 26.1 Basics

First of all we have to decide how to represent the state space. There are currently two implementations. One is based on records the other one on the concept called 'statespace' that was introduced with Isabelle 2007 (see HOL/Statespace) . In contrast to records a 'statespace' does not define a new type, but provides a notion of state, based on locales. Logically the state is modelled as a function from (abstract) names to (abstract) values and the statespace infrastructure organises distinctness of names an projection/injection of concrete values into the abstract one. Towards the user the interface of records and statespaces is quite similar. However, statespaces offer more flexibility, inherited from the locale infrastructure, in particular multiple inheritance and renaming of components.

In this user guide we prefer statespaces, but give some comments on the usage of records in Section 26.9.

```

hoarestate vars =
  A :: nat
  I :: nat
  M :: nat
  N :: nat
  R :: nat
  S :: nat

```

The command **hoarestate** is a simple preprocessor for the command **statespaces** which decorates the state components with the suffix *'*, to avoid cluttering the namespace. Also note that underscores are printed as hyphens

in this documentation. So what you see as  $A'$  in this document is actually  $A_'$ . Every component name becomes a fixed variable in the locale *vars* and can no longer be used for logical variables.

Lookup of a component  $A'$  in a state  $s$  is written as  $s \cdot A'$ , and update with a value *term*  $v$  as  $s \langle A' := v \rangle$ .

To deal with local and global variables in the context of procedures the program state is organised as a record containing the two componets *locals* and *globals*. The variables defined in hoarestate *vars* reside in the *locals* part.

Here is a first example.

**lemma** (*in vars*)  $\Gamma \vdash \{ \!| N = 5 \!| \} 'N ::= 2 * 'N \{ \!| N = 10 \!| \}$   
*<proof>*

We enable the locale of statespace *vars* by the **in vars** directive. The verification condition generator is invoked via the *vcg* method and leaves us with the expected subgoal that can be proved by simplification.

If we refer to components (variables) of the state-space of the program we always mark these with  $'$  (in assertions and also in the program itself). It is the acute-symbol and is present on most keyboards. The assertions of the Hoare tuple are ordinary Isabelle sets. As we usually want to refer to the state space in the assertions, we provide special brackets for them. They can be written as  $\{ | \ | \}$  in ASCII or  $\{ \!| \!| \}$  with symbols. Internally, marking variables has two effects. First of all we refer to the implicit state and secondary we get rid of the suffix  $'$ . So the assertion  $\{ \!| N = 5 \!| \}$  internally gets expanded to  $\{ s \cdot locals \ s \cdot N' = 5 \}$  written in ordinary set comprehension notation of Isabelle. It describes the set of states where the  $N'$  component is equal to 5. An empty context and an empty postcondition for abrupt termination can be omitted. The lemma above is a shorthand for  $\Gamma, \{ \!| \!| \} \vdash \{ \!| N = 5 \!| \} 'N ::= 2 * 'N \{ \!| N = 10 \!| \}, \{ \!| \!| \}$ .

We can step through verification condition generation by the method *vcg-step*.

**lemma** (*in vars*)  $\Gamma, \{ \!| \!| \} \vdash \{ \!| N = 5 \!| \} 'N ::= 2 * 'N \{ \!| N = 10 \!| \}$   
*<proof>*

Although our assertions work semantically on the state space, stepping through verification condition generation “feels” like the expected syntactic substitutions of traditional Hoare logic. This is achieved by light simplification on the assertions calculated by the Hoare rules.

**lemma** (*in vars*)  $\Gamma \vdash \{ \!| N = 5 \!| \} 'N ::= 2 * 'N \{ \!| N = 10 \!| \}$   
*<proof>*

The next example shows how we deal with the while loop. Note the invariant annotation.

**lemma** (*in vars*)  
 $\Gamma, \{\} \vdash \{ \prime M = 0 \wedge \prime S = 0 \}$   
*WHILE*  $\prime M \neq a$   
*INV*  $\{ \prime S = \prime M * b \}$   
*DO*  $\prime S ::= \prime S + b;; \prime M ::= \prime M + 1$  *OD*  
 $\{ \prime S = a * b \}$   
*<proof>*

## 26.2 Procedures

### 26.2.1 Declaration

Our first procedure is a simple square procedure. We provide the command **procedures**, to declare and define a procedure.

**procedures**  
*Square* ( $N::nat|R::nat$ )  
**where**  $I::nat$  **in**  
 $\prime R ::= \prime N * \prime N$

A procedure is given by the signature of the procedure followed by the procedure body. The signature consists of the name of the procedure and a list of parameters together with their types. The parameters in front of the pipe | are value parameters and behind the pipe are the result parameters. Value parameters model call by value semantics. The value of a result parameter at the end of the procedure is passed back to the caller. Local variables follow the *where*. If there are no local variables the *where ... in* can be omitted. The variable  $I$  is actually unused in the body, but is used in the examples below.

The **procedures** command provides convenient syntax for procedure calls (that creates the proper *init*, *return* and *result* functions on the fly) and creates locales and statespaces to reason about the procedure. The purpose of locales is to set up logical contexts to support modular reasoning. Locales can be seen as freeze-dried proof contexts that get alive as you setup a new lemma or theorem ([2]). The locale the user deals with is named *Square-impl*. It defines the procedure name (internally *Square-'proc*), the procedure body (named *Square-body*) and the statespaces for parameters and local and global variables. Moreover it contains the assumption  $\Gamma$  *Square-'proc* = *Some Square-body*, which states that the procedure is properly defined in the procedure context.

The purpose of the locale is to give us easy means to setup the context in which we prove programs correct. In this locale the procedure context  $\Gamma$  is fixed. So we always use this letter for the procedure specification. This is crucial, if we prove programs under the assumption of some procedure specifications.

The **procedures** command generates syntax, so that we can either write

$CALL\ Square('I, 'R)$  or  $'I ::= CALL\ Square()$  for the procedure call. The internal term is the following:

$$\begin{aligned} &call\ (\lambda s.\ s(\langle locals := locals\ s\langle N-'Square-' := locals\ s\cdot I-'Square-' \rangle \rangle)) \\ &Square-'proc\ (\lambda s\ t.\ s(\langle globals := globals\ t \rangle)) \\ &(\lambda i\ t.\ 'R ::= locals\ t\cdot R-'Square-') \end{aligned}$$

Note the additional decoration (with the procedure name) of the parameter and local variable names.

