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.

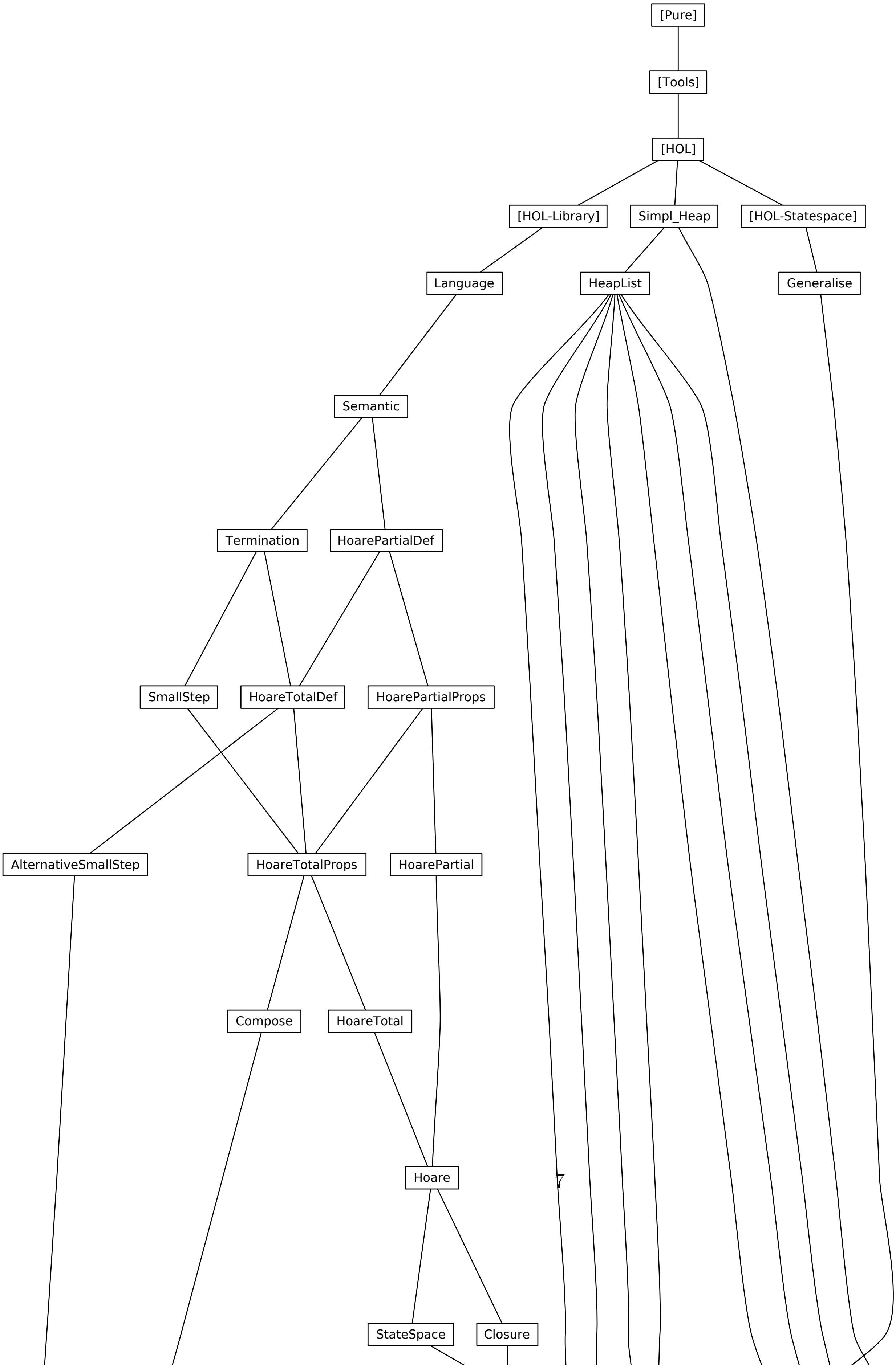
— Simpl —

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

Norbert W. Schirmer

March 19, 2025

Contents


[Pure]
[Tools]
[HOL]
[HOL-Library]
Simpl_Heap
[HOL-Statespace]
Language
HeapList
Generalise
Semantic
Termination
HoarePartialDef
SmallStep
HoareTotalDef
HoarePartialProps
AlternativeSmallStep
HoareTotalProps
HoarePartial
Compose
HoareTotal
Hoare
StateSpace
Closure

1 Introduction

The work presented in these theories was developed within the German Verisoft project[1]. 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 ⇒ 's*
  | *Spec ('s × '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 ⇒ ('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 ⇒ 's*) ⇒ (*'s,'p,'f*) *com* **where**
  raise f = *Seq* (*Basic f*) *Throw*

definition
  condCatch:: (*'s,'p,'f*) *com* ⇒ *'s bexp* ⇒ (*'s,'p,'f*) *com* ⇒ (*'s,'p,'f*) *com* **where**
  condCatch c_1 *b* c_2 = *Catch* c_1 (*Cond b* c_2 *Throw*)

definition
  bind:: (*'s ⇒ 'v*) ⇒ (*'v ⇒* (*'s,'p,'f*) *com*) ⇒ (*'s,'p,'f*) *com* **where**
  bind e c = *DynCom* (λ*s. c* (*e s*))

[1] http://www.verisoft.de

definition
bseq:: $('s,'p,'f)$ *com* ⇒ $('s,'p,'f)$ *com* ⇒ $('s,'p,'f)$ *com* **where**
bseq = *Seq*

definition
block-exn:: $['s\Rightarrow's,('s,'p,'f)$ *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-exn init bdy return result-exn c =
DynCom (λ*s*. (*Seq* (*Catch* (*Seq* (*Basic init*) *bdy*) (*Seq* (*Basic* (λ*t*. *result-exn* (*return s t*) *t*)) *Throw*))
(*DynCom* (λ*t*. *Seq* (*Basic* (*return s*)) (*c s t*))))
)

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

primrec *guards*:: $('f \times 's$ *set* $)$ *list* ⇒ $('s,'p,'f)$ *com* ⇒ $('s,'p,'f)$ *com*
where
guards [] *c* = *c* |
guards (*g*#*gs*) *c* = *Guard* (*fst g*) (*snd g*) (*guards gs c*)

definition *maybe-guard*:: $'f \Rightarrow 's$ *set* ⇒ $('s,'p,'f)$ *com* ⇒ $('s,'p,'f)$ *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-exn:: $'f \Rightarrow 's$ *set* ⇒ $('s \Rightarrow 's) \Rightarrow ('s \Rightarrow 'p) \Rightarrow$
$('s \Rightarrow 's \Rightarrow 's) \Rightarrow ('s \Rightarrow 's \Rightarrow 's) \Rightarrow ('s \Rightarrow 's \Rightarrow ('s,'p,'f)$ *com*$) \Rightarrow ('s,'p,'f)$ *com* **where**
dynCall-exn f g init p return result-exn c =
maybe-guard f g (*DynCom* (λ*s*. *call-exn init* (*p s*) *return result-exn 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-exn init bdy return* (λ*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

$specAnno :: ('a \Rightarrow 's\ assn) \Rightarrow ('a \Rightarrow ('s,'p,'f)\ com) \Rightarrow$
$('a \Rightarrow 's\ assn) \Rightarrow ('a \Rightarrow 's\ assn) \Rightarrow ('s,'p,'f)\ com$
where $specAnno\ P\ c\ Q\ A = (c\ undefined)$

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

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

definition $if\text{-}rel :: ('s \Rightarrow bool) \Rightarrow ('s \Rightarrow 's) \Rightarrow ('s \Rightarrow 's) \Rightarrow ('s \Rightarrow 's) \Rightarrow ('s \times 's)\ set$
where $if\text{-}rel\ b\ f\ g\ h = \{(s,t).\ if\ b\ s\ then\ t = f\ s\ else\ t = g\ s \lor t = h\ s\}$

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

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

lemma *call-call-exn*: $call\ init\ p\ return\ result = call\text{-}exn\ init\ p\ return\ (\lambda s\ t.\ s)\ result$
⟨*proof*⟩

lemma *dynCall-dynCall-exn*: $dynCall\ init\ p\ return\ result = dynCall\text{-}exn\ undefined\ UNIV\ init\ p\ return\ (\lambda s\ t.\ s)\ result$
⟨*proof*⟩

2.3 Operations on Simpl-Syntax

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

primrec $flatten :: ('s,'p,'f)\ com \Rightarrow ('s,'p,'f)\ com\ list$
where
$flatten\ Skip = [Skip]\ |$
$flatten\ (Basic\ f) = [Basic\ f]\ |$
$flatten\ (Spec\ r) = [Spec\ r]\ |$
$flatten\ (Seq\ c_1\ c_2) = flatten\ c_1\ @\ flatten\ c_2\ |$
$flatten\ (Cond\ b\ c_1\ c_2) = [Cond\ b\ c_1\ c_2]\ |$
$flatten\ (While\ b\ c) = [While\ b\ c]\ |$
$flatten\ (Call\ p) = [Call\ p]\ |$
$flatten\ (DynCom\ c) = [DynCom\ c]\ |$
$flatten\ (Guard\ f\ g\ c) = [Guard\ f\ g\ c]\ |$
$flatten\ Throw = [Throw]\ |$

flatten (*Catch* c_1 c_2) = [*Catch* c_1 c_2]

primrec *sequence*:: (($'s$,$'p$,$'f$) *com* ⇒ ($'s$,$'p$,$'f$) *com* ⇒ ($'s$,$'p$,$'f$) *com*) ⇒
($'s$,$'p$,$'f$) *com list* ⇒ ($'s$,$'p$,$'f$) *com*
where
sequence seq [] = *Skip* |
sequence seq (*c*#*cs*) = (*case cs of* [] ⇒ *c*
| - ⇒ *seq c* (*sequence seq cs*))

primrec *normalize*:: ($'s$,$'p$,$'f$) *com* ⇒ ($'s$,$'p$,$'f$) *com*
where
normalize Skip = *Skip* |
normalize (*Basic f*) = *Basic f* |
normalize (*Spec r*) = *Spec r* |
normalize (*Seq* c_1 c_2) = *sequence Seq*
((*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* (λ*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* ≠ []
⟨*proof*⟩

lemma *flatten-single*: ∀ *c* ∈ *set* (*flatten c'*). *flatten c* = [*c*]
⟨*proof*⟩

lemma *flatten-sequence-id*:
⟦*cs*≠[];∀ *c* ∈ *set cs*. *flatten c* = [*c*]⟧ ⟹ *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*: $\bigwedge x\ xs.$ *flatten* (*normalize c*) = *x*#*xs*
⟹ (*case xs of* [] ⇒ *normalize c* = *x*
| (*x'*#*xs'*) ⇒ *normalize c*= *Seq x* (*sequence Seq xs*))
⟨*proof*⟩

lemma *flatten-raise* [*simp*]: *flatten* (*raise f*) = [*Basic f*, *Throw*]
⟨*proof*⟩

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

lemma *flatten-bind* [*simp*]: *flatten* (*bind e c*) = [*bind e c*]
⟨*proof*⟩

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

lemma *flatten-block-exn* [*simp*]:
flatten (*block-exn init bdy return result-exn result*) = [*block-exn init bdy return result-exn result*]
⟨*proof*⟩

lemma *flatten-block* [*simp*]:
flatten (*block init bdy return result*) = [*block init bdy return result*]
⟨*proof*⟩

lemma *flatten-call* [*simp*]: *flatten* (*call init p return result*) = [*call init p return result*]
⟨*proof*⟩

lemma *flatten-dynCall* [*simp*]: *flatten* (*dynCall init p return result*) = [*dynCall init p return result*]
⟨*proof*⟩

lemma *flatten-call-exn* [*simp*]: *flatten* (*call-exn init p return result-exn result*) = [*call-exn init p return result-exn result*]
⟨*proof*⟩

lemma *flatten-dynCall-exn* [*simp*]: *flatten* (*dynCall-exn f g init p return result-exn result*) = [*dynCall-exn f g init p return result-exn result*]
⟨*proof*⟩

lemma *flatten-fcall* [*simp*]: *flatten* (*fcall init p return result c*) = [*fcall init p return 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* (λ*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* (λ*s t*. *normalize* (*c s t*))

⟨*proof*⟩

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

lemma *normalize-call* [*simp*]:
normalize (*call init p return c*) = *call init p return* (*λi 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* (*λi t*. *normalize* (*c i t*))
⟨*proof*⟩

lemma *normalize-dynCall* [*simp*]:
normalize (*dynCall init p return c*) =
dynCall init p return (*λs 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 (*λs t*. *normalize* (*c s t*))
⟨*proof*⟩

lemma *normalize-fcall* [*simp*]:
normalize (*fcall init p return result c*) =
fcall init p return result (*λv*. *normalize* (*c v*))
⟨*proof*⟩

lemma *normalize-switch* [*simp*]:
normalize (*switch v Vcs*) = *switch v* (*map* (*λ*(*V*,*c*). (*V*,*normalize c*)) *Vcs*)
⟨*proof*⟩

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

Sequencial 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_1 c_2) = (*Seq* (*strip-guards F* c_1) (*strip-guards F* c_2)) |
strip-guards F (*Cond b* c_1 c_2) = *Cond b* (*strip-guards F* c_1) (*strip-guards F* c_2) |
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_1 c_2) = *Catch* (*strip-guards F* c_1) (*strip-guards F* c_2)

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* Γ) = *strip F* Γ
⟨*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* (λ*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* (λ*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* (λ*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 (λ*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 (λ*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 (λ*s t*. *strip-guards F* (*c s t*))
⟨*proof*⟩

lemma *strip-guards-guards* [*simp*]: *strip-guards F* (*guards gs c*) =
guards (*filter* (λ(*f*,*g*). *f* ∉ *F*) *gs*) (*strip-guards F c*)