The abstract syntax for the procedure call is *call init p return result*. The *init* function copies the values of the actual parameters to the formal parameters, the *return* function copies the global variables back (in our case there are no global variables), and the *result* function additionally copies the values of the formal result parameters to the actual locations. Actual value parameters can be all kind of expressions, since we only need their value. But result parameters must be proper “lvalues”: variables (including dereferenced pointers) or array locations, since we have to assign values to them.

### 26.2.2 Verification

A procedure specification is an ordinary Hoare tuple. We use the parameterless call for the specification;  $'R ::= PROC\ Square('N)$  is syntactic sugar for *Call Square-proc*. This emphasises that the specification describes the internal behaviour of the procedure, whereas parameter passing corresponds to the procedure call. The following precondition fixes the current value  $'N$  to the logical variable  $n$ . Universal quantification of  $n$  enables us to adapt the specification to an actual parameter. The specification is used in the rule for procedure call when we come upon a call to *Square*. Thus  $n$  plays the role of the auxiliary variable  $Z$ .

To verify the procedure we need to verify the body. We use a derived variant of the general recursion rule, tailored for non recursive procedures: *HoarePartial.ProcNoRec1*:

$$\begin{aligned} \llbracket \forall Z.\ \Gamma, \Theta \vdash_{/F} (P\ Z)\ the\ (\Gamma\ p)\ (Q\ Z), (A\ Z); p \in dom\ \Gamma \rrbracket &\implies \forall Z. \\ \Gamma, \Theta \vdash_{/F} (P\ Z)\ Call\ p\ (Q\ Z), (A\ Z) & \end{aligned}$$

The naming convention for the rule is the following: The *1* expresses that we look at one procedure, and *NoRec* that the procedure is non recursive.

**lemma** (*in Square-impl*)

**shows**  $\forall n.\ \Gamma \vdash \langle 'N = n \rangle\ 'R ::= PROC\ Square('N)\ \langle 'R = n * n \rangle$

The directive *in* has the effect that the context of the locale *Square-impl* is included to the current lemma, and that the lemma is added as a fact to the locale, after it

is proven. The next time locale *Square-impl* is invoked this lemma is immediately available as fact, which the verification condition generator can use.

*<proof>*

If the procedure is non recursive and there is no specification given, the verification condition generator automatically expands the body.

**lemma** (in *Square-impl*) *Square-spec*:

**shows**  $\forall n. \Gamma \vdash \{N = n\} \ 'R ::= PROC\ Square('N) \ \{R = n * n\}$

*<proof>*

An important naming convention is to name the specification as *<procedure-name>-spec*. The verification condition generator refers to this name in order to search for a specification in the theorem database.

### 26.2.3 Usage

Let us see how we can use procedure specifications.

**lemma** (in *Square-impl*)

**shows**  $\Gamma \vdash \{I = 2\} \ 'R ::= CALL\ Square('I) \ \{R = 4\}$

Remember that we have already proven *Square-spec* in the locale *Square-impl*. This is crucial for verification condition generation. When reaching a procedure call, it looks for the specification (by its name) and applies the rule *HoarePartial.ProcSpec* instantiated with the specification (as last premise). Before we apply the verification condition generator, let us take some time to think of what we can expect. Let's look at the specification *Square-spec* again:

$\forall n. \Gamma \vdash \{N = n\} \ 'R ::= PROC\ Square('N) \ \{R = n * n\}$

The specification talks about the formal parameters *N* and *R*. The precondition  $\{N = n\}$  just fixes the initial value of *N*. The actual parameters are *I* and *R*. We have to adapt the specification to this calling context.  $\forall n. \Gamma \vdash \{I = n\} \ 'R ::= CALL\ Square() \ \{R = n * n\}$ . From the postcondition  $\{R = n * n\}$  we have to derive the actual postcondition  $\{R = 4\}$ . So we gain something like:  $\{n * n = 4\}$ . The precondition is  $\{I = 2\}$  and the specification tells us  $\{I = n\}$  for the pre-state. So the value of *n* is the value of *I* in the pre-state. So we arrive at  $\{I = 2\} \subseteq \{I * I = 4\}$ .

*<proof>*

The adaption of the procedure specification to the actual calling context is done due to the *init*, *return* and *result* functions in the rule *HoarePartial.ProcSpec* (or in the variant *HoarePartial.ProcSpecNoAbrupt* which already incorporates the fact that the postcondition for abrupt termination is the empty set). For the readers interested in the internals, here a version without vcg.

**lemma** (in *Square-impl*)

**shows**  $\Gamma \vdash \{I = 2\} \ 'R ::= CALL\ Square('I) \ \{R = 4\}$

*<proof>*

### 26.2.4 Recursion

We want to define a procedure for the factorial. We first define a HOL function that calculates it, to specify the procedure later on.

```
primrec fac:: nat ⇒ nat
where
fac 0 = 1 |
fac (Suc n) = (Suc n) * fac n
⟨proof⟩
```

Now we define the procedure.

```
procedures
  Fac (N::nat | R::nat)
  IF 'N = 0 THEN 'R ::= 1
  ELSE 'R ::= CALL Fac('N - 1);;
  'R ::= 'N * 'R
  FI
```

Now let us prove that our implementation of *Fac* meets its specification.

```
lemma (in Fac-impl)
shows ∀ n. Γ ⊢ { 'N = n } 'R ::= PROC Fac('N) { 'R = fac n }
⟨proof⟩
```

Since the factorial is implemented recursively, the main ingredient of this proof is, to assume that the specification holds for the recursive call of *Fac* and prove the body correct. The assumption for recursive calls is added to the context by the rule *HoarePartial.ProcRec1* (also derived from the general rule for mutually recursive procedures):

$$\llbracket \forall Z. \Gamma, \Theta \cup (\bigcup_Z \{(P\ Z, p, Q\ Z, A\ Z)\}) \vdash_{/F} (P\ Z)\ \text{the}\ (\Gamma\ p)\ (Q\ Z), (A\ Z); \\ p \in \text{dom}\ \Gamma \rrbracket \implies \forall Z. \Gamma, \Theta \vdash_{/F} (P\ Z)\ \text{Call}\ p\ (Q\ Z), (A\ Z)$$

The verification condition generator infers the specification out of the context  $\Theta$  when it encounters a recursive call of the factorial.