⟨*proof*⟩

lemma *strip-guards-dynCall-exn* [*simp*]:
strip-guards F (*dynCall-exn f g init p return result-exn c*) =
dynCall-exn f (*if f* ∈ *F then UNIV else g*) *init p return result-exn* (λ*s t*. *strip-guards F* (*c s t*))
⟨*proof*⟩

lemma *strip-guards-fcall* [*simp*]:
strip-guards F (*fcall init p return result c*) =
fcall init p return result (λ*v*. *strip-guards F* (*c v*))
⟨*proof*⟩

lemma *strip-guards-switch* [*simp*]:
strip-guards F (*switch v Vc*) =
switch v (*map* (λ(*V*,*c*). (*V*,*strip-guards F c*)) *Vc*)
⟨*proof*⟩

lemma *strip-guards-guaranteeStrip* [*simp*]:
strip-guards F (*guaranteeStrip f g c*) =
(*if f* ∈ *F then strip-guards F c*
else guaranteeStrip f g (*strip-guards F c*))
⟨*proof*⟩

lemma *guaranteeStripPair-split-conv* [*simp*]: *case-prod c* (*guaranteeStripPair f g*) = *c f g*
⟨*proof*⟩

lemma *strip-guards-while* [*simp*]:
strip-guards F (*while gs b c*) =
while (*filter* (λ(*f*,*g*). *f* ∉ *F*) *gs*) *b* (*strip-guards F c*)
⟨*proof*⟩

lemma *strip-guards-whileAnno* [*simp*]:
strip-guards F (*whileAnno b I V c*) = *whileAnno b I V* (*strip-guards F c*)
⟨*proof*⟩

lemma *strip-guards-whileAnnoG* [*simp*]:
strip-guards F (*whileAnnoG gs b I V c*) =
whileAnnoG (*filter* (λ(*f*,*g*). *f* ∉ *F*) *gs*) *b I V* (*strip-guards F c*)
⟨*proof*⟩

lemma *strip-guards-specAnno* [*simp*]:
strip-guards F (*specAnno P c Q A*) =
specAnno P (λ*s*. *strip-guards F* (*c undefined*)) *Q A*
⟨*proof*⟩

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*:: $'f \Rightarrow ('s,'p,'g)\ com \Rightarrow ('s,'p,'f)\ com$
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_1 c_2) = (*Seq* (*mark-guards f* c_1) (*mark-guards f* c_2)) |
mark-guards f (*Cond b* c_1 c_2) = *Cond b* (*mark-guards f* c_1) (*mark-guards f* c_2) |
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* ($\lambda s.$ (*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_1 c_2) = *Catch* (*mark-guards f* c_1) (*mark-guards f* c_2)

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* ($\lambda v.$ *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* ($\lambda s\ t.$ *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* ($\lambda s\ t.$ *mark-guards f* (*c s t*))
⟨*proof*⟩

lemma *mark-guards-call* [*simp*]:
$mark\text{-}guards\ f\ (call\ init\ p\ return\ c) =$
$call\ init\ p\ return\ (\lambda s\ t.\ mark\text{-}guards\ f\ (c\ s\ t))$
⟨*proof*⟩

lemma *mark-guards-call-exn* [*simp*]:
$mark\text{-}guards\ f\ (call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c) =$
$call\text{-}exn\ init\ p\ return\ result\text{-}exn\ (\lambda s\ t.\ mark\text{-}guards\ f\ (c\ s\ t))$
⟨*proof*⟩

lemma *mark-guards-dynCall* [*simp*]:
$mark\text{-}guards\ f\ (dynCall\ init\ p\ return\ c) =$
$dynCall\ init\ p\ return\ (\lambda s\ t.\ mark\text{-}guards\ f\ (c\ s\ t))$
⟨*proof*⟩

lemma *mark-guards-guards* [*simp*]:
$mark\text{-}guards\ f\ (guards\ gs\ c) = guards\ (map\ (\lambda(f',g).\ (f,g))\ gs)\ (mark\text{-}guards\ f\ c)$
⟨*proof*⟩

lemma *mark-guards-dynCall-exn* [*simp*]:
$mark\text{-}guards\ f\ (dynCall\text{-}exn\ f'\ g\ init\ p\ return\ result\text{-}exn\ c) =$
$dynCall\text{-}exn\ f\ g\ init\ p\ return\ result\text{-}exn\ (\lambda s\ t.\ mark\text{-}guards\ f\ (c\ s\ t))$
⟨*proof*⟩

lemma *mark-guards-fcall* [*simp*]:
$mark\text{-}guards\ f\ (fcall\ init\ p\ return\ result\ c) =$
$fcall\ init\ p\ return\ result\ (\lambda v.\ mark\text{-}guards\ f\ (c\ v))$
⟨*proof*⟩

lemma *mark-guards-switch* [*simp*]:
$mark\text{-}guards\ f\ (switch\ v\ vs) =$
$switch\ v\ (map\ (\lambda(V,c).\ (V,mark\text{-}guards\ f\ c))\ vs)$
⟨*proof*⟩

lemma *mark-guards-guaranteeStrip* [*simp*]:
$mark\text{-}guards\ f\ (guaranteeStrip\ f'\ g\ c) = guaranteeStrip\ f\ g\ (mark\text{-}guards\ f\ c)$
⟨*proof*⟩

lemma *mark-guards-while* [*simp*]:
$mark\text{-}guards\ f\ (while\ gs\ b\ c) =$
$while\ (map\ (\lambda(f',g).\ (f,g))\ gs)\ b\ (mark\text{-}guards\ f\ c)$
⟨*proof*⟩

lemma *mark-guards-whileAnno* [*simp*]:
$mark\text{-}guards\ f\ (whileAnno\ b\ I\ V\ c) = whileAnno\ b\ I\ V\ (mark\text{-}guards\ f\ c)$
⟨*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 (λ*s*. *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* ⇒ ('*f* × '*s set* × ('*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* ⇒ ('*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* (λ*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* ∩ *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* ⟹ *c* = *Skip*
⟨*proof*⟩

lemma *merge-guards-res-Basic*: *merge-guards c* = *Basic f* ⟹ *c* = *Basic f*
⟨*proof*⟩

lemma *merge-guards-res-Spec*: *merge-guards c* = *Spec r* ⟹ *c* = *Spec r*
⟨*proof*⟩

lemma *merge-guards-res-Seq*: *merge-guards c* = *Seq c1 c2* ⟹
  ∃ *c1' c2'*. *c* = *Seq c1' c2'* ∧ *merge-guards c1'* = *c1* ∧ *merge-guards c2'* = *c2*

⟨*proof*⟩

lemma *merge-guards-res-Cond*: *merge-guards* c = *Cond* b *c1 c2* $\Longrightarrow$
$\exists$ *c1' c2'*. c = *Cond* b *c1' c2'* $\wedge$ *merge-guards c1'* = *c1* $\wedge$ *merge-guards c2'* = *c2*
⟨*proof*⟩

lemma *merge-guards-res-While*: *merge-guards* c = *While* b c' $\Longrightarrow$
$\exists$ c''. c = *While* b c'' $\wedge$ *merge-guards* c'' = c'
⟨*proof*⟩

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

lemma *merge-guards-res-DynCom*: *merge-guards* c = *DynCom* c' $\Longrightarrow$
$\exists$ c''. c = *DynCom* c'' $\wedge$ (λs. (*merge-guards* (c'' s))) = c'
⟨*proof*⟩

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

lemma *merge-guards-res-Catch*: *merge-guards* c = *Catch c1 c2* $\Longrightarrow$
$\exists$ *c1' c2'*. c = *Catch c1' c2'* $\wedge$ *merge-guards c1'* = *c1* $\wedge$ *merge-guards c2'* = *c2*
⟨*proof*⟩

lemma *merge-guards-res-Guard*:
merge-guards c = *Guard* f g c' $\Longrightarrow$ $\exists$ c'' f' g'. c = *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*: *merge-guards* (*guards* [] c) = *merge-guards* c
⟨*proof*⟩

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

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

lemma *merge-guards-bind* [*simp*]:
merge-guards (*bind* e c) = *bind* e (λv. *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* ($\lambda s\ t.$ *merge-guards* (*c s t*))
⟨*proof*⟩

lemma *merge-guards-block* [*simp*]:
merge-guards (*block init bdy return c*) =
block init (*merge-guards bdy*) *return* ($\lambda s\ t.$ *merge-guards* (*c s t*))
⟨*proof*⟩

lemma *merge-guards-call* [*simp*]:
merge-guards (*call init p return c*) =
call init p return ($\lambda s\ t.$ *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 ($\lambda s\ t.$ *merge-guards* (*c s t*))
⟨*proof*⟩

lemma *merge-guards-dynCall* [*simp*]:
merge-guards (*dynCall init p return c*) =
dynCall init p return ($\lambda s\ t.$ *merge-guards* (*c s t*))
⟨*proof*⟩

lemma *merge-guards-fcall* [*simp*]:
merge-guards (*fcall init p return result c*) =
fcall init p return result ($\lambda v.$ *merge-guards* (*c v*))
⟨*proof*⟩

lemma *merge-guards-switch* [*simp*]:
merge-guards (*switch v vs*) =
switch v (*map* ($\lambda(V,c).$ (V,*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*]:
merge-guards (*whileAnno b I V c*) = *whileAnno b I V* (*merge-guards c*)
⟨*proof*⟩

lemma *merge-guards-specAnno* [*simp*]:
merge-guards (*specAnno P c Q A*) =
specAnno P (*λs. merge-guards* (*c 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* ⇒ *bool*
where
noguards Skip = *True* |
noguards (*Basic f*) = *True* |
noguards (*Spec r*) = *True* |
noguards (*Seq* c_1 c_2) = (*noguards* c_1 ∧ *noguards* c_2) |
noguards (*Cond b* c_1 c_2) = (*noguards* c_1 ∧ *noguards* c_2) |
noguards (*While b c*) = (*noguards c*) |
noguards (*Call p*) = *True* |
noguards (*DynCom c*) = (∀ *s. noguards* (*c s*)) |
noguards (*Guard f g c*) = *False* |
noguards Throw = *True* |
noguards (*Catch* c_1 c_2) = (*noguards* c_1 ∧ *noguards* c_2)

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

primrec *nothrows*:: (*'s,'p,'f*) *com* ⇒ *bool*
where
nothrows Skip = *True* |
nothrows (*Basic f*) = *True* |
nothrows (*Spec r*) = *True* |
nothrows (*Seq* c_1 c_2) = (*nothrows* c_1 ∧ *nothrows* c_2) |
nothrows (*Cond b* c_1 c_2) = (*nothrows* c_1 ∧ *nothrows* c_2) |
nothrows (*While b c*) = *nothrows c* |
nothrows (*Call p*) = *True* |
nothrows (*DynCom c*) = (∀ *s. nothrows* (*c s*)) |
nothrows (*Guard f g c*) = *nothrows c* |
nothrows Throw = *False* |
nothrows (*Catch* c_1 c_2) = (*nothrows* c_1 ∧ *nothrows* c_2)

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 == 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 ⇒ *None*
| *Some t* ⇒ (*case e1* $\cap_g$ *e2 of*
None ⇒ *None*
| *Some e* ⇒ *Some* (*Cond cnd1 t e*)))
else None)
(*While cnd1 c1* $\cap_g$ *While cnd2 c2*) =
(*if cnd1* = *cnd2*
then (*case c1* $\cap_g$ *c2 of*
None ⇒ *None*
| *Some c* ⇒ *Some* (*While cnd1 c*))
else None)
(*Call p1* $\cap_g$ *Call p2*) =
(*if p1* = *p2*
then Some (*Call p1*)
else None)
(*DynCom P1* $\cap_g$ *DynCom P2*) =
(*if* (∀ *s*. (*P1 s* $\cap_g$ *P2 s*) ≠ *None*)
then Some (*DynCom* (λ*s*. *the* (*P1 s* $\cap_g$ *P2 s*)))
else None)
(*Guard m1 g1 c1* $\cap_g$ *Guard m2 g2 c2*) =
(*if m1* = *m2 then*
(*case c1* $\cap_g$ *c2 of*
None ⇒ *None*
| *Some c* ⇒ *Some* (*Guard m1* (*g1* ∩ *g2*) *c*))
else None)
(*Throw* $\cap_g$ *Throw*) = *Some Throw*
(*Catch a1 a2* $\cap_g$ *Catch b1 b2*) =
(*case a1* $\cap_g$ *b1 of*
None ⇒ *None*
| *Some c1* ⇒ (*case a2* $\cap_g$ *b2 of*
None ⇒ *None*
| *Some c2* ⇒ *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*:
⋀*c*. (*c1* $\cap_g$ *c2*) = *Some c* ⟹
(*strip-guards UNIV c* = *strip-guards UNIV c1*) ∧
(*strip-guards UNIV c* = *strip-guards UNIV c2*)
⟨*proof*⟩

lemma *inter-guards-sym*: ⋀*c*. (*c1* $\cap_g$ *c2*) = *Some c* ⟹ (*c2* $\cap_g$ *c1*) = *Some c*
⟨*proof*⟩

lemma *inter-guards-Skip*: (*Skip* $\cap_g$ *c2*) = *Some c* = (*c2*=*Skip* ∧ *c*=*Skip*)
⟨*proof*⟩

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*:
$(Guard\ f\ g1\ bdy1 \cap_g\ c2) = Some\ c =$
$(\exists\, g2\ bdy2\ bdy.\ c2{=}Guard\ f\ g2\ bdy2 \wedge (bdy1 \cap_g\ bdy2) = Some\ bdy \wedge$
$c{=}Guard\ f\ (g1 \cap g2)\ bdy)$
$\langle proof \rangle$

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

lemma *inter-guards-Catch*:

$(Catch\ a1\ a2 \cap_g c2) = Some\ c =$
$(\exists\, b1\ b2\ d1\ d2.\ c2{=}Catch\ b1\ b2 \wedge (a1 \cap_g b1) = Some\ d1 \wedge$
$(a2 \cap_g b2) = Some\ d2 \wedge c{=}Catch\ d1\ d2)$
⟨*proof*⟩

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* :: $('s,'p,'f)\ com \Rightarrow ('s,'p,'f)\ com \Rightarrow bool$
$(\langle\text{-} \subseteq_g \text{-}\rangle\ [20,20]\ 19)$ **where**
$Skip \subseteq_g Skip$
$|\ f1 = f2 \Longrightarrow Basic\ f1 \subseteq_g Basic\ f2$
$|\ r1 = r2 \Longrightarrow Spec\ r1 \subseteq_g Spec\ r2$
$|\ a1 \subseteq_g b1 \Longrightarrow a2 \subseteq_g b2 \Longrightarrow Seq\ a1\ a2 \subseteq_g Seq\ b1\ b2$
$|\ cnd1 = cnd2 \Longrightarrow t1 \subseteq_g t2 \Longrightarrow e1 \subseteq_g e2 \Longrightarrow Cond\ cnd1\ t1\ e1 \subseteq_g Cond\ cnd2\ t2\ e2$
$|\ cnd1 = cnd2 \Longrightarrow c1 \subseteq_g c2 \Longrightarrow While\ cnd1\ c1 \subseteq_g While\ cnd2\ c2$
$|\ p1 = p2 \Longrightarrow Call\ p1 \subseteq_g Call\ p2$
$|\ (\bigwedge s.\ P1\ s \subseteq_g P2\ s) \Longrightarrow DynCom\ P1 \subseteq_g DynCom\ P2$
$|\ m1 = m2 \Longrightarrow g1 = g2 \Longrightarrow c1 \subseteq_g c2 \Longrightarrow Guard\ m1\ g1\ c1 \subseteq_g Guard\ m2\ g2\ c2$
$|\ c1 \subseteq_g c2 \Longrightarrow c1 \subseteq_g Guard\ m2\ g2\ c2$
$|\ Throw \subseteq_g Throw$
$|\ a1 \subseteq_g b1 \Longrightarrow a2 \subseteq_g b2 \Longrightarrow Catch\ a1\ a2 \subseteq_g Catch\ b1\ b2$

lemma *subseteq-guards-Skip*:
$c = Skip$ **if** $c \subseteq_g Skip$
⟨*proof*⟩

lemma *subseteq-guards-Basic*:
$c = Basic\ f$ **if** $c \subseteq_g Basic\ f$
⟨*proof*⟩

lemma *subseteq-guards-Spec*:
$c = Spec\ r$ **if** $c \subseteq_g Spec\ r$
⟨*proof*⟩

lemma *subseteq-guards-Seq*:
$\exists\, c1'\ c2'.\ c = Seq\ c1'\ c2' \wedge (c1' \subseteq_g c1) \wedge (c2' \subseteq_g c2)$ **if** $c \subseteq_g Seq\ c1\ c2$
⟨*proof*⟩

lemma *subseteq-guards-Cond*:
$\exists\, c1'\ c2'.\ c{=}Cond\ b\ c1'\ c2' \wedge (c1' \subseteq_g c1) \wedge (c2' \subseteq_g c2)$ **if** $c \subseteq_g Cond\ b\ c1\ c2$
⟨*proof*⟩

lemma *subseteq-guards-While*:
$\exists c''.\ c{=}While\ b\ c'' \wedge (c'' \subseteq_g c')$ **if** $c \subseteq_g While\ b\ c'$
⟨*proof*⟩

lemma *subseteq-guards-Call*:
$c = Call\ p$ **if** $c \subseteq_g Call\ p$
⟨*proof*⟩

lemma *subseteq-guards-DynCom*:
$\exists C'.\ c{=}DynCom\ C' \wedge (\forall s.\ C'\ s \subseteq_g C\ s)$ **if** $c \subseteq_g DynCom\ C$
⟨*proof*⟩

lemma *subseteq-guards-Guard*:
$(c \subseteq_g c') \vee (\exists c''.\ c = Guard\ f\ g\ c'' \wedge (c'' \subseteq_g c'))$ **if** $c \subseteq_g Guard\ f\ g\ c'$
⟨*proof*⟩

lemma *subseteq-guards-Throw*:
$c = Throw$ **if** $c \subseteq_g Throw$
⟨*proof*⟩

lemma *subseteq-guards-Catch*:
$\exists c1'\ c2'.\ c = Catch\ c1'\ c2' \wedge (c1' \subseteq_g c1) \wedge (c2' \subseteq_g c2)$ **if** $c \subseteq_g Catch\ c1\ c2$
⟨*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'*:
$\exists f'\ b'\ c'.\ d = Guard\ f'\ b'\ c'$ **if** $Guard\ f\ b\ c \subseteq_g d$
⟨*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 (‹-|-› [*90*, *91*] *90*)

datatype (*'s*,*'f*) *xstate* = *Normal 's* | *Abrupt 's* | *Fault 'f* | *Stuck*

definition *isAbr*::(*'s*,*'f*) *xstate* ⇒ *bool*
  where *isAbr S* = (∃ *s*. *S*=*Abrupt s*)

lemma *isAbr-simps* [*simp*]:
isAbr (*Normal s*) = *False*
isAbr (*Abrupt s*) = *True*
isAbr (*Fault f*) = *False*
isAbr Stuck = *False*
⟨*proof*⟩

lemma *isAbrE* [*consumes 1*, *elim?*]: ⟦*isAbr S*; ⋀*s*. *S*=*Abrupt s* ⟹ *P*⟧ ⟹ *P*
  ⟨*proof*⟩

lemma *not-isAbrD*:
¬ *isAbr s* ⟹ (∃ *s'*. *s*=*Normal s'*) ∨ *s* = *Stuck* ∨ (∃ *f*. *s*=*Fault f*)
  ⟨*proof*⟩

definition *isFault*:: (*'s*,*'f*) *xstate* ⇒ *bool*
  where *isFault S* = (∃ *f*. *S*=*Fault f*)

lemma *isFault-simps* [*simp*]:
isFault (*Normal s*) = *False*
isFault (*Abrupt s*) = *False*
isFault (*Fault f*) = *True*
isFault Stuck = *False*
⟨*proof*⟩

lemma *isFaultE* [*consumes 1*, *elim?*]: ⟦*isFault s*; ⋀*f*. *s*=*Fault f* ⟹ *P*⟧ ⟹ *P*
  ⟨*proof*⟩

lemma *not-isFault-iff*: (¬ *isFault t*) = (∀ *f*. *t* ≠ *Fault f*)
  ⟨*proof*⟩

3.1 Big-Step Execution: Γ⊢⟨*c*, *s*⟩ ⇒ *t*

The procedure environment

type-synonym (*'s*,*'p*,*'f*) *body* = *'p* ⇒ (*'s*,*'p*,*'f*) *com option*

inductive
  exec::[(*'s*,*'p*,*'f*) *body*,(*'s*,*'p*,*'f*) *com*,(*'s*,*'f*) *xstate*,(*'s*,*'f*) *xstate*]
                    ⇒ *bool* (‹-⊢ ⟨-,-⟩ ⇒ -› [*60*,*20*,*98*,*98*] *89*)
  for Γ::(*'s*,*'p*,*'f*) *body*
where
  Skip: Γ⊢⟨*Skip*,*Normal s*⟩ ⇒ *Normal s*

| *Guard*: ⟦*s*∈*g*; Γ⊢⟨*c*,*Normal s*⟩ ⇒ *t*⟧
          ⟹
          Γ⊢⟨*Guard f g c*,*Normal s*⟩ ⇒ *t*

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

| *FaultProp* [*intro*,*simp*]: $\Gamma \vdash \langle c, Fault\ f\rangle \Rightarrow\ Fault\ f$

| *Basic*: $\Gamma \vdash \langle Basic\ f, Normal\ s\rangle \Rightarrow\ Normal\ (f\ s)$

| *Spec*: $(s,t) \in r$
$\Longrightarrow$
$\Gamma \vdash \langle Spec\ r, Normal\ s\rangle \Rightarrow\ Normal\ t$

| *SpecStuck*: $\forall t.\ (s,t) \notin r$
$\Longrightarrow$
$\Gamma \vdash \langle Spec\ r, Normal\ s\rangle \Rightarrow\ Stuck$

| *Seq*: $[\![\Gamma \vdash \langle c_1, Normal\ s\rangle \Rightarrow\ s';\ \Gamma \vdash \langle c_2, s'\rangle \Rightarrow\ t]\!]$
$\Longrightarrow$
$\Gamma \vdash \langle Seq\ c_1\ c_2, Normal\ s\rangle \Rightarrow\ t$

| *CondTrue*: $[\![s \in b;\ \Gamma \vdash \langle c_1, Normal\ s\rangle \Rightarrow\ t]\!]$
$\Longrightarrow$
$\Gamma \vdash \langle Cond\ b\ c_1\ c_2, Normal\ s\rangle \Rightarrow\ t$

| *CondFalse*: $[\![s \notin b;\ \Gamma \vdash \langle c_2, Normal\ s\rangle \Rightarrow\ t]\!]$
$\Longrightarrow$
$\Gamma \vdash \langle Cond\ b\ c_1\ c_2, Normal\ s\rangle \Rightarrow\ t$

| *WhileTrue*: $[\![s \in b;\ \Gamma \vdash \langle c, Normal\ s\rangle \Rightarrow\ s';\ \Gamma \vdash \langle While\ b\ c, s'\rangle \Rightarrow\ t]\!]$
$\Longrightarrow$
$\Gamma \vdash \langle While\ b\ c, Normal\ s\rangle \Rightarrow\ t$

| *WhileFalse*: $[\![s \notin b]\!]$
$\Longrightarrow$
$\Gamma \vdash \langle While\ b\ c, Normal\ s\rangle \Rightarrow\ Normal\ s$

| *Call*: $[\![\Gamma\ p = Some\ bdy; \Gamma \vdash \langle bdy, Normal\ s\rangle \Rightarrow\ t]\!]$
$\Longrightarrow$
$\Gamma \vdash \langle Call\ p, Normal\ s\rangle \Rightarrow\ t$

| *CallUndefined*: $[\![\Gamma\ p = None]\!]$
$\Longrightarrow$
$\Gamma \vdash \langle Call\ p, Normal\ s\rangle \Rightarrow\ Stuck$

| *StuckProp* [*intro*,*simp*]: $\Gamma \vdash \langle c, Stuck\rangle \Rightarrow\ Stuck$

| *DynCom*: $[\![\Gamma \vdash \langle (c\ s), Normal\ s\rangle \Rightarrow\ t]\!]$
$\Longrightarrow$
$\Gamma \vdash \langle DynCom\ c, Normal\ s\rangle \Rightarrow\ t$

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

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

| *CatchMatch*: $[\![\Gamma\vdash\langle c_1,Normal\ s\rangle \Rightarrow\ Abrupt\ s'; \Gamma\vdash\langle c_2,Normal\ s'\rangle \Rightarrow\ t]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle Catch\ c_1\ c_2,Normal\ s\rangle \Rightarrow\ t$
| *CatchMiss*: $[\![\Gamma\vdash\langle c_1,Normal\ s\rangle \Rightarrow\ t; \neg isAbr\ t]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle Catch\ c_1\ c_2,Normal\ s\rangle \Rightarrow\ t$

inductive-cases *exec-elim-cases* [*cases set*]:
$\Gamma\vdash\langle c,Fault\ f\rangle \Rightarrow\ t$
$\Gamma\vdash\langle c,Stuck\rangle \Rightarrow\ t$
$\Gamma\vdash\langle c,Abrupt\ s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Skip,s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Seq\ c1\ c2,s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Guard\ f\ g\ c,s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Basic\ f,s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Spec\ r,s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Cond\ b\ c1\ c2,s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle While\ b\ c,s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Call\ p,s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle DynCom\ c,s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Throw,s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Catch\ c1\ c2,s\rangle \Rightarrow\ t$

inductive-cases *exec-Normal-elim-cases* [*cases set*]:
$\Gamma\vdash\langle c,Fault\ f\rangle \Rightarrow\ t$
$\Gamma\vdash\langle c,Stuck\rangle \Rightarrow\ t$
$\Gamma\vdash\langle c,Abrupt\ s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Skip,Normal\ s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Guard\ f\ g\ c,Normal\ s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Basic\ f,Normal\ s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Spec\ r,Normal\ s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Seq\ c1\ c2,Normal\ s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Cond\ b\ c1\ c2,Normal\ s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle While\ b\ c,Normal\ s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Call\ p,Normal\ s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle DynCom\ c,Normal\ s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Throw,Normal\ s\rangle \Rightarrow\ t$
$\Gamma\vdash\langle Catch\ c1\ c2,Normal\ s\rangle \Rightarrow\ t$

lemma *exec-block-exn*:
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Normal\ t; \Gamma\vdash\langle c\ s\ t,Normal\ (return\ s\ t)\rangle \Rightarrow\ u]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle block\text{-}exn\ init\ bdy\ return\ result\text{-}exn\ c,Normal\ s\rangle \Rightarrow\ u$

⟨*proof*⟩

lemma *exec-block*:
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Normal\ t;\ \Gamma\vdash\langle c\ s\ t,Normal\ (return\ s\ t)\rangle \Rightarrow\ u]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle block\ init\ bdy\ return\ c,Normal\ s\rangle \Rightarrow\ u$
⟨*proof*⟩

lemma *exec-block-exnAbrupt*:
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Abrupt\ t]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle block\text{-}exn\ init\ bdy\ return\ result\text{-}exn\ c,Normal\ s\rangle \Rightarrow\ Abrupt\ (result\text{-}exn\ (return\ s\ t)\ t)$
⟨*proof*⟩

lemma *exec-blockAbrupt*:
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Abrupt\ t]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle block\ init\ bdy\ return\ c,Normal\ s\rangle \Rightarrow\ Abrupt\ (return\ s\ t)$
⟨*proof*⟩

lemma *exec-block-exnFault*:
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Fault\ f]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle block\text{-}exn\ init\ bdy\ return\ result\text{-}exn\ c,Normal\ s\rangle \Rightarrow\ Fault\ f$
⟨*proof*⟩

lemma *exec-blockFault*:
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Fault\ f]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle block\ init\ bdy\ return\ c,Normal\ s\rangle \Rightarrow\ Fault\ f$
⟨*proof*⟩

lemma *exec-block-exnStuck*:
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Stuck]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle block\text{-}exn\ init\ bdy\ return\ result\text{-}exn\ c,Normal\ s\rangle \Rightarrow\ Stuck$
⟨*proof*⟩

lemma *exec-blockStuck*:
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Stuck]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle block\ init\ bdy\ return\ c,Normal\ s\rangle \Rightarrow\ Stuck$
⟨*proof*⟩

lemma *exec-call*:
$[\![\Gamma\ p{=}Some\ bdy;\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Normal\ t;\ \Gamma\vdash\langle c\ s\ t,Normal\ (return\ s\ t)\rangle \Rightarrow\ u]\!]$
$\Longrightarrow$