### 26.3 Global Variables and Heap

Now we define and verify some procedures on heap-lists. We consider list structures consisting of two fields, a content element *cont* and a reference to the next list element *next*. We model this by the following state space where every field has its own heap.

```
hoarestate globals-heap =
  next :: ref ⇒ ref
  cont :: ref ⇒ nat
```

It is mandatory to start the state name with 'globals'. This is exploited by the syntax translations to store the components in the *globals* part of the state.

Updates to global components inside a procedure are always propagated to the caller. This is implicitly done by the parameter passing syntax translations.

We first define an append function on lists. It takes two references as parameters. It appends the list referred to by the first parameter with the list referred to by the second parameter. The statespace of the global variables has to be imported.

```
procedures (imports globals-heap)
  append(p :: ref, q::ref | p::ref)
    IF 'p=Null THEN 'p ::= 'q
    ELSE 'p→'next ::= CALL append('p→'next,'q) FI
```

The difference of a global and a local variable is that global variables are automatically copied back to the procedure caller. We can study this effect on the translation of  $'p ::= CALL\ append()$ :

```
call
(λs. s(locals := locals s(p'append' := locals s·p'append' ,
  q'append' := locals s·q'append')))
append'proc (λs t. s(globals := globals t))
(λi t. 'p ::= locals t·p'append')
```

Below we give two specifications this time. One captures the functional behaviour and focuses on the entities that are potentially modified by the procedure, the second one is a pure frame condition.

The functional specification below introduces two logical variables besides the state space variable  $\sigma$ , namely  $Ps$  and  $Qs$ . They are universally quantified and range over both the pre-and the postcondition, so that we are able to properly instantiate the specification during the proofs. The syntax  $\{\sigma. \dots\}$  is a shorthand to fix the current state:  $\{s. \sigma = s \dots\}$ . Moreover  $\sigma_x$  abbreviates the lookup of variable  $x$  in the state  $\sigma$ .

The approach to specify procedures on lists basically follows [5]. From the pointer structure in the heap we (relationally) abstract to HOL lists of references. Then we can specify further properties on the level of HOL lists, rather than on the heap. The basic abstractions are:

```
Path x h y [] = (x = y)
Path x h y (p · ps) = (x = p ∧ x ≠ Null ∧ Path (h x) h y ps)
```

*Path* (*x*::*ref*) (*h*::*ref* ⇒ *ref*) (*y*::*ref*) (*ps*::*ref list*): *ps* is a list of references that we can obtain out of the heap  $h$  by starting with the reference  $x$ , following the references in  $h$  up to the reference  $y$ .

$List\ p\ h\ ps = Path\ p\ h\ Null\ ps$

A list  $List\ p\ h\ ps$  is a path starting in  $p$  and ending up in  $Null$ .

**lemma** (in *append-impl*) *append-spec1*:

**shows**  $\forall \sigma\ Ps\ Qs.$

$\Gamma \vdash \{ \sigma. List\ 'p\ 'next\ Ps \wedge List\ 'q\ 'next\ Qs \wedge set\ Ps \cap set\ Qs = \{ \} \}$   
 $\quad 'p := PROC\ append('p, 'q)$   
 $\quad \{ List\ 'p\ 'next\ (Ps @ Qs) \wedge (\forall x. x \notin set\ Ps \longrightarrow 'next\ x = \sigma_{next}\ x) \}$   
*<proof>*

If the verification condition generator works on a procedure call it checks whether it can find a modifies clause in the context. If one is present the procedure call is simplified before the Hoare rule *HoarePartial.ProcSpec* is applied. Simplification of the procedure call means that the “copy back” of the global components is simplified. Only those components that occur in the modifies clause are actually copied back. This simplification is justified by the rule *HoarePartial.ProcModifyReturn*. So after this simplification all global components that do not appear in the modifies clause are treated as local variables.

We study the effect of the modifies clause on the following examples, where we want to prove that (@) does not change the *cont* part of the heap.

**lemma** (in *append-impl*)

**shows**  $\Gamma \vdash \{ 'cont=c \} 'p := CALL\ append(Null, Null) \{ 'cont=c \}$   
*<proof>*

We now add the frame condition. The list in the modifies clause names all global state components that may be changed by the procedure. Note that we know from the modifies clause that the *cont* parts are not changed. Also a small side note on the syntax. We use ordinary brackets in the postcondition of the modifies clause, and also the state components do not carry the acute, because we explicitly note the state  $t$  here.

**lemma** (in *append-impl*) *append-modifies*:

**shows**  $\forall \sigma. \Gamma \vdash_{UNIV} \{ \sigma \} 'p := PROC\ append('p, 'q)$   
 $\quad \{ t. t\ may\ only\ modify\ globals\ \sigma\ in\ [next] \}$   
*<proof>*

We tell the verification condition generator to use only the modifies clauses and not to search for functional specifications by the parameter *spec=modifies*. It also tries to solve the verification conditions automatically. Again it is crucial to name the lemma with this naming scheme, since the verification condition generator searches for these names.

The modifies clause is equal to a state update specification of the following form.

**lemma** (in *append-impl*) **shows**  $\{ t. t\ may\ only\ modify\ globals\ Z\ in\ [next] \}$

=  
 $\{t. \exists next. globals t=update id id next-' (K-statefun next) (globals Z)\}$   
 $\langle proof \rangle$

Now that we have proven the frame-condition, it is available within the locale *append-impl* and the *vcg* exploits it.

**lemma** (in *append-impl*)  
**shows**  $\Gamma \vdash \{cont=c\} 'p ::= CALL append(Null,Null) \{cont=c\}$   
 $\langle proof \rangle$

Of course we could add the modifies clause to the functional specification as well. But separating both has the advantage that we split up the verification work. We can make use of the modifies clause before we apply the functional specification in a fully automatic fashion.

To prove that a procedure respects the modifies clause, we only need the modifies clauses of the procedures called in the body. We do not need the functional specifications. So we can always prove the modifies clause without functional specifications, but we may need the modifies clause to prove the functional specifications. So usually the modifies clause is proved before the proof of the functional specification, so that it can already be used by the verification condition generator.