$\Gamma\vdash\langle call\ init\ p\ return\ c,Normal\ s\rangle \Rightarrow\ u$
⟨*proof*⟩

lemma *exec-callAbrupt*:
$[\![\Gamma\ p{=}Some\ bdy;\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Abrupt\ t]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\ init\ p\ return\ c,Normal\ s\rangle \Rightarrow\ Abrupt\ (return\ s\ t)$
⟨*proof*⟩

lemma *exec-callFault*:
$[\![\Gamma\ p{=}Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Fault\ f]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\ init\ p\ return\ c,Normal\ s\rangle \Rightarrow\ Fault\ f$
⟨*proof*⟩

lemma *exec-callStuck*:
$[\![\Gamma\ p{=}Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Stuck]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\ init\ p\ return\ c,Normal\ s\rangle \Rightarrow\ Stuck$
⟨*proof*⟩

lemma *exec-callUndefined*:
$[\![\Gamma\ p{=}None]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\ init\ p\ return\ c,Normal\ s\rangle \Rightarrow\ Stuck$
⟨*proof*⟩

lemma *exec-call-exn*:
$[\![\Gamma\ p{=}Some\ bdy;\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Normal\ t;\ \Gamma\vdash\langle c\ s\ t,Normal\ (return\ s\ t)\rangle \Rightarrow\ u]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c,Normal\ s\rangle \Rightarrow\ u$
⟨*proof*⟩

lemma *exec-call-exnAbrupt*:
$[\![\Gamma\ p{=}Some\ bdy;\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Abrupt\ t]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c,Normal\ s\rangle \Rightarrow\ Abrupt\ (result\text{-}exn\ (return\ s\ t)\ t)$
⟨*proof*⟩

lemma *exec-call-exnFault*:
$[\![\Gamma\ p{=}Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Fault\ f]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c,Normal\ s\rangle \Rightarrow\ Fault\ f$
⟨*proof*⟩

lemma *exec-call-exnStuck*:
$[\![\Gamma\ p{=}Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Stuck]\!]$

$\Longrightarrow$
$\Gamma \vdash \langle$*call-exn init p return result-exn c*,*Normal s*$\rangle \Rightarrow$ *Stuck*
$\langle$*proof*$\rangle$

lemma *exec-call-exnUndefined*:
$[\![\Gamma$ *p*=*None*$]\!]$
$\Longrightarrow$
$\Gamma \vdash \langle$*call-exn init p return result-exn c*,*Normal s*$\rangle \Rightarrow$ *Stuck*
$\langle$*proof*$\rangle$

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

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

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

lemma *exec-Call-body-aux*:
Γ *p*=*Some bdy* $\Longrightarrow$
$\Gamma \vdash \langle$*Call p*,*s*$\rangle \Rightarrow t = \Gamma \vdash \langle$*bdy*,*s*$\rangle \Rightarrow t$
$\langle$*proof*$\rangle$

lemma *exec-Call-body'*:
$p \in$ *dom* $\Gamma \Longrightarrow$
$\Gamma \vdash \langle$*Call p*,*s*$\rangle \Rightarrow t = \Gamma \vdash \langle$*the* $(\Gamma\ p)$,*s*$\rangle \Rightarrow t$
$\langle$*proof*$\rangle$

lemma *exec-block-exn-Normal-elim* [*consumes 1*]:
assumes *exec-block*: $\Gamma \vdash \langle$*block-exn init bdy return result-exn c*,*Normal s*$\rangle \Rightarrow$ *t*
assumes *Normal*:
$\bigwedge t'$.
$[\![\Gamma \vdash \langle$*bdy*,*Normal* (*init s*)$\rangle \Rightarrow$ *Normal t'*;
$\Gamma \vdash \langle$*c s t'*,*Normal* (*return s t'*)$\rangle \Rightarrow$ *t*$]\!]$
$\Longrightarrow P$
assumes *Abrupt*:
$\bigwedge t'$.
$[\![\Gamma \vdash \langle$*bdy*,*Normal* (*init s*)$\rangle \Rightarrow$ *Abrupt t'*;
t = *Abrupt* (*result-exn* (*return s t'*) *t'*)$]\!]$
$\Longrightarrow P$
assumes *Fault*:
$\bigwedge f$.
$[\![\Gamma \vdash \langle$*bdy*,*Normal* (*init s*)$\rangle \Rightarrow$ *Fault f*;

$t = \textit{Fault } f]\!]$
$\Longrightarrow P$
assumes *Stuck*:
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Stuck;$
$t = Stuck]\!]$
$\Longrightarrow P$
assumes
$[\![\Gamma\ p = None;\ t = Stuck]\!] \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *exec-block-Normal-elim* [*consumes 1*]:
assumes *exec-block*: $\Gamma\vdash\langle block\ init\ bdy\ return\ c,Normal\ s\rangle \Rightarrow\ t$
assumes *Normal*:
$\bigwedge t'.$
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Normal\ t';$
$\Gamma\vdash\langle c\ s\ t',Normal\ (return\ s\ t')\rangle \Rightarrow\ t]\!]$
$\Longrightarrow P$
assumes *Abrupt*:
$\bigwedge t'.$
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Abrupt\ t';$
$t = Abrupt\ (return\ s\ t')]\!]$
$\Longrightarrow P$
assumes *Fault*:
$\bigwedge f.$
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Fault\ f;$
$t = Fault\ f]\!]$
$\Longrightarrow P$
assumes *Stuck*:
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Stuck;$
$t = Stuck]\!]$
$\Longrightarrow P$
assumes
Undef: $[\![\Gamma\ p = None;\ t = Stuck]\!] \Longrightarrow P$
shows P
⟨*proof*⟩

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

$\Longrightarrow P$
assumes *Fault*:
$\bigwedge bdy\ f.$
$[\![\Gamma\ p = Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Fault\ f;$
$t = Fault\ f]\!]$
$\Longrightarrow P$
assumes *Stuck*:
$\bigwedge bdy.$
$[\![\Gamma\ p = Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Stuck;$
$t = Stuck]\!]$
$\Longrightarrow P$
assumes *Undef*:
$[\![\Gamma\ p = None;\ t = Stuck]\!] \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *exec-call-Normal-elim* [*consumes 1*]:
assumes *exec-call*: $\Gamma\vdash\langle call\ init\ p\ return\ c,Normal\ s\rangle \Rightarrow\ t$
assumes *Normal*:
$\bigwedge bdy\ t'.$
$[\![\Gamma\ p = Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Normal\ t';$
$\Gamma\vdash\langle c\ s\ t',Normal\ (return\ s\ t')\rangle \Rightarrow\ t]\!]$
$\Longrightarrow P$
assumes *Abrupt*:
$\bigwedge bdy\ t'.$
$[\![\Gamma\ p = Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Abrupt\ t';$
$t = Abrupt\ (return\ s\ t')]\!]$
$\Longrightarrow P$
assumes *Fault*:
$\bigwedge bdy\ f.$
$[\![\Gamma\ p = Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Fault\ f;$
$t = Fault\ f]\!]$
$\Longrightarrow P$
assumes *Stuck*:
$\bigwedge bdy.$
$[\![\Gamma\ p = Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow\ Stuck;$
$t = Stuck]\!]$
$\Longrightarrow P$
assumes *Undef*:
$[\![\Gamma\ p = None;\ t = Stuck]\!] \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *exec-dynCall*:
$[\![\Gamma\vdash\langle call\ init\ (p\ s)\ return\ c,Normal\ s\rangle \Rightarrow\ t]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle dynCall\ init\ p\ return\ c,Normal\ s\rangle \Rightarrow\ t$
⟨*proof*⟩

lemma *exec-dynCall-exn*:
$[\![\Gamma\vdash\langle$*call-exn init* $(p\ s)$ *return result-exn c*,*Normal* $s\rangle \Rightarrow\ t]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle$*dynCall-exn f UNIV init p return result-exn c*,*Normal* $s\rangle \Rightarrow\ t$
⟨*proof*⟩

lemma *exec-dynCall-Normal-elim*:
assumes *exec*: $\Gamma\vdash\langle$*dynCall init p return c*,*Normal* $s\rangle \Rightarrow\ t$
assumes *call*: $\Gamma\vdash\langle$*call init* $(p\ s)$ *return c*,*Normal* $s\rangle \Rightarrow\ t \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *exec-guards-Normal-elim-cases* [*consumes 1*, *case-names noFault someFault*]:
assumes *exec-guards*: $\Gamma\vdash\langle$*guards gs c*,*Normal* $s\rangle \Rightarrow t$
assumes *noFault*: $\forall f\ g.\ (f,\ g) \in$ *set gs* $\longrightarrow s \in g \Longrightarrow \Gamma\vdash\langle c$,*Normal* $s\rangle \Rightarrow t \Longrightarrow P$
assumes *someFault*: $\bigwedge f\ g.$ *find* $(\lambda(f,g).\ s \notin g)$ *gs* = *Some* $(f,\ g) \Longrightarrow t =$ *Fault f* $\Longrightarrow P$
shows P
⟨*proof*⟩

lemma *exec-guards-noFault*:
assumes *exec*: $\Gamma\vdash\langle c$,*Normal* $s\rangle \Rightarrow t$
assumes *noFault*: $\forall f\ g.\ (f,\ g) \in$ *set gs* $\longrightarrow s \in g$
shows $\Gamma\vdash\langle$*guards gs c*,*Normal* $s\rangle \Rightarrow t$
⟨*proof*⟩

lemma *exec-guards-Fault*:
assumes *Fault*: *find* $(\lambda(f,g).\ s \notin g)$ *gs* = *Some* $(f,\ g)$
shows $\Gamma\vdash\langle$*guards gs c*,*Normal* $s\rangle \Rightarrow$ *Fault f*
⟨*proof*⟩

lemma *exec-guards-DynCom*:
assumes *exec-c*: $\Gamma\vdash\langle$*guards gs* $(c\ s)$, *Normal* $s\rangle \Rightarrow t$
shows $\Gamma\vdash\langle$*guards gs* (*DynCom c*), *Normal* $s\rangle \Rightarrow t$
⟨*proof*⟩

lemma *exec-guards-DynCom-Normal-elim*:
assumes *exec*: $\Gamma\vdash\langle$*guards gs* (*DynCom c*), *Normal* $s\rangle \Rightarrow t$
assumes *call*: $\Gamma\vdash\langle$*guards gs* $(c\ s)$, *Normal* $s\rangle \Rightarrow\ t \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *exec-maybe-guard-DynCom*:
assumes *exec-c*: $\Gamma\vdash\langle$*maybe-guard f g* $(c\ s)$, *Normal* $s\rangle \Rightarrow t$
shows $\Gamma\vdash\langle$*maybe-guard f g* (*DynCom c*), *Normal* $s\rangle \Rightarrow t$
⟨*proof*⟩

lemma *exec-maybe-guard-Normal-elim-cases* [*consumes 1*, *case-names noFault someFault*]:
assumes *exec-guards*: $\Gamma\vdash\langle$*maybe-guard f g c*,*Normal s*$\rangle \Rightarrow t$
assumes *noFault*: $s \in g \Longrightarrow \Gamma\vdash\langle c,$*Normal s*$\rangle \Rightarrow t \Longrightarrow P$
assumes *someFault*: $s \notin g \Longrightarrow t =$ *Fault f* $\Longrightarrow P$
shows P
⟨*proof*⟩

lemma *exec-maybe-guard-noFault*:
assumes *exec*: $\Gamma\vdash\langle c,$*Normal s*$\rangle \Rightarrow t$
assumes *noFault*: $s \in g$
shows $\Gamma\vdash\langle$*maybe-guard f g c*,*Normal s*$\rangle \Rightarrow t$
⟨*proof*⟩

lemma *exec-maybe-guard-Fault*:
assumes *Fault*: $s \notin g$
shows $\Gamma\vdash\langle$*maybe-guard f g c*,*Normal s*$\rangle \Rightarrow$ *Fault f*
⟨*proof*⟩

lemma *exec-maybe-guard-DynCom-Normal-elim*:
assumes *exec*: $\Gamma\vdash\langle$*maybe-guard f g* (*DynCom c*), *Normal s*$\rangle \Rightarrow t$
assumes *call*: $\Gamma\vdash\langle$*maybe-guard f g* (*c s*), *Normal s*$\rangle \Rightarrow t \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *exec-dynCall-exn-Normal-elim*:
assumes *exec*: $\Gamma\vdash\langle$*dynCall-exn f g init p return result-exn c*,*Normal s*$\rangle \Rightarrow t$
assumes *call*: $\Gamma\vdash\langle$*maybe-guard f g* (*call-exn init* (*p s*) *return result-exn c*),*Normal s*$\rangle \Rightarrow t \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *exec-Call-body*:
Γ *p*=*Some bdy* $\Longrightarrow$
$\Gamma\vdash\langle$*Call p*,*s*$\rangle \Rightarrow t = \Gamma\vdash\langle$*the* ($\Gamma$ *p*),*s*$\rangle \Rightarrow t$
⟨*proof*⟩

lemma *exec-Seq'*: $[\![\Gamma\vdash\langle c1,s\rangle \Rightarrow s'; \Gamma\vdash\langle c2,s'\rangle \Rightarrow s'']\!]$
$\Longrightarrow$
$\Gamma\vdash\langle$*Seq c1 c2*,*s*$\rangle \Rightarrow s''$
⟨*proof*⟩