## 26.4 Total Correctness

When proving total correctness the additional proof burden to the user is to come up with a well-founded relation and to prove that certain states get smaller according to this relation. Proving that a relation is well-founded can be quite hard. But fortunately there are ways to construct and stick together relations so that they are well-founded by construction. This infrastructure is already present in Isabelle/HOL. For example, *measure f* is always well-founded; the lexicographic product of two well-founded relations is again well-founded and the inverse image construction *inv-image* of a well-founded relation is again well-founded. The constructions are best explained by some equations:

$$\begin{aligned} ((x, y) \in \text{measure } f) &= (f x < f y) \\ (((a, b), x, y) \in r \text{ <*\textit{lex}*\> } s) &= ((a, x) \in r \vee a = x \wedge (b, y) \in s) \\ ((x, y) \in \text{inv-image } r f) &= ((f x, f y) \in r) \end{aligned}$$

Another useful construction is *<\*\textit{mlex}\*\>* which is a combination of a measure and a lexicographic product:

$$((x, y) \in f \text{ <*\textit{mlex}*\> } r) = (f x < f y \vee f x = f y \wedge (x, y) \in r)$$

In contrast to the lexicographic product it does not construct a product type. The state may either decrease according to the measure function *f* or the measure stays the same and the state decreases because of the relation *r*.

Lets look at a loop:

**lemma** (in vars)  
 $\Gamma \vdash_t \{ \text{'M} = 0 \wedge \text{'S} = 0 \}$   
 WHILE 'M  $\neq$  a  
 INV  $\{ \text{'S} = \text{'M} * b \wedge \text{'M} \leq a \}$   
 VAR MEASURE a - 'M  
 DO 'S ::= 'S + b;; 'M ::= 'M + 1 OD  
 $\{ \text{'S} = a * b \}$   
 <proof>

The variant annotation is preceded by VAR. The capital MEASURE is a shorthand for *measure* ( $\lambda s. a - sM$ ). Analogous there is a capital  $\langle *MLEX* \rangle$ .

**lemma** (in Fac-impl) Fac-spec':  
**shows**  $\forall \sigma. \Gamma \vdash_t \{ \sigma \} \text{'R} ::= \text{PROC Fac}(\text{'N}) \{ \text{'R} = \text{fac } \sigma \text{'N} \}$   
 <proof>

**lemma** (in append-impl) append-spec2:  
**shows**  $\forall \sigma \text{ Ps Qs}. \Gamma \vdash_t$   
 $\{ \sigma. \text{List 'p 'next Ps} \wedge \text{List 'q 'next Qs} \wedge \text{set Ps} \cap \text{set Qs} = \{ \} \}$   
 $\text{'p} ::= \text{PROC append}(\text{'p}, \text{'q})$   
 $\{ \text{List 'p 'next (Ps@Qs)} \wedge (\forall x. x \notin \text{set Ps} \longrightarrow \text{'next } x = \sigma \text{next } x) \}$   
 <proof>

In case of the lists above, we have used a relational list abstraction *List* to construct the HOL lists *Ps* and *Qs* for the pre- and postcondition. To supply a proper measure function we use a functional abstraction *list*. The functional abstraction can be defined by means of the relational list abstraction, since the lists are already uniquely determined by the relational abstraction:

$islist \text{ p h} = (\exists ps. \text{List } \text{p h } ps)$   
 $list \text{ p h} = (\text{THE } ps. \text{List } \text{p h } ps)$

**lemma**  $\text{List } \text{p h } ps = (islist \text{ p h} \wedge ps = list \text{ p h})$

The next contrived example is taken from [3], to illustrate a more complex termination criterion for mutually recursive procedures. The procedures do not calculate anything useful.

**procedures**  
*pedal*(*N::nat*,*M::nat*)  
 IF 0 < 'N THEN  
 IF 0 < 'M THEN  
 CALL coast('N - 1, 'M - 1) FI;;  
 CALL pedal('N - 1, 'M)  
 FI  
**and**  
*coast*(*N::nat*,*M::nat*)  
 CALL pedal('N, 'M);;  
 IF 0 < 'M THEN CALL coast('N, 'M - 1) FI

In the recursive calls in procedure *pedal* the first argument always decreases. In the body of *coast* in the recursive call of *coast* the second argument decreases, but in the call to *pedal* no argument decreases. Therefore an relation only on the state space is insufficient. We have to take the procedure names into account, too. We consider the procedure *coast* to be “bigger” than *pedal* when we construct a well-founded relation on the product of state space and procedure names.

$\langle ML \rangle$

We provide the ML function `gen_proc_rec` to automatically derive a convenient rule for recursion for a given number of mutually recursive procedures.

**lemma** (in *pedal-coast-clique*)

**shows**  $(\forall \sigma. \Gamma \vdash_t \{\sigma\} \text{PROC } \textit{pedal}('N, 'M) \text{UNIV}) \wedge$   
 $(\forall \sigma. \Gamma \vdash_t \{\sigma\} \text{PROC } \textit{coast}('N, 'M) \text{UNIV})$

$\langle \textit{proof} \rangle$

We can achieve the same effect without  $\langle *mlex* \rangle$  by using the ordinary lexicographic product  $\langle *lex* \rangle$ , *inv-image* and *measure*

**lemma** (in *pedal-coast-clique*)

**shows**  $(\forall \sigma. \Gamma \vdash_t \{\sigma\} \text{PROC } \textit{pedal}('N, 'M) \text{UNIV}) \wedge$   
 $(\forall \sigma. \Gamma \vdash_t \{\sigma\} \text{PROC } \textit{coast}('N, 'M) \text{UNIV})$

$\langle \textit{proof} \rangle$

By doing some arithmetic we can express the termination condition with a single measure function.

**lemma** (in *pedal-coast-clique*)

**shows**  $(\forall \sigma. \Gamma \vdash_t \{\sigma\} \text{PROC } \textit{pedal}('N, 'M) \text{UNIV}) \wedge$   
 $(\forall \sigma. \Gamma \vdash_t \{\sigma\} \text{PROC } \textit{coast}('N, 'M) \text{UNIV})$

$\langle \textit{proof} \rangle$

## 26.5 Guards

The purpose of a guard is to guard the **(sub-) expressions** of a statement against runtime faults. Typical runtime faults are array bound violations, dereferencing null pointers or arithmetical overflow. Guards make the potential runtime faults explicit, since the expressions themselves never “fail” because they are ordinary HOL expressions. To relieve the user from typing in lots of standard guards for every subexpression, we supply some input syntax for the common language constructs that automatically generate the guards. For example the guarded assignment  $'M ::=_g ('M + 1) \textit{div} 'N$  gets expanded to guarded command  $(\textit{False}, \{\!\{ \textit{in-range} ('M + 1) \wedge 'N \neq 0 \wedge \textit{in-range} (('M + 1) \textit{div} 'N) \}\!\}) \mapsto 'M ::= ('M + 1) \textit{div} 'N$ . Here *in-range* is uninterpreted by now.

**lemma** (in *vars*)  $\Gamma \vdash \{\!\{ \textit{True} \}\!\} 'M ::=_g ('M + 1) \textit{div} 'N \{\!\{ \textit{True} \}\!\}$

$\langle \textit{proof} \rangle$

The user can supply on (overloaded) definition of *in-range* to fit to his needs. Currently guards are generated for:

- overflow and underflow of numbers (*in-range*). For subtraction of natural numbers  $a - b$  the guard  $b \leq a$  is generated instead of *in-range* to guard against underflows.
- division by 0
- dereferencing of *Null* pointers
- array bound violations

Following (input) variants of guarded statements are available:

- Assignment:  $\dots :=_g \dots$
- If:  $IF_g \dots$
- While:  $WHILE_g \dots$
- Call:  $CALL_g \dots$  or  $\dots := CALL_g \dots$

## 26.6 Miscellaneous Techniques

### 26.6.1 Modifies Clause

We look at some issues regarding the modifies clause with the example of insertion sort for heap lists.

**primrec** *sorted*:: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a list  $\Rightarrow$  bool

**where**

*sorted le* [] = True |

*sorted le* (x#xs) = (( $\forall y \in \text{set } xs. \text{le } x y$ )  $\wedge$  *sorted le* xs)

**procedures** (**imports** *globals-heap*)

*insert*(*r*::ref,*p*::ref | *p*::ref)

IF *r*=Null THEN SKIP

ELSE IF *p*=Null THEN *p* ::= *r*; *p*→*next* ::= Null

ELSE IF *r*→*cont*  $\leq$  *p*→*cont*

THEN *r*→*next* ::= *p*; *p* ::= *r*

ELSE *p*→*next* ::= CALL *insert*(*r*, *p*→*next*)

FI

FI

FI

**lemma** (**in** *insert-impl*) *insert-modifies*:

$\forall \sigma. \Gamma \vdash_{UNIV} \{\sigma\} \text{ 'p ::= PROC insert('r, 'p)$

$\{t. t \text{ may-only-modify-globals } \sigma \text{ in } [next]\}$   
 $\langle proof \rangle$

**lemma** (in *insert-impl*) *insert-spec*:

$\forall \sigma Ps. \Gamma \vdash$   
 $\{\sigma. List \ 'p \ 'next \ Ps \wedge \text{sorted} (\leq) (map \ 'cont \ Ps) \wedge$   
 $\ 'r \neq \text{Null} \wedge \ 'r \notin \text{set } Ps\}$   
 $\ 'p := PROC \ insert(\ 'r, \ 'p)$   
 $\{\exists Qs. List \ 'p \ 'next \ Qs \wedge \text{sorted} (\leq) (map \ \sigma cont \ Qs) \wedge$   
 $\ \text{set } Qs = \text{insert} \ \sigma_r (\text{set } Ps) \wedge$   
 $\ (\forall x. x \notin \text{set } Qs \longrightarrow \ 'next \ x = \sigma_{next} \ x)\}$   
 $\langle proof \rangle$

In the postcondition of the functional specification there is a small but important subtlety. Whenever we talk about the *cont* part we refer to the one of the pre-state. The reason is that we have separated out the information that *cont* is not modified by the procedure, to the modifies clause. So whenever we talk about unmodified parts in the postcondition we have to use the pre-state part, or explicitly state an equality in the postcondition. The reason is simple. If the postcondition would talk about *'cont* instead of  $\sigma cont$ , we get a new instance of *cont* during verification and the postcondition would only state something about this new instance. But as the verification condition generator uses the modifies clause the caller of *insert* instead still has the old *cont* after the call. That's the sense of the modifies clause. So the caller and the specification simply talk about two different things, without being able to relate them (unless an explicit equality is added to the specification).

## 26.6.2 Annotations

Annotations (like loop invariants) are mere syntactic sugar of statements that are used by the *vcg*. Logically a statement with an annotation is equal to the statement without it. Hence annotations can be introduced by the user while building a proof:

$$HoarePartial.annotateI: \frac{\Gamma, \Theta \vdash_{/F} P \text{ anno } Q, A \quad c = \text{anno}}{\Gamma, \Theta \vdash_{/F} P \ c \ Q, A}$$

When introducing annotations it can easily happen that these mess around with the nesting of sequential composition. Then after stripping the annotations the resulting statement is no longer syntactically identical to original one, only equivalent modulo associativity of sequential composition. The following rule also deals with this case:

$$HoarePartial.annotate-normalI: \frac{\Gamma, \Theta \vdash_{/F} P \text{ anno } Q, A \quad \text{Language.normalize } c = \text{Language.normalize anno}}{\Gamma, \Theta \vdash_{/F} P \ c \ Q, A}$$

## Loop Annotations

```

procedures (imports globals-heap)
  insertSort(p::ref | p::ref)
  where r::ref q::ref in
    'r ::= Null;;
    WHILE ('p ≠ Null) DO
      'q ::= 'p;;
      'p ::= 'p → 'next;;
      'r ::= CALL insert('q, 'r)
    OD;;
    'p ::= 'r

```

**lemma** (**in** *insertSort-impl*) *insertSort-modifies*:

**shows**

$$\forall \sigma. \Gamma \vdash /UNIV \{ \sigma \} \ 'p ::= PROC \ insertSort('p$$

$$\{ t. t \text{ may-only-modify-globals } \sigma \text{ in } [next] \}$$

*<proof>*

Insertion sort is not implemented recursively here, but with a loop. Note that the while loop is not annotated with an invariant in the procedure definition. The invariant only comes into play during verification. Therefore we annotate the loop first, before we run the *vcg*.

**lemma** (**in** *insertSort-impl*) *insertSort-spec*:

**shows**  $\forall \sigma \ Ps.$

$$\Gamma \vdash \{ \sigma. List \ 'p \ 'next \ Ps \}$$

$$'p ::= PROC \ insertSort('p$$

$$\{ \exists Qs. List \ 'p \ 'next \ Qs \wedge sorted \ (\leq) \ (map \ \sigma cont \ Qs) \wedge$$

$$set \ Qs = set \ Ps \}$$

*<proof>*

The method *hoare-rule* automatically solves the side-condition that the annotated program is the same as the original one after stripping the annotations.