lemma *exec-assoc*: $\Gamma\vdash\langle$*Seq c1* (*Seq c2 c3*),*s*$\rangle \Rightarrow t = \Gamma\vdash\langle$*Seq* (*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]
⇒ bool (‹-⊢ ⟨-,-⟩ =-⇒ -› [60,20,98,65,98] 89)
for Γ::('s,'p,'f) body
where
Skip: Γ⊢⟨Skip,Normal s⟩ =n⇒ Normal s
| *Guard*: ⟦s∈g; Γ⊢⟨c,Normal s⟩ =n⇒ t⟧
⟹
Γ⊢⟨Guard f g c,Normal s⟩ =n⇒ t

| *GuardFault*: s∉g ⟹ Γ⊢⟨Guard f g c,Normal s⟩ =n⇒ Fault f

| *FaultProp* [intro,simp]: Γ⊢⟨c,Fault f⟩ =n⇒ Fault f

| *Basic*: Γ⊢⟨Basic f,Normal s⟩ =n⇒ Normal (f s)

| *Spec*: (s,t) ∈ r
⟹
Γ⊢⟨Spec r,Normal s⟩ =n⇒ Normal t

| *SpecStuck*: ∀ t. (s,t) ∉ r
⟹
Γ⊢⟨Spec r,Normal s⟩ =n⇒ Stuck

| *Seq*: ⟦Γ⊢⟨c_1,Normal s⟩ =n⇒ s'; Γ⊢⟨c_2,s'⟩ =n⇒ t⟧
⟹
Γ⊢⟨Seq c_1 c_2,Normal s⟩ =n⇒ t

| *CondTrue*: ⟦s ∈ b; Γ⊢⟨c_1,Normal s⟩ =n⇒ t⟧
⟹
Γ⊢⟨Cond b c_1 c_2,Normal s⟩ =n⇒ t

| *CondFalse*: ⟦s ∉ b; Γ⊢⟨c_2,Normal s⟩ =n⇒ t⟧
⟹
Γ⊢⟨Cond b c_1 c_2,Normal s⟩ =n⇒ t

| *WhileTrue*: ⟦s ∈ b; Γ⊢⟨c,Normal s⟩ =n⇒ s';
Γ⊢⟨While b c,s'⟩ =n⇒ t⟧
⟹
Γ⊢⟨While b c,Normal s⟩ =n⇒ t

| *WhileFalse*: ⟦s ∉ b⟧
⟹
Γ⊢⟨While b c,Normal s⟩ =n⇒ Normal s

| *Call*: ⟦Γ p=Some bdy;Γ⊢⟨bdy,Normal s⟩ =n⇒ t⟧
⟹
Γ⊢⟨Call p ,Normal s⟩ =Suc n⇒ t

| *CallUndefined*: $[\![\Gamma\ p=None]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle Call\ p\ ,Normal\ s\rangle =Suc\ n\Rightarrow\ Stuck$

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

| *DynCom*: $[\![\Gamma\vdash\langle (c\ s),Normal\ s\rangle =n\Rightarrow\ t]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle DynCom\ c,Normal\ s\rangle =n\Rightarrow\ t$

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

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

| *CatchMatch*: $[\![\Gamma\vdash\langle c_1,Normal\ s\rangle =n\Rightarrow\ Abrupt\ s'; \Gamma\vdash\langle c_2,Normal\ s'\rangle =n\Rightarrow\ t]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle Catch\ c_1\ c_2,Normal\ s\rangle =n\Rightarrow\ t$
| *CatchMiss*: $[\![\Gamma\vdash\langle c_1,Normal\ s\rangle =n\Rightarrow\ t; \neg isAbr\ t]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle Catch\ c_1\ c_2,Normal\ s\rangle =n\Rightarrow\ t$

inductive-cases *execn-elim-cases* [*cases set*]:
$\Gamma\vdash\langle c,Fault\ f\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle c,Stuck\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle c,Abrupt\ s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle Skip,s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle Seq\ c1\ c2,s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle Guard\ f\ g\ c,s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle Basic\ f,s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle Spec\ r,s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle Cond\ b\ c1\ c2,s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle While\ b\ c,s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle Call\ p\ ,s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle DynCom\ c,s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle Throw,s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle Catch\ c1\ c2,s\rangle =n\Rightarrow\ t$

inductive-cases *execn-Normal-elim-cases* [*cases set*]:
$\Gamma\vdash\langle c,Fault\ f\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle c,Stuck\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle c,Abrupt\ s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle Skip,Normal\ s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle Guard\ f\ g\ c,Normal\ s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle Basic\ f,Normal\ s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle Spec\ r,Normal\ s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle Seq\ c1\ c2,Normal\ s\rangle =n\Rightarrow\ t$
$\Gamma\vdash\langle Cond\ b\ c1\ c2,Normal\ s\rangle =n\Rightarrow\ t$

Γ⊢⟨*While b c*,*Normal s*⟩ =*n*⇒ *t*
Γ⊢⟨*Call p*,*Normal s*⟩ =*n*⇒ *t*
Γ⊢⟨*DynCom c*,*Normal s*⟩ =*n*⇒ *t*
Γ⊢⟨*Throw*,*Normal s*⟩ =*n*⇒ *t*
Γ⊢⟨*Catch c1 c2*,*Normal s*⟩ =*n*⇒ *t*

lemma *execn-Skip'*: Γ⊢⟨*Skip*,*t*⟩ =*n*⇒ *t*
⟨*proof*⟩

lemma *execn-Fault-end*: **assumes** *exec*: Γ⊢⟨*c*,*s*⟩ =*n*⇒ *t* **and** *s*: *s*=*Fault f*
shows *t*=*Fault f*
⟨*proof*⟩

lemma *execn-Stuck-end*: **assumes** *exec*: Γ⊢⟨*c*,*s*⟩ =*n*⇒ *t* **and** *s*: *s*=*Stuck*
shows *t*=*Stuck*
⟨*proof*⟩

lemma *execn-Abrupt-end*: **assumes** *exec*: Γ⊢⟨*c*,*s*⟩ =*n*⇒ *t* **and** *s*: *s*=*Abrupt s'*
shows *t*=*Abrupt s'*
⟨*proof*⟩

lemma *execn-block-exn*:
⟦Γ⊢⟨*bdy*,*Normal* (*init s*)⟩ =*n*⇒ *Normal t*; Γ⊢⟨*c s t*,*Normal* (*return s t*)⟩ =*n*⇒ *u*⟧
⟹
Γ⊢⟨*block-exn init bdy return result-exn c*,*Normal s*⟩ =*n*⇒ *u*
⟨*proof*⟩

lemma *execn-block*:
⟦Γ⊢⟨*bdy*,*Normal* (*init s*)⟩ =*n*⇒ *Normal t*; Γ⊢⟨*c s t*,*Normal* (*return s t*)⟩ =*n*⇒ *u*⟧
⟹
Γ⊢⟨*block init bdy return c*,*Normal s*⟩ =*n*⇒ *u*
⟨*proof*⟩

lemma *execn-block-exnAbrupt*:
⟦Γ⊢⟨*bdy*,*Normal* (*init s*)⟩ =*n*⇒ *Abrupt t*⟧
⟹
Γ⊢⟨*block-exn init bdy return result-exn c*,*Normal s*⟩ =*n*⇒ *Abrupt* (*result-exn* (*return s t*) *t*)
⟨*proof*⟩

lemma *execn-blockAbrupt*:
⟦Γ⊢⟨*bdy*,*Normal* (*init s*)⟩ =*n*⇒ *Abrupt t*⟧
⟹
Γ⊢⟨*block init bdy return c*,*Normal s*⟩ =*n*⇒ *Abrupt* (*return s t*)
⟨*proof*⟩

lemma *execn-block-exnFault*:

$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =n\Rightarrow\ Fault\ f]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle block\text{-}exn\ init\ bdy\ return\ result\text{-}exn\ c,Normal\ s\rangle =n\Rightarrow\ Fault\ f$
⟨*proof*⟩

lemma *execn-blockFault*:
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =n\Rightarrow\ Fault\ f]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle block\ init\ bdy\ return\ c,Normal\ s\rangle =n\Rightarrow\ Fault\ f$
⟨*proof*⟩

lemma *execn-block-exnStuck*:
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =n\Rightarrow\ Stuck]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle block\text{-}exn\ init\ bdy\ return\ result\text{-}exn\ c,Normal\ s\rangle =n\Rightarrow\ Stuck$
⟨*proof*⟩

lemma *execn-blockStuck*:
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =n\Rightarrow\ Stuck]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle block\ init\ bdy\ return\ c,Normal\ s\rangle =n\Rightarrow\ Stuck$
⟨*proof*⟩

lemma *execn-call*:
$[\![\Gamma\ p=Some\ bdy;\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =n\Rightarrow\ Normal\ t;$
$\Gamma\vdash\langle c\ s\ t,Normal\ (return\ s\ t)\rangle =Suc\ n\Rightarrow\ u]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\ init\ p\ return\ c,Normal\ s\rangle =Suc\ n\Rightarrow\ u$
⟨*proof*⟩

lemma *execn-call-exn*:
$[\![\Gamma\ p=Some\ bdy;\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =n\Rightarrow\ Normal\ t;$
$\Gamma\vdash\langle c\ s\ t,Normal\ (return\ s\ t)\rangle =Suc\ n\Rightarrow\ u]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c,Normal\ s\rangle =Suc\ n\Rightarrow\ u$
⟨*proof*⟩

lemma *execn-callAbrupt*:
$[\![\Gamma\ p=Some\ bdy;\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =n\Rightarrow\ Abrupt\ t]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\ init\ p\ return\ c,Normal\ s\rangle =Suc\ n\Rightarrow\ Abrupt\ (return\ s\ t)$
⟨*proof*⟩

lemma *execn-call-exnAbrupt*:
$[\![\Gamma\ p=Some\ bdy;\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =n\Rightarrow\ Abrupt\ t]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c,Normal\ s\rangle =Suc\ n\Rightarrow\ Abrupt\ (result\text{-}exn$

$(return\ s\ t)\ t)$
⟨proof⟩

lemma *execn-callFault*:
$[\![\Gamma\ p{=}Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle\ ={n}\Rightarrow\ Fault\ f]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\ init\ p\ return\ c,Normal\ s\rangle\ =Suc\ n\Rightarrow\ Fault\ f$
⟨proof⟩

lemma *execn-call-exnFault*:
$[\![\Gamma\ p{=}Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle\ ={n}\Rightarrow\ Fault\ f]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c,Normal\ s\rangle\ =Suc\ n\Rightarrow\ Fault\ f$
⟨proof⟩

lemma *execn-callStuck*:
$[\![\Gamma\ p{=}Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle\ ={n}\Rightarrow\ Stuck]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\ init\ p\ return\ c,Normal\ s\rangle\ =Suc\ n\Rightarrow\ Stuck$
⟨proof⟩

lemma *execn-call-exnStuck*:
$[\![\Gamma\ p{=}Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle\ ={n}\Rightarrow\ Stuck]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c,Normal\ s\rangle\ =Suc\ n\Rightarrow\ Stuck$
⟨proof⟩

lemma *execn-callUndefined*:
$[\![\Gamma\ p{=}None]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\ init\ p\ return\ c,Normal\ s\rangle\ =Suc\ n\Rightarrow\ Stuck$
⟨proof⟩

lemma *execn-call-exnUndefined*:
$[\![\Gamma\ p{=}None]\!]$
$\Longrightarrow$
$\Gamma\vdash\langle call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c,Normal\ s\rangle\ =Suc\ n\Rightarrow\ Stuck$
⟨proof⟩

lemma *execn-block-exn-Normal-elim* [*consumes 1*]:
assumes *execn-block*: $\Gamma\vdash\langle block\text{-}exn\ init\ bdy\ return\ result\text{-}exn\ c,Normal\ s\rangle\ ={n}\Rightarrow\ t$
assumes *Normal*:
$\bigwedge t'.$
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle\ ={n}\Rightarrow\ Normal\ t';$
$\Gamma\vdash\langle c\ s\ t',Normal\ (return\ s\ t')\rangle\ ={n}\Rightarrow\ t]\!]$
$\Longrightarrow P$
assumes *Abrupt*:
$\bigwedge t'.$
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle\ ={n}\Rightarrow\ Abrupt\ t';$

$t = Abrupt\ (result\text{-}exn\ (return\ s\ t')\ t')]\!]$
$\Longrightarrow P$
assumes *Fault*:
$\bigwedge f.$
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =n\Rightarrow\ Fault\ f;$
$t = Fault\ f]\!]$
$\Longrightarrow P$
assumes *Stuck*:
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =n\Rightarrow\ Stuck;$
$t = Stuck]\!]$
$\Longrightarrow P$
assumes *Undef*:
$[\![\Gamma\ p = None;\ t = Stuck]\!] \Longrightarrow P$
shows P
$\langle proof\rangle$

lemma *execn-block-Normal-elim* [*consumes 1*]:
assumes *execn-block*: $\Gamma\vdash\langle block\ init\ bdy\ return\ c,Normal\ s\rangle =n\Rightarrow\ t$
assumes *Normal*:
$\bigwedge t'.$
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =n\Rightarrow\ Normal\ t';$
$\Gamma\vdash\langle c\ s\ t',Normal\ (return\ s\ t')\rangle =n\Rightarrow\ t]\!]$
$\Longrightarrow P$
assumes *Abrupt*:
$\bigwedge t'.$
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =n\Rightarrow\ Abrupt\ t';$
$t = Abrupt\ (return\ s\ t')]\!]$
$\Longrightarrow P$
assumes *Fault*:
$\bigwedge f.$
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =n\Rightarrow\ Fault\ f;$
$t = Fault\ f]\!]$
$\Longrightarrow P$
assumes *Stuck*:
$[\![\Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =n\Rightarrow\ Stuck;$
$t = Stuck]\!]$
$\Longrightarrow P$
assumes *Undef*:
$[\![\Gamma\ p = None;\ t = Stuck]\!] \Longrightarrow P$
shows P
$\langle proof\rangle$