## Specification Annotations

When verifying a larger block of program text, it might be useful to split up the block and to prove the parts in isolation. This is especially useful to isolate loops. On the level of the Hoare calculus the parts can then be combined with the consequence rule. To automate this process we introduce the derived command *specAnno*, which allows to introduce a Hoare tuple (inclusive auxiliary variables) in the program text:

$$specAnno \ P \ c \ Q \ A = c \ undefined$$

The whole annotation reduces to the body *c undefined*. The type of the assertions *P*, *Q* and *A* is '*a* ⇒ '*s* set and the type of command *c* is '*a* ⇒ ('*s*,

$\langle p, \langle f \rangle \text{ com}$ . All entities formally depend on an auxiliary (logical) variable of type  $\langle a$ . The body  $c$  formally also depends on this variable, since a nested annotation or loop invariant may also depend on this logical variable. But the raw body without annotations does not depend on the logical variable. The logical variable is only used by the verification condition generator. We express this by defining the whole *specAnno* to be equivalent with the body applied to an arbitrary variable.

The Hoare rule for *specAnno* is mainly an instance of the consequence rule:  

$$\llbracket [P \subseteq \{s \mid \exists Z. s \in P' Z \wedge Q' Z \subseteq Q \wedge A' Z \subseteq A\}; \forall Z. \Gamma, \Theta \vdash_{/F} (P' Z) c Z (Q' Z), (A' Z); \forall Z. c Z = c \text{ undefined}] \implies \Gamma, \Theta \vdash_{/F} P \text{ specAnno } P' c Q' A' Q, A$$

The side-condition  $\forall Z. c Z = c \text{ undefined}$  expresses the intention of body  $c$  explained above: The raw body is independent of the auxiliary variable. This side-condition is solved automatically by the *vcg*. The concrete syntax for this specification annotation is shown in the following example:

**lemma** (in vars)  $\Gamma \vdash \{\sigma\}$   

$$\begin{aligned} & \langle I := \langle M \rangle;; \\ & \text{ANNO } \tau. \{ \langle \tau. \langle I = \langle \sigma M \rangle \rangle \\ & \quad \langle M := \langle N \rangle;; \langle N := \langle I \rangle \\ & \quad \{ \langle M = \langle \tau N \rangle \wedge \langle N = \langle \tau I \rangle \} \\ & \{ \langle M = \langle \sigma N \rangle \wedge \langle N = \langle \sigma M \rangle \} \end{aligned}$$

With the annotation we can name an intermediate state  $\tau$ . Since the postcondition refers to  $\sigma$  we have to link the information about the equivalence of  $\langle \tau I$  and  $\langle \sigma M$  in the specification in order to be able to derive the postcondition.

$\langle \text{proof} \rangle$

**lemma** (in vars)  

$$\begin{aligned} & \Gamma \vdash \{\sigma\} \\ & \langle I := \langle M \rangle;; \\ & \text{ANNO } \tau. \{ \langle \tau. \langle I = \langle \sigma M \rangle \rangle \\ & \quad \langle M := \langle N \rangle;; \langle N := \langle I \rangle \\ & \quad \{ \langle M = \langle \tau N \rangle \wedge \langle N = \langle \tau I \rangle \} \\ & \{ \langle M = \langle \sigma N \rangle \wedge \langle N = \langle \sigma M \rangle \} \end{aligned}$$

$\langle \text{proof} \rangle$

Note that *vcg-step* changes the order of sequential composition, to allow the user to decompose sequences by repeated calls to *vcg-step*, whereas *vcg* preserves the order.

The above example illustrates how we can introduce a new logical state variable  $\tau$ . You can introduce multiple variables by using a tuple:

**lemma** (in vars)  

$$\begin{aligned} & \Gamma \vdash \{\sigma\} \\ & \langle I := \langle M \rangle;; \end{aligned}$$

$$\begin{array}{l}
\text{ANNO } (n, i, m). \{ \text{'I} = {}^\sigma M \wedge \text{'N} = n \wedge \text{'I} = i \wedge \text{'M} = m \} \\
\text{'M} ::= \text{'N}; \text{'N} ::= \text{'I} \\
\{ \text{'M} = n \wedge \text{'N} = i \} \\
\{ \text{'M} = {}^\sigma N \wedge \text{'N} = {}^\sigma M \} \\
\langle \text{proof} \rangle
\end{array}$$

## Lemma Annotations

The specification annotations described before split the verification into several Hoare triples which result in several subgoals. If we instead want to proof the Hoare triples independently as separate lemmas we can use the *LEMMA* annotation to plug together the lemmas. It inserts the lemma in the same fashion as the specification annotation.

**lemma** (in vars) *foo-lemma*:

$$\begin{array}{l}
\forall n \ m. \Gamma \vdash \{ \text{'N} = n \wedge \text{'M} = m \} \text{'N} ::= \text{'N} + 1;; \text{'M} ::= \text{'M} + 1 \\
\{ \text{'N} = n + 1 \wedge \text{'M} = m + 1 \} \\
\langle \text{proof} \rangle
\end{array}$$

**lemma** (in vars)

$$\begin{array}{l}
\Gamma \vdash \{ \text{'N} = n \wedge \text{'M} = m \} \\
\text{LEMMA } \text{foo-lemma} \\
\text{'N} ::= \text{'N} + 1;; \text{'M} ::= \text{'M} + 1 \\
\text{END}; \\
\text{'N} ::= \text{'N} + 1 \\
\{ \text{'N} = n + 2 \wedge \text{'M} = m + 1 \} \\
\langle \text{proof} \rangle
\end{array}$$

**lemma** (in vars)

$$\begin{array}{l}
\Gamma \vdash \{ \text{'N} = n \wedge \text{'M} = m \} \\
\text{LEMMA } \text{foo-lemma} \\
\text{'N} ::= \text{'N} + 1;; \text{'M} ::= \text{'M} + 1 \\
\text{END}; \\
\text{LEMMA } \text{foo-lemma} \\
\text{'N} ::= \text{'N} + 1;; \text{'M} ::= \text{'M} + 1 \\
\text{END} \\
\{ \text{'N} = n + 2 \wedge \text{'M} = m + 2 \} \\
\langle \text{proof} \rangle
\end{array}$$

**lemma** (in vars)

$$\begin{array}{l}
\Gamma \vdash \{ \text{'N} = n \wedge \text{'M} = m \} \\
\text{'N} ::= \text{'N} + 1;; \text{'M} ::= \text{'M} + 1;; \\
\text{'N} ::= \text{'N} + 1;; \text{'M} ::= \text{'M} + 1 \\
\{ \text{'N} = n + 2 \wedge \text{'M} = m + 2 \} \\
\langle \text{proof} \rangle
\end{array}$$

### 26.6.3 Total Correctness of Nested Loops

When proving termination of nested loops it is sometimes necessary to express that the loop variable of the outer loop is not modified in the inner loop. To express this one has to fix the value of the outer loop variable before the inner loop and use this value in the invariant of the inner loop. This can be achieved by surrounding the inner while loop with an *ANNO* specification as explained previously. However, this leads to repeating the invariant of the inner loop three times: in the invariant itself and in the the pre- and postcondition of the *ANNO* specification. Moreover one has to deal with the additional subgoal introduced by *ANNO* that expresses how the pre- and postcondition is connected to the invariant. To avoid this extra specification and verification work, we introduce an variant of the annotated while-loop, where one can introduce logical variables by *FIX*. As for the *ANNO* specification multiple logical variables can be introduced via a tuple (*FIX* (*a,b,c*)).