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

assumes *Abrupt*:
$\bigwedge bdy\ i\ t'.$
$[\![\Gamma\ p = Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =i\Rightarrow\ Abrupt\ t';\ n = Suc\ i;$
$t = Abrupt\ (result\text{-}exn\ (return\ s\ t')\ t')]\!]$
$\Longrightarrow P$
assumes *Fault*:
$\bigwedge bdy\ i\ f.$
$[\![\Gamma\ p = Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =i\Rightarrow\ Fault\ f;\ n = Suc\ i;$
$t = Fault\ f]\!]$
$\Longrightarrow P$
assumes *Stuck*:
$\bigwedge bdy\ i.$
$[\![\Gamma\ p = Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =i\Rightarrow\ Stuck;\ n = Suc\ i;$
$t = Stuck]\!]$
$\Longrightarrow P$
assumes *Undef*:
$\bigwedge i.\ [\![\Gamma\ p = None;\ n = Suc\ i;\ t = Stuck]\!] \Longrightarrow P$
shows *P*
⟨*proof*⟩

lemma *execn-call-Normal-elim* [*consumes 1*]:
assumes *exec-call*: $\Gamma\vdash\langle call\ init\ p\ return\ c,Normal\ s\rangle =n\Rightarrow\ t$
assumes *Normal*:
$\bigwedge bdy\ i\ t'.$
$[\![\Gamma\ p = Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =i\Rightarrow\ Normal\ t';$
$\Gamma\vdash\langle c\ s\ t',Normal\ (return\ s\ t')\rangle =Suc\ i\Rightarrow\ t;\ n = Suc\ i]\!]$
$\Longrightarrow P$
assumes *Abrupt*:
$\bigwedge bdy\ i\ t'.$
$[\![\Gamma\ p = Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =i\Rightarrow\ Abrupt\ t';\ n = Suc\ i;$
$t = Abrupt\ (return\ s\ t')]\!]$
$\Longrightarrow P$
assumes *Fault*:
$\bigwedge bdy\ i\ f.$
$[\![\Gamma\ p = Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =i\Rightarrow\ Fault\ f;\ n = Suc\ i;$
$t = Fault\ f]\!]$
$\Longrightarrow P$
assumes *Stuck*:
$\bigwedge bdy\ i.$
$[\![\Gamma\ p = Some\ bdy;\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle =i\Rightarrow\ Stuck;\ n = Suc\ i;$
$t = Stuck]\!]$
$\Longrightarrow P$
assumes *Undef*:
$\bigwedge i.\ [\![\Gamma\ p = None;\ n = Suc\ i;\ t = Stuck]\!] \Longrightarrow P$
shows *P*
⟨*proof*⟩

lemma *execn-dynCall*:
⟦Γ⊢⟨*call init* (*p s*) *return c*,*Normal s*⟩ =*n*⇒ *t*⟧
⟹
Γ⊢⟨*dynCall init p return c*,*Normal s*⟩ =*n*⇒ *t*
⟨*proof*⟩

lemma *execn-dynCall-exn*:
⟦Γ⊢⟨*call-exn init* (*p s*) *return result-exn c*,*Normal s*⟩ =*n*⇒ *t*⟧
⟹
Γ⊢⟨*dynCall-exn f UNIV init p return result-exn c*,*Normal s*⟩ =*n*⇒ *t*
⟨*proof*⟩

lemma *execn-dynCall-Normal-elim*:
assumes *exec*: Γ⊢⟨*dynCall init p return c*,*Normal s*⟩ =*n*⇒ *t*
assumes Γ⊢⟨*call init* (*p s*) *return c*,*Normal s*⟩ =*n*⇒ *t* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *execn-guards-Normal-elim-cases* [*consumes 1*, *case-names noFault some-Fault*]:
assumes *exec-guards*: Γ⊢⟨*guards gs c*,*Normal s*⟩ =*n*⇒ *t*
assumes *noFault*: ∀*f g*. (*f*, *g*) ∈ *set gs* ⟶ *s* ∈ *g* ⟹ Γ⊢⟨*c*,*Normal s*⟩ =*n*⇒ *t* ⟹ *P*
assumes *someFault*: ⋀*f g*. *find* (λ(*f*,*g*). *s* ∉ *g*) *gs* = *Some* (*f*, *g*) ⟹ *t* = *Fault f* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *execn-maybe-guard-Normal-elim-cases* [*consumes 1*, *case-names noFault someFault*]:
assumes *exec-guards*: Γ⊢⟨*maybe-guard f g c*,*Normal s*⟩ =*n*⇒ *t*
assumes *noFault*: *s* ∈ *g* ⟹ Γ⊢⟨*c*,*Normal s*⟩ =*n*⇒ *t* ⟹ *P*
assumes *someFault*: *s* ∉ *g* ⟹ *t* = *Fault f* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *execn-guards-noFault*:
assumes *exec*: Γ⊢⟨*c*,*Normal s*⟩ =*n*⇒ *t*
assumes *noFault*: ∀*f g*. (*f*, *g*) ∈ *set gs* ⟶ *s* ∈ *g*
shows Γ⊢⟨*guards gs c*,*Normal s*⟩ =*n*⇒ *t*
⟨*proof*⟩

lemma *execn-guards-Fault*:
assumes *Fault*: *find* (λ(*f*,*g*). *s* ∉ *g*) *gs* = *Some* (*f*, *g*)
shows Γ⊢⟨*guards gs c*,*Normal s*⟩ =*n*⇒ *Fault f*
⟨*proof*⟩

lemma *execn-maybe-guard-noFault*:
assumes *exec*: Γ⊢⟨*c*,*Normal s*⟩ =*n*⇒ *t*

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

lemma *execn-maybe-guard-Fault*:
assumes *Fault*: $s \notin g$
shows $\Gamma\vdash\langle$*maybe-guard f g c*,*Normal s*$\rangle$ $=n\Rightarrow$ *Fault f*
⟨*proof*⟩

lemma *execn-guards-DynCom-Normal-elim*:
assumes *exec*: $\Gamma\vdash\langle$*guards gs* (*DynCom c*), *Normal s*$\rangle$ $=n\Rightarrow$ *t*
assumes *call*: $\Gamma\vdash\langle$*guards gs* (*c s*), *Normal s*$\rangle$ $=n\Rightarrow$ *t* $\Longrightarrow$ *P*
shows *P*
⟨*proof*⟩

lemma *execn-maybe-guard-DynCom-Normal-elim*:
assumes *exec*: $\Gamma\vdash\langle$*maybe-guard f g* (*DynCom c*), *Normal s*$\rangle$ $=n\Rightarrow$ *t*
assumes *call*: $\Gamma\vdash\langle$*maybe-guard f g* (*c s*), *Normal s*$\rangle$ $=n\Rightarrow$ *t* $\Longrightarrow$ *P*
shows *P*
⟨*proof*⟩

lemma *execn-guards-DynCom*:
assumes *exec-c*: $\Gamma\vdash\langle$*guards gs* (*c s*), *Normal s*$\rangle$ $=n\Rightarrow$ *t*
shows $\Gamma\vdash\langle$*guards gs* (*DynCom c*), *Normal s*$\rangle$ $=n\Rightarrow$ *t*
⟨*proof*⟩

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

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

lemma *execn-Seq'*:
⟦$\Gamma\vdash\langle$*c1*,*s*$\rangle$ $=n\Rightarrow$ s'; $\Gamma\vdash\langle$*c2*,$s'\rangle$ $=n\Rightarrow$ s''⟧
$\Longrightarrow$
$\Gamma\vdash\langle$*Seq c1 c2*,*s*$\rangle$ $=n\Rightarrow$ s''
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *execn-Suc*:
 $\Gamma\vdash\langle c,s\rangle$ =*n*⇒ *t* ⟹ $\Gamma\vdash\langle c,s\rangle$ =*Suc n*⇒ *t*
 ⟨*proof*⟩

lemma *execn-assoc*:
 $\Gamma\vdash\langle$*Seq c1* (*Seq c2 c3*),*s*⟩ =*n*⇒ *t* = $\Gamma\vdash\langle$*Seq* (*Seq c1 c2*) *c3*,*s*⟩ =*n*⇒ *t*
 ⟨*proof*⟩

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

lemma *exec-to-execn*:
 assumes *execn*: $\Gamma\vdash\langle c,s\rangle$ ⇒ *t*
 shows ∃ *n*. $\Gamma\vdash\langle c,s\rangle$ =*n*⇒ *t*
⟨*proof*⟩

theorem *exec-iff-execn*: ($\Gamma\vdash\langle c,s\rangle$ ⇒ *t*) = (∃ *n*. $\Gamma\vdash\langle c,s\rangle$ =*n*⇒ *t*)
 ⟨*proof*⟩

definition *nfinal-notin*:: (*'s,'p,'f*) *body* ⇒ (*'s,'p,'f*) *com* ⇒ (*'s,'f*) *xstate* ⇒ *nat*
 ⇒ (*'s,'f*) *xstate set* ⇒ *bool*
 (‹-⊢ ⟨-,-⟩ =-⇒∉-› [*60*,*20*,*98*,*65*,*60*] *89*) **where**
Γ⊢ ⟨*c*,*s*⟩ =*n*⇒∉*T* = (∀ *t*. Γ⊢ ⟨*c*,*s*⟩ =*n*⇒ *t* ⟶ *t*∉*T*)

definition *final-notin*:: (*'s,'p,'f*) *body* ⇒ (*'s,'p,'f*) *com* ⇒ (*'s,'f*) *xstate*
 ⇒ (*'s,'f*) *xstate set* ⇒ *bool*
 (‹-⊢ ⟨-,-⟩ ⇒∉-› [*60*,*20*,*98*,*60*] *89*) **where**
Γ⊢ ⟨*c*,*s*⟩ ⇒∉*T* = (∀ *t*. Γ⊢ ⟨*c*,*s*⟩ ⇒*t* ⟶ *t*∉*T*)

lemma *final-notinI*: ⟦⋀*t*. Γ⊢⟨*c*,*s*⟩ ⇒ *t* ⟹ *t* ∉ *T*⟧ ⟹ Γ⊢⟨*c*,*s*⟩ ⇒∉*T*
 ⟨*proof*⟩

lemma *noFaultStuck-Call-body'*: *p* ∈ *dom* Γ ⟹
Γ⊢⟨*Call p*,*Normal s*⟩ ⇒∉({*Stuck*} ∪ *Fault* ‘ (−*F*)) =
Γ⊢⟨*the* (Γ *p*),*Normal s*⟩ ⇒∉({*Stuck*} ∪ *Fault* ‘ (−*F*))
 ⟨*proof*⟩

lemma *noFault-startn*:
 assumes *execn*: Γ⊢⟨*c*,*s*⟩ =*n*⇒ *t* **and** *t*: *t*≠*Fault f*
 shows *s*≠*Fault f*
⟨*proof*⟩

lemma *noFault-start*:

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

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

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

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'
⟨*proof*⟩

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'
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

$\langle proof \rangle$

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

lemma *noFaultnI*: $[\![\bigwedge t.\ \Gamma\vdash\langle c,s\rangle =n\Rightarrow t \Longrightarrow t\neq Fault\ f]\!] \Longrightarrow \Gamma\vdash\langle c,s\rangle =n\Rightarrow\notin\{Fault\ f\}$
$\langle proof \rangle$

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

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

lemma *noStucknI*: $[\![\bigwedge t.\ \Gamma\vdash\langle c,s\rangle =n\Rightarrow t \Longrightarrow t\neq Stuck]\!] \Longrightarrow \Gamma\vdash\langle c,s\rangle =n\Rightarrow\notin\{Stuck\}$
$\langle proof \rangle$

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

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

lemma *noFaultI*: $[\![\bigwedge t.\ \Gamma\vdash\langle c,s\rangle \Rightarrow t \Longrightarrow t\neq Fault\ f]\!] \Longrightarrow \Gamma\vdash\langle c,s\rangle \Rightarrow\notin\{Fault\ f\}$
$\langle proof \rangle$

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

lemma *noFaultE*:
$[\![\Gamma\vdash\langle c,s\rangle \Rightarrow\notin\{Fault\ f\};\ \Gamma\vdash\langle c,s\rangle \Rightarrow Fault\ f]\!] \Longrightarrow P$
$\langle proof \rangle$

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

lemma *noStuckI*: $[\![\bigwedge t.\ \Gamma\vdash\langle c,s\rangle \Rightarrow t \Longrightarrow t\neq Stuck]\!] \Longrightarrow \Gamma\vdash\langle c,s\rangle \Rightarrow\notin\{Stuck\}$
$\langle proof \rangle$

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*:
$[\![\Gamma\vdash\langle c,s\rangle \Rightarrow\notin\{Stuck\};\ \Gamma\vdash\langle c,s\rangle \Rightarrow Stuck]\!] \Longrightarrow P$
⟨*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*: $[\![\Gamma\vdash\langle c,s\rangle =n\Rightarrow\notin\{Fault\ f\};\ \Gamma\vdash\langle c,s\rangle =n\Rightarrow t]\!] \Longrightarrow t\neq Fault\ f$
⟨*proof*⟩

lemma *noFault-execD*: $[\![\Gamma\vdash\langle c,s\rangle \Rightarrow\notin\{Fault\ f\};\ \Gamma\vdash\langle c,s\rangle \Rightarrow t]\!] \Longrightarrow t\neq Fault\ f$
⟨*proof*⟩

lemma *noFaultn-exec-startD*: $[\![\Gamma\vdash\langle c,s\rangle =n\Rightarrow\notin\{Fault\ f\};\ \Gamma\vdash\langle c,s\rangle =n\Rightarrow t]\!] \Longrightarrow s\neq Fault$
f
⟨*proof*⟩

lemma *noFault-exec-startD*: $[\![\Gamma\vdash\langle c,s\rangle \Rightarrow\notin\{Fault\ f\};\ \Gamma\vdash\langle c,s\rangle \Rightarrow t]\!] \Longrightarrow s\neq Fault\ f$
⟨*proof*⟩

lemma *noStuckn-execD*: $[\![\Gamma\vdash\langle c,s\rangle =n\Rightarrow\notin\{Stuck\};\ \Gamma\vdash\langle c,s\rangle =n\Rightarrow t]\!] \Longrightarrow t\neq Stuck$
⟨*proof*⟩

lemma *noStuck-execD*: $[\![\Gamma\vdash\langle c,s\rangle \Rightarrow\notin\{Stuck\};\ \Gamma\vdash\langle c,s\rangle \Rightarrow t]\!] \Longrightarrow t\neq Stuck$
⟨*proof*⟩

lemma *noStuckn-exec-startD*: $[\![\Gamma\vdash\langle c,s\rangle =n\Rightarrow\notin\{Stuck\};\ \Gamma\vdash\langle c,s\rangle =n\Rightarrow t]\!] \Longrightarrow s\neq Stuck$
⟨*proof*⟩

lemma *noStuck-exec-startD*: $[\![\Gamma\vdash\langle c,s\rangle \Rightarrow\notin\{Stuck\};\ \Gamma\vdash\langle c,s\rangle \Rightarrow t]\!] \Longrightarrow s\neq Stuck$
⟨*proof*⟩

lemma *noFaultStuckn-execD*:
$[\![\Gamma\vdash\langle c,s\rangle =n\Rightarrow\notin\{Fault\ True,Fault\ False,Stuck\};\ \Gamma\vdash\langle c,s\rangle =n\Rightarrow t]\!] \Longrightarrow$
$t\notin\{Fault\ True,Fault\ False,Stuck\}$
⟨*proof*⟩

lemma *noFaultStuck-execD*: $[\![\Gamma\vdash\langle c,s\rangle \Rightarrow\notin\{Fault\ True,Fault\ False,Stuck\};\ \Gamma\vdash\langle c,s\rangle$
$\Rightarrow t]\!]$
$\Longrightarrow t\notin\{Fault\ True,Fault\ False,Stuck\}$
⟨*proof*⟩

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

$\Longrightarrow$ $s \notin \{Fault\ True, Fault\ False, Stuck\}$
⟨proof⟩

lemma *noFaultStuck-exec-startD*:
$[\![\Gamma\vdash\langle c,s\rangle \Rightarrow\notin\{Fault\ True,\ Fault\ False, Stuck\};\ \Gamma\vdash\langle c,s\rangle \Rightarrow t]\!]$
$\Longrightarrow$ $s \notin \{Fault\ True, Fault\ False, Stuck\}$
⟨proof⟩

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

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 $s \in g$
⟨proof⟩

lemma *final-notin-to-finaln*:
assumes *notin*: $\Gamma\vdash\langle c,s\rangle \Rightarrow\notin T$
shows $\Gamma\vdash\langle c,s\rangle =n\Rightarrow\notin T$
⟨proof⟩

lemma *noFault-Call-body*:
$\Gamma\ p=Some\ bdy\Longrightarrow$
$\Gamma\vdash\langle Call\ p\ , Normal\ s\rangle \Rightarrow\notin\{Fault\ f\} =$
$\Gamma\vdash\langle the\ (\Gamma\ p), Normal\ s\rangle \Rightarrow\notin\{Fault\ f\}$
⟨proof⟩

lemma *noStuck-Call-body*:
$\Gamma\ p=Some\ bdy\Longrightarrow$
$\Gamma\vdash\langle Call\ p, Normal\ s\rangle \Rightarrow\notin\{Stuck\} =$
$\Gamma\vdash\langle the\ (\Gamma\ p), Normal\ s\rangle \Rightarrow\notin\{Stuck\}$
⟨proof⟩

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

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

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

lemma *Seq-NoFaultStuckD2*:
assumes *noabort*: $\Gamma\vdash\langle Seq\ c1\ c2,s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ F)$

shows $\forall t.\ \Gamma\vdash\langle c1,s\rangle \Rightarrow t \longrightarrow t\notin (\{Stuck\} \cup Fault\ `\ F) \longrightarrow$
$\Gamma\vdash\langle c2,t\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ F)$
$\langle proof\rangle$ **lemma** *Seq-NoFaultStuckD1*:
assumes *noabort*: $\Gamma\vdash\langle Seq\ c1\ c2,s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ F)$
shows $\Gamma\vdash\langle c1,s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ F)$
$\langle proof\rangle$

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

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

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

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

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

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 $\bigwedge s\ t.\ \Gamma\vdash\langle sequence\ Seq\ (flatten\ c),s\rangle =n\Rightarrow t \Longrightarrow \Gamma\vdash\langle c,s\rangle =n\Rightarrow t$
$\langle proof\rangle$

lemma *execn-normalize-to-execn*:
shows $\bigwedge s\ t\ n.\ \Gamma\vdash\langle normalize\ c,s\rangle =n\Rightarrow t \Longrightarrow \Gamma\vdash\langle c,s\rangle =n\Rightarrow t$
⟨*proof*⟩

lemma *execn-normalize-iff-execn*:
$\Gamma\vdash\langle normalize\ c,s\rangle =n\Rightarrow t = \Gamma\vdash\langle c,s\rangle =n\Rightarrow t$
⟨*proof*⟩

lemma *exec-sequence-app*:
assumes *exec-xs*: $\Gamma\vdash\langle sequence\ Seq\ xs,Normal\ s\rangle \Rightarrow s'$
assumes *exec-ys*: $\Gamma\vdash\langle sequence\ Seq\ ys,s'\rangle \Rightarrow t$
shows $\Gamma\vdash\langle sequence\ Seq\ (xs@ys),Normal\ s\rangle \Rightarrow t$
⟨*proof*⟩

lemma *exec-sequence-appD*:
assumes *exec-xs-ys*: $\Gamma\vdash\langle sequence\ Seq\ (xs\ @\ ys),Normal\ s\rangle \Rightarrow t$
shows $\exists\, s'.\ \Gamma\vdash\langle sequence\ Seq\ xs,Normal\ s\rangle \Rightarrow s' \wedge \Gamma\vdash\langle sequence\ Seq\ ys,s'\rangle \Rightarrow t$
⟨*proof*⟩

lemma *exec-sequence-appE* [*consumes 1*]:
$[\![\Gamma\vdash\langle sequence\ Seq\ (xs\ @\ ys),Normal\ s\rangle \Rightarrow t;$
$\bigwedge s'.\ [\![\Gamma\vdash\langle sequence\ Seq\ xs,Normal\ s\rangle \Rightarrow s';\Gamma\vdash\langle sequence\ Seq\ ys,s'\rangle \Rightarrow t]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
⟨*proof*⟩

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

lemma *exec-sequence-flatten-to-exec*:
assumes *exec-seq*: $\Gamma\vdash\langle sequence\ Seq\ (flatten\ c),s\rangle \Rightarrow t$
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 $\Gamma\vdash\langle normalize\ c,s\rangle \Rightarrow t$
⟨*proof*⟩

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

lemma *exec-normalize-iff-exec*:
$\Gamma\vdash\langle normalize\ c,s\rangle \Rightarrow t = \Gamma\vdash\langle c,s\rangle \Rightarrow t$
⟨*proof*⟩

3.4 Lemmas about $c_1 \subseteq_g c_2$

lemma *execn-to-execn-subseteq-guards*: $\bigwedge c\ s\ t\ n.\ [\![c \subseteq_g c'; \Gamma\vdash\langle c,s\rangle =n\Rightarrow t]\!]$
$\Longrightarrow \exists t'.\ \Gamma\vdash\langle c',s\rangle =n\Rightarrow t' \wedge$
$(isFault\ t \longrightarrow isFault\ t') \wedge (\neg\ isFault\ t' \longrightarrow t'=t)$
⟨*proof*⟩

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

3.5 Lemmas about *merge-guards*

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

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

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

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

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

corollary *exec-to-exec-merge-guards*:
$\Gamma\vdash\langle c,\ s\rangle \Rightarrow t \Longrightarrow \Gamma\vdash\langle merge\text{-}guards\ c,s\rangle \Rightarrow t$
⟨*proof*⟩

corollary *exec-merge-guards-to-exec*:
$\Gamma\vdash\langle merge\text{-}guards\ c,s\rangle \Rightarrow t \Longrightarrow \Gamma\vdash\langle c,\ s\rangle \Rightarrow t$
⟨*proof*⟩

3.6 Lemmas about *mark-guards*

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

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

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

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

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

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

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

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*: $\neg\ isFault\ t$
shows $\Gamma\vdash\langle strip\text{-}guards\ F\ c,s\rangle =n\Rightarrow t$
$\langle proof\rangle$

lemma *execn-to-execn-strip-guards-Fault*:
assumes *exec-c*: $\Gamma\vdash\langle c,s\rangle =n\Rightarrow t$

shows $\bigwedge f.\ [\![t=Fault\ f;\ f \notin F]\!] \Longrightarrow \Gamma\vdash\langle strip\text{-}guards\ F\ c,s\rangle =n\Rightarrow Fault\ f$
$\langle proof\rangle$

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

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

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

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

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

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

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

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

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

lemma *exec-to-exec-strip*:
assumes *exec-c*: $\Gamma\vdash\langle c,s\rangle \Rightarrow t$
assumes *t-not-Fault*: ¬ *isFault t*
shows *strip F* $\Gamma\vdash\langle c,s\rangle \Rightarrow t$
⟨*proof*⟩

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

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

3.8 Lemmas about $c_1 \cap_g c_2$

lemma *inter-guards-execn-Normal-noFault*:
⋀*c c2 s t n*. ⟦(*c1* $\cap_g$ *c2*) = *Some c*; $\Gamma\vdash\langle c,$*Normal s*$\rangle =n\Rightarrow t$; ¬ *isFault t*⟧
⟹ $\Gamma\vdash\langle$*c1*,*Normal s*$\rangle =n\Rightarrow t \wedge \Gamma\vdash\langle$*c2*,*Normal s*$\rangle =n\Rightarrow t$
⟨*proof*⟩

lemma *inter-guards-execn-noFault*:
assumes *c*: (*c1* $\cap_g$ *c2*) = *Some c*
assumes *exec-c*: $\Gamma\vdash\langle c,s\rangle =n\Rightarrow t$
assumes *noFault*: ¬ *isFault t*
shows $\Gamma\vdash\langle$*c1*,$s\rangle =n\Rightarrow t \wedge \Gamma\vdash\langle$*c2*,$s\rangle =n\Rightarrow t$

⟨*proof*⟩

lemma *inter-guards-exec-noFault*:
assumes *c*: $(c1 \cap_g c2) = Some\ c$
assumes *exec-c*: $\Gamma\vdash\langle c,s\rangle \Rightarrow t$
assumes *noFault*: $\neg\ isFault\ t$
shows $\Gamma\vdash\langle c1,s\rangle \Rightarrow t \wedge \Gamma\vdash\langle c2,s\rangle \Rightarrow t$
⟨*proof*⟩

lemma *inter-guards-execn-Normal-Fault*:
$\bigwedge c\ c2\ s\ n.\ [\![(c1 \cap_g c2) = Some\ c;\ \Gamma\vdash\langle c,Normal\ s\rangle =n\Rightarrow Fault\ f]\!]$
$\Longrightarrow (\Gamma\vdash\langle c1,Normal\ s\rangle =n\Rightarrow Fault\ f \vee \Gamma\vdash\langle c2,Normal\ s\rangle =n\Rightarrow Fault\ f)$
⟨*proof*⟩

lemma *inter-guards-execn-Fault*:
assumes *c*: $(c1 \cap_g c2) = Some\ c$
assumes *exec-c*: $\Gamma\vdash\langle c,s\rangle =n\Rightarrow Fault\ f$
shows $\Gamma\vdash\langle c1,s\rangle =n\Rightarrow Fault\ f \vee \Gamma\vdash\langle c2,s\rangle =n\Rightarrow Fault\ f$
⟨*proof*⟩

lemma *inter-guards-exec-Fault*:
assumes *c*: $(c1 \cap_g c2) = Some\ c$
assumes *exec-c*: $\Gamma\vdash\langle c,s\rangle \Rightarrow Fault\ f$
shows $\Gamma\vdash\langle c1,s\rangle \Rightarrow Fault\ f \vee \Gamma\vdash\langle c2,s\rangle \Rightarrow Fault\ f$
⟨*proof*⟩

3.9 Restriction of Procedure Environment

lemma *restrict-SomeD*: $(m|_A)\ x = Some\ y \Longrightarrow m\ x = Some\ y$
⟨*proof*⟩

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

lemma *restrict-in-dom*: $x \in A \Longrightarrow (m|_A)\ x = m\ x$
⟨*proof*⟩

lemma *exec-restrict-to-exec*:
assumes *exec-restrict*: $\Gamma|_A\vdash\langle c,s\rangle \Rightarrow t$
assumes *notStuck*: $t\neq Stuck$
shows $\Gamma\vdash\langle c,s\rangle \Rightarrow t$
⟨*proof*⟩

lemma *execn-restrict-to-execn*:
assumes *exec-restrict*: $\Gamma|_A\vdash\langle c,s\rangle =n\Rightarrow t$

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

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

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

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

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

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

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

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

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

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

lemma *notStuck-WhileTrueD1*:

$[\![\Gamma\vdash\langle \mathit{While}\ b\ c,\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\};\ s \in b]\!]$
$\Longrightarrow \Gamma\vdash\langle c,\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\}$
⟨*proof*⟩

lemma *notStuck-WhileTrueD2*:
$[\![\Gamma\vdash\langle \mathit{While}\ b\ c,\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\};\ \Gamma\vdash\langle c,\mathit{Normal}\ s\rangle \Rightarrow s';\ s \in b]\!]$
$\Longrightarrow \Gamma\vdash\langle \mathit{While}\ b\ c,s'\rangle \Rightarrow\notin\{\mathit{Stuck}\}$
⟨*proof*⟩

lemma *notStuck-CallD*:
$[\![\Gamma\vdash\langle \mathit{Call}\ p\ ,\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\};\ \Gamma\ p = \mathit{Some}\ \mathit{bdy}]\!]$
$\Longrightarrow \Gamma\vdash\langle \mathit{bdy},\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\}$
⟨*proof*⟩

lemma *notStuck-CallDefinedD*:
$[\![\Gamma\vdash\langle \mathit{Call}\ p,\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\}]\!]$
$\Longrightarrow \Gamma\ p \neq \mathit{None}$
⟨*proof*⟩

lemma *notStuck-DynComD*:
$[\![\Gamma\vdash\langle \mathit{DynCom}\ c,\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\}]\!]$
$\Longrightarrow \Gamma\vdash\langle (c\ s),\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\}$
⟨*proof*⟩

lemma *notStuck-CatchD1*:
$[\![\Gamma\vdash\langle \mathit{Catch}\ c1\ c2,\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\}]\!] \Longrightarrow \Gamma\vdash\langle c1,\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\}$
⟨*proof*⟩

lemma *notStuck-CatchD2*:
$[\![\Gamma\vdash\langle \mathit{Catch}\ c1\ c2,\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\};\ \Gamma\vdash\langle c1,\mathit{Normal}\ s\rangle \Rightarrow \mathit{Abrupt}\ s']\!]$
$\Longrightarrow \Gamma\vdash\langle c2,\mathit{Normal}\ s'\rangle \Rightarrow\notin\{\mathit{Stuck}\}$
⟨*proof*⟩