The Hoare logic rule for the augmented while-loop is a mixture of the invariant rule for loops and the consequence rule for *ANNO*:

$$\begin{aligned} \llbracket P \subseteq \{s \mid \exists Z. s \in IZ \wedge (\forall t. t \in IZ \cap \neg b \longrightarrow t \in Q)\}; \forall Z \sigma. \Gamma, \Theta \vdash_{t/F} \\ (\{\sigma\} \cap IZ \cap b) \ c \ Z \ (\{t \mid (t, \sigma) \in VZ\} \cap IZ), A; \forall Z. c \ Z = c \ \text{undefined}; \\ \forall Z. wf \ (VZ) \rrbracket \Longrightarrow \Gamma, \Theta \vdash_{t/F} P \ \text{whileAnnoFix } b \ I \ V \ c \ Q, A \end{aligned}$$

The first premise expresses that the precondition implies the invariant and that the invariant together with the negated loop condition implies the postcondition. Since both implications may depend on the choice of the auxiliary variable *Z* these two implications are expressed in a single premise and not in two of them as for the usual while rule. The second premise is the preservation of the invariant by the loop body. And the third premise is the side-condition that the computational part of the body does not depend on the auxiliary variable. Finally the last premise is the well-foundedness of the variant. The last two premises are usually discharged automatically by the verification condition generator. Hence usually two subgoals remain for the user, stemming from the first two premises.

The following example illustrates the usage of this rule. The outer loop increments the loop variable *M* while the inner loop increments *N*. To discharge the proof obligation for the termination of the outer loop, we need to know that the inner loop does not mess around with *M*. This is expressed by introducing the logical variable *m* and fixing the value of *M* to it.

**lemma** (in *vars*)

$$\begin{aligned} \Gamma \vdash_t \{ \prime M = 0 \wedge \prime N = 0 \} \\ \text{WHILE } (\prime M < i) \\ \text{INV } \{ \prime M \leq i \wedge (\prime M \neq 0 \longrightarrow \prime N = j) \wedge \prime N \leq j \} \\ \text{VAR MEASURE } (i - \prime M) \end{aligned}$$

```

DO
  'N ::= 0;;
  WHILE ('N < j)
  FIX m.
  INV { 'M=m ∧ 'N ≤ j }
  VAR MEASURE (j - 'N)
  DO
    'N ::= 'N + 1
  OD;;
  'M ::= 'M + 1
  OD
  { 'M=i ∧ ('M≠0 → 'N=j) }
⟨proof⟩

```

## 26.7 Functional Correctness, Termination and Runtime Faults

Total correctness of a program with guards conceptually leads to three verification tasks.

- functional (partial) correctness
- absence of runtime faults
- termination

In case of a modifies specification the functional correctness part can be solved automatically. But the absence of runtime faults and termination may be non trivial. Fortunately the modifies clause is usually just a helpful companion of another specification that expresses the “real” functional behaviour. Therefor the task to prove the absence of runtime faults and termination can be dealt with during the proof of this functional specification. In most cases the absence of runtime faults and termination heavily build on the functional specification parts. So after all there is no reason why we should again prove the absence of runtime faults and termination for the modifies clause. Therefor it suffices to have partial correctness of the modifies clause for a program were all guards are ignored. This leads to the following pattern:

**procedures** *foo* ( $N::nat|M::nat$ )

'M ::= 'M  
 — think of body with guards instead

*foo-spec*:  $\forall \sigma. \Gamma \vdash_t (P \ \sigma) \ 'M ::= PROC \ foo('N) \ (Q \ \sigma)$   
*foo-modifies*:  $\forall \sigma. \Gamma \vdash_{UNIV} \{ \sigma \} \ 'M ::= PROC \ foo('N)$   
 $\{ t. t \ may\ only\ modify\ globals \ \sigma \ in \ [] \}$

The verification condition generator can solve those modifies clauses automatically and can use them to simplify calls to *foo* even in the context of total correctness.

## 26.8 Procedures and Locales

Verification of a larger program is organised on the granularity of procedures. We proof the procedures in a bottom up fashion. Of course you can also always use Isabelle’s dummy proof *sorry* to prototype your formalisation. So you can write the theory in a bottom up fashion but actually prove the lemmas in any other order.

Here are some explanations of handling of locales. In the examples below, consider *proc<sub>1</sub>* and *proc<sub>2</sub>* to be “leaf” procedures, which do not call any other procedure. Procedure *proc* directly calls *proc<sub>1</sub>* and *proc<sub>2</sub>*.

**lemma** (in *proc<sub>1</sub>-impl*) *proc<sub>1</sub>-modifies*:  
**shows** ...

After the proof of *proc<sub>1</sub>-modifies*, the **in** directive stores the lemma in the locale *proc<sub>1</sub>-impl*. When we later on include *proc<sub>1</sub>-impl* or prove another theorem in locale *proc<sub>1</sub>-impl* the lemma *proc<sub>1</sub>-modifies* will already be available as fact.

**lemma** (in *proc<sub>1</sub>-impl*) *proc<sub>1</sub>-spec*:  
**shows** ...

**lemma** (in *proc<sub>2</sub>-impl*) *proc<sub>2</sub>-modifies*:  
**shows** ...

**lemma** (in *proc<sub>2</sub>-impl*) *proc<sub>2</sub>-spec*:  
**shows** ...

**lemma** (in *proc-impl*) *proc-modifies*:  
**shows** ...

Note that we do not explicitly include anything about *proc<sub>1</sub>* or *proc<sub>2</sub>* here. This is handled automatically. When defining an *impl*-locale it imports all *impl*-locales of procedures that are called in the body. In case of *proc-impl* this means, that *proc<sub>1</sub>-impl* and *proc<sub>2</sub>-impl* are imported. This has the neat effect that all theorems that are proven in *proc<sub>1</sub>-impl* and *proc<sub>2</sub>-impl* are also present in *proc-impl*.

**lemma** (in *proc-impl*) *proc-spec*:  
**shows** ...

As we have seen in this example you only have to prove a procedure in its own *impl* locale. You do not have to include any other locale.

## 26.9 Records

Before *statespaces* where introduced the state was represented as a *record*. This is still supported. Compared to the flexibility of *statespaces* there are some drawbacks in particular with respect to modularity. Even names of local variables and parameters are globally visible and records can only be extended in a linear fashion, whereas *statespaces* also allow multiple inher-

itance. The usage of records is quite similar to the usage of statespaces. We repeat the example of an append function for heap lists. First we define the global components. Again the appearance of the prefix ‘globals’ is mandatory. This is the way the syntax layer distinguishes local and global variables.

```
record globals-list =
  next' :: ref ⇒ ref
  cont' :: ref ⇒ nat
```

The local variables also have to be defined as a record before the actual definition of the procedure. The parent record *state* defines a generic *globals* field as a place-holder for the record of global components. In contrast to the statespace approach there is no single *locals* slot. The local components are just added to the record.

```
record 'g list-vars = 'g state +
  p' :: ref
  q' :: ref
  r' :: ref
  root' :: ref
  tmp' :: ref
```

Since the parameters and local variables are determined by the record, there are no type annotations or definitions of local variables while defining a procedure.

**procedures**

```
append'(p,q|p) =
  IF 'p=Null THEN 'p ::= 'q
  ELSE 'p → 'next ::= CALL append'('p → 'next, 'q) FI
```

As in the statespace approach, a locale called *append'-impl* is created. Note that we do not give any explicit information which global or local state-record to use. Since the records are already defined we rely on Isabelle’s type inference. Dealing with the locale is analogous to the case with statespaces.

**lemma** (in *append'-impl*) *append'-modifies*:

```
shows
  ∀σ. Γ ⊢ {σ} 'p ::= PROC append'('p, 'q)
  {t. t may-only-modify-globals σ in [next]}
  ⟨proof⟩
```

**lemma** (in *append'-impl*) *append'-spec*:

```
shows ∀σ Ps Qs. Γ ⊢
  {σ. List 'p 'next Ps ∧ List 'q 'next Qs ∧ set Ps ∩ set Qs = {}}
  'p ::= PROC append'('p, 'q)
  {List 'p 'next (Ps@Qs) ∧ (∀x. x ∉ set Ps → 'next x = σnext x)}
  ⟨proof⟩
```

However, in some corner cases the inferred state type in a procedure definition can be too general which raises problems when attempting to proof a suitable specifications in the locale. Consider for example the simple procedure body  $\acute{p} ::= NULL$  for a procedure *init*.

```
procedures init (|p) =
   $\acute{p} ::= Null$ 
```

Here Isabelle can only infer the local variable record. Since no reference to any global variable is made the type fixed for the global variables (in the locale *init'-impl*) is a type variable say  $\acute{g}$  and not a *globals-list* record. Any specification mentioning *next* or *cont* restricts the state type and cannot be added to the locale *init'-impl*. Hence we have to restrict the body  $\acute{p} ::= NULL$  in the first place by adding a typing annotation:

```
procedures init' (|p) =
   $\acute{p} ::= Null :: (('a\ globals-list-scheme, 'b) list-vars-scheme, char list, 'c) com$ 
```

### 26.9.1 Extending State Spaces

The records in Isabelle are extensible [7, 6]. In principle this can be exploited during verification. The state space can be extended while we add procedures. But there is one major drawback:

- records can only be extended in a linear fashion (there is no multiple inheritance)

You can extend both the main state record as well as the record for the global variables.

### 26.9.2 Mapping Variables to Record Fields

Generally the state space (global and local variables) is flat and all components are accessible from everywhere. Locality or globality of variables is achieved by the proper *init* and *return/result* functions in procedure calls. What is the best way to map programming language variables to the state records? One way is to disambiguate all names, by using the procedure names as prefix or the structure names for heap components. This leads to long names and lots of record components. But for local variables this is not necessary, since variable *i* of procedure *A* and variable *i* of procedure *B* can be mapped to the same record component, without any harm, provided they have the same logical type. Therefor for local variables it is preferable to map them per type. You only have to distinguish a variable with the same name if they have a different type. Note that all pointers just have logical type *ref*. So you even do not have to distinguish between a pointer *p* to a integer and a pointer *p* to a list. For global components (global variables

and heap structures) you have to disambiguate the name. But hopefully the field names of structures have different names anyway. Also note that there is no notion of hiding of a global component by a local one in the logic. You have to disambiguate global and local names! As the names of the components show up in the specifications and the proof obligations, names are even more important as for programming. Try to find meaningful and short names, to avoid cluttering up your reasoning.

## References

- [1] E. Alkassar, N. Schirmer, and A. Starostin. Formal pervasive verification of a paging mechanism. In *14th intl Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS08), to appear*, LNCS. Springer, 2008.
- [2] C. Ballarin. Locales and locale expressions in Isabelle/Isar. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs: International Workshop, TYPES 2003, Torino, Italy, April 30–May 4, 2003, Selected Papers*, number 3085 in LNCS, pages 34–50. Springer, 2004.
- [3] P. V. Homeier. *Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures*. PhD thesis, Department of Computer Science, University of California, Los Angeles, 1995.
- [4] D. Leinenbach and E. Petrova. Pervasive compiler verification – from verified programs to verified systems. In *3rd intl Workshop on Systems Software Verification (SSV08), to appear*. Elsevier Science B. V., 2008.
- [5] F. Mehta and T. 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.
- [6] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs’98*, volume 1479 of LNCS. Springer, 1998.
- [7] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of LNCS. Springer, 2002. <http://www.in.tum.de/~nipkow/LNCS2283/>.

- [8] V. Ortner and N. Schirmer. Verification of BDD normalization. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 2005*, volume 3603 of *LNCS*, pages 261–277. Springer, 2005.
- [9] N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006. Available from <http://mediatum2.ub.tum.de/doc/601799/601799.pdf>.
- [10] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 97–108, New York, NY, USA, 2007. ACM Press.