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-Γ*: $\forall p \in \mathit{dom}\ \Gamma.\ \mathit{noguards}\ (\mathit{the}\ (\Gamma\ p))$
assumes *s-no-Fault*: ¬ *isFault s*
shows ¬ *isFault t*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *execn-nothrows-no-Abrupt*:
assumes *execn*: $\Gamma\vdash\langle c,s\rangle =n\Rightarrow t$
assumes *nothrows-c*: *nothrows c*
assumes *nothrows-Γ*: $\forall p \in dom\ \Gamma.\ nothrows\ (the\ (\Gamma\ p))$
assumes *s-no-Abrupt*: $\neg(isAbr\ s)$
shows $\neg(isAbr\ t)$
⟨*proof*⟩

lemma *exec-nothrows-no-Abrupt*:
assumes *exec*: $\Gamma\vdash\langle c,s\rangle \Rightarrow t$
assumes *nothrows-c*: *nothrows c*
assumes *nothrows-Γ*: $\forall p \in dom\ \Gamma.\ nothrows\ (the\ (\Gamma\ p))$
assumes *s-no-Abrupt*: $\neg(isAbr\ s)$
shows $\neg(isAbr\ t)$
⟨*proof*⟩

end

4 Hoare Logic for Partial Correctness

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

type-synonym $('s,'p)\ quadruple = ('s\ assn \times 'p \times 's\ assn \times 's\ assn)$

4.1 Validity of Hoare Tuples: $\Gamma,\Theta\models_{/F} P\ c\ Q,A$

definition
valid :: $[('s,'p,'f)\ body,'f\ set,'s\ assn,('s,'p,'f)\ com,'s\ assn,'s\ assn] => bool$
(‹-⊨'/-/ - - -,-› [61,60,1000, 20, 1000,1000] 60)
where
$\Gamma\models_{/F} P\ c\ Q,A \equiv \forall s\ t.\ \Gamma\vdash\langle c,s\rangle \Rightarrow t \longrightarrow s \in Normal\ `\ P \longrightarrow t \notin Fault\ `\ F$
$\longrightarrow\ t \in\ Normal\ `\ Q \cup Abrupt\ `\ A$

definition
cvalid::
$[('s,'p,'f)\ body,('s,'p)\ quadruple\ set,'f\ set,$
$'s\ assn,('s,'p,'f)\ com,'s\ assn,'s\ assn] =>bool$
(‹-,-⊨'/-/ - - -,-› [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 :: $[('s,'p,'f)\ body,nat,'f\ set,$
$'s\ assn,('s,'p,'f)\ com,'s\ assn,'s\ assn] => bool$
(‹-⊨-:'/-/ - - -,-› [61,60,60,1000, 20, 1000,1000] 60)

where
$\Gamma \models n{:}_{/F}\ P\ c\ Q,A \equiv \forall\, s\ t.\ \Gamma \vdash \langle c,s\ \rangle = n \Rightarrow t \longrightarrow s \in Normal\ `\ P \longrightarrow t \notin Fault\ `\ F \longrightarrow t \in\ Normal\ `\ Q \cup Abrupt\ `\ A$

definition
cnvalid::
$[('s,'p,'f)\ body,('s,'p)\ quadruple\ set,nat,'f\ set,$
$'s\ assn,('s,'p,'f)\ com,'s\ assn,'s\ assn] \Rightarrow bool$
(‹-,-⊨-:$_{/-}$/ - - -,-› [61,60,60,60,1000, 20, 1000,1000] 60)
where
$\Gamma,\Theta \models n{:}_{/F}\ P\ c\ Q,A \equiv (\forall\,(P,p,Q,A){\in}\Theta.\ \Gamma \models n{:}_{/F}\ P\ (Call\ p)\ Q,A) \longrightarrow \Gamma\ \models n{:}_{/F}\ P\ c\ Q,A$

notation (*ASCII*)
valid (‹-|='/-/ - - -,-› [61,60,1000, 20, 1000,1000] 60) **and**
cvalid (‹-,-|='/-/ - - -,-› [61,60,60,1000, 20, 1000,1000] 60) **and**
nvalid (‹-|=-:'/-/ - - -,-› [61,60,60,1000, 20, 1000,1000] 60) **and**
cnvalid (‹-,-|=-:'/-/ - - -,-› [61,60,60,60,1000, 20, 1000,1000] 60)

4.2 Properties of Validity

lemma *valid-iff-nvalid*: $\Gamma \models_{/F}\ P\ c\ Q,A = (\forall\, n.\ \Gamma \models n{:}_{/F}\ P\ c\ Q,A)$
⟨*proof*⟩

lemma *cnvalid-to-cvalid*: $(\forall\, n.\ \Gamma,\Theta \models n{:}_{/F}\ P\ c\ Q,A) \Longrightarrow \Gamma,\Theta \models_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *nvalidI*:
$[\![\bigwedge s\ t.\ [\![\Gamma \vdash \langle c,Normal\ s\ \rangle = n \Rightarrow t;s \in P;\ t \notin\ Fault\ `\ F]\!] \Longrightarrow t \in Normal\ `\ Q \cup Abrupt\ `\ A]\!]$
$\Longrightarrow \Gamma \models n{:}_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *validI*:
$[\![\bigwedge s\ t.\ [\![\Gamma \vdash \langle c,Normal\ s\ \rangle \Rightarrow t;s \in P;\ t \notin Fault\ `\ F]\!] \Longrightarrow t \in Normal\ `\ Q \cup Abrupt\ `\ A]\!]$
$\Longrightarrow \Gamma \models_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *cvalidI*:
$[\![\bigwedge s\ t.\ [\![\forall\,(P,p,Q,A){\in}\Theta.\ \Gamma \models_{/F}\ P\ (Call\ p)\ Q,A;\Gamma \vdash \langle c,Normal\ s\rangle \Rightarrow t;s \in P;t \notin Fault\ `\ F]\!]$
$\Longrightarrow t \in Normal\ `\ Q \cup Abrupt\ `\ A]\!]$
$\Longrightarrow \Gamma,\Theta \models_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *cvalidD*:
$[\![\Gamma,\Theta\models_{/F} P\ c\ Q,A;\forall (P,p,Q,A)\in\Theta.\ \Gamma\models_{/F} P\ (Call\ p)\ Q,A;\Gamma\vdash\langle c,Normal\ s\rangle \Rightarrow t;s \in P;t\notin Fault\ `\ F]\!]$
$\Longrightarrow t \in Normal\ `\ Q \cup Abrupt\ `\ A$
$\langle proof\rangle$

lemma *cnvalidI*:
$[\![\bigwedge s\ t.\ [\![\forall (P,p,Q,A)\in\Theta.\ \Gamma\models n{:}_{/F} P\ (Call\ p)\ Q,A;$
$\Gamma\vdash\langle c,Normal\ s\ \rangle =n\Rightarrow t;s \in P;t\notin Fault\ `\ F]\!]$
$\Longrightarrow t \in Normal\ `\ Q \cup Abrupt\ `\ A]\!]$
$\Longrightarrow \Gamma,\Theta\models n{:}_{/F} P\ c\ Q,A$
$\langle proof\rangle$

lemma *cnvalidD*:
$[\![\Gamma,\Theta\models n{:}_{/F} P\ c\ Q,A;\forall (P,p,Q,A)\in\Theta.\ \Gamma\models n{:}_{/F} P\ (Call\ p)\ Q,A;$
$\Gamma\vdash\langle c,Normal\ s\ \rangle =n\Rightarrow t;s \in P;$
$t\notin Fault\ `\ F]\!]$
$\Longrightarrow t \in Normal\ `\ Q \cup Abrupt\ `\ A$
$\langle proof\rangle$

lemma *nvalid-augment-Faults*:
assumes *validn*:$\Gamma\models n{:}_{/F} P\ c\ Q,A$
assumes F': $F \subseteq F'$
shows $\Gamma\models n{:}_{/F'} P\ c\ Q,A$
$\langle proof\rangle$

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

lemma *nvalid-to-nvalid-strip*:
assumes *validn*:$\Gamma\models n{:}_{/F} P\ c\ Q,A$
assumes F': $F' \subseteq -F$
shows *strip* $F'\ \Gamma\models n{:}_{/F} P\ c\ Q,A$
$\langle proof\rangle$

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

4.3 The Hoare Rules: $\Gamma,\Theta\vdash_{/F}\ P\ c\ Q,A$

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

inductive *hoarep*::[('s,'p,'f) body,('s,'p) quadruple set,'f set,
'*s assn*,('s,'p,'f) *com*, '*s assn*,'*s assn*] => *bool*
(‹(3-,-/⊢'/- (-/ (-)/ -,/-))› [60,60,60,1000,20,1000,1000]60)
for Γ::('s,'p,'f) *body*
where
Skip: $\Gamma,\Theta\vdash_{/F}\ Q\ Skip\ Q,A$

| *Basic*: $\Gamma,\Theta\vdash_{/F}\ \{s.\ f\ s \in Q\}\ (Basic\ f)\ Q,A$

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

| *Seq*: $[\![\Gamma,\Theta\vdash_{/F}\ P\ c_1\ R,A;\ \Gamma,\Theta\vdash_{/F}\ R\ c_2\ Q,A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\ P\ (Seq\ c_1\ c_2)\ Q,A$

| *Cond*: $[\![\Gamma,\Theta\vdash_{/F}\ (P \cap b)\ c_1\ Q,A;\ \Gamma,\Theta\vdash_{/F}\ (P \cap -\ b)\ c_2\ Q,A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\ P\ (Cond\ b\ c_1\ c_2)\ Q,A$

| *While*: $\Gamma,\Theta\vdash_{/F}\ (P \cap b)\ c\ P,A$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\ P\ (While\ b\ c)\ (P \cap -\ b),A$

| *Guard*: $\Gamma,\Theta\vdash_{/F}\ (g \cap P)\ c\ Q,A$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\ (g \cap P)\ (Guard\ f\ g\ c)\ Q,A$

| *Guarantee*: $[\![f \in F;\ \Gamma,\Theta\vdash_{/F}\ (g \cap P)\ c\ Q,A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\ P\ (Guard\ f\ g\ c)\ Q,A$

| *CallRec*:
$[\![(P,p,Q,A) \in Specs;$
$\forall (P,p,Q,A) \in Specs.\ p \in dom\ \Gamma \wedge \Gamma,\Theta \cup Specs\vdash_{/F}\ P\ (the\ (\Gamma\ p))\ Q,A\ ]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (Call\ p)\ Q,A$

| *DynCom*:
$\forall s \in P.\ \Gamma,\Theta\vdash_{/F}\ P\ (c\ s)\ Q,A$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\ P\ (DynCom\ c)\ Q,A$

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

| *Catch*: $[\![\Gamma,\Theta\vdash_{/F} P\ c_1\ Q,R;\ \Gamma,\Theta\vdash_{/F} R\ c_2\ Q,A]\!] \Longrightarrow\ \Gamma,\Theta\vdash_{/F} P\ Catch\ c_1\ c_2\ Q,A$

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

| *Asm*: $[\![(P,p,Q,A) \in \Theta]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F} P\ (Call\ p)\ Q,A$

| *ExFalso*: $[\![\forall n.\ \Gamma,\Theta\models n{:}_{/F}\ P\ c\ Q,A;\ \neg\ \Gamma\models_{/F} P\ c\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F} P\ c\ Q,A$
— This is a hack rule that enables us to derive completeness for an arbitrary context Θ, from completeness for an empty context.

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

lemma *hoare-strip-*Γ:
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 $\bigwedge\Theta'.\ \Theta \subseteq \Theta' \Longrightarrow \Gamma,\Theta'\vdash_{/F} P\ p\ Q,A$
⟨*proof*⟩

4.4 Some Derived Rules

lemma *Conseq'*: $\forall s.\ s \in P \longrightarrow$
$(\exists P'\ Q'\ A'.$
$(\forall\ Z.\ \Gamma,\Theta\vdash_{/F} (P'\ Z)\ c\ (Q'\ Z),(A'\ Z)) \wedge$
$(\exists Z.\ s \in P'\ Z \wedge (Q'\ Z \subseteq Q) \wedge (A'\ Z \subseteq A)))$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F} P\ c\ Q,A$
⟨*proof*⟩

lemma *conseq*:$[\![\forall Z.\ \Gamma,\Theta \vdash_{/F} (P'\ Z)\ c\ (Q'\ Z),(A'\ Z);$
$\forall s.\ s \in P \longrightarrow (\exists\ Z.\ s{\in}P'\ Z \wedge (Q'\ Z \subseteq Q) \wedge (A'\ Z \subseteq A))]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F} P\ c\ Q,A$
⟨*proof*⟩

theorem *conseqPrePost* [*trans*]:

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

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

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

lemma *CallRec'*:
$[\![p\in Procs;\ Procs \subseteq\ dom\ \Gamma;$
$\forall p\in Procs.$
$\forall Z.\ \Gamma,\Theta \cup (\bigcup p\in Procs.\ \bigcup Z.\ \{((P\ p\ Z),p,Q\ p\ Z,A\ p\ Z)\})$
$\vdash_{/F} (P\ p\ Z)\ (the\ (\Gamma\ p))\ (Q\ p\ Z),(A\ p\ Z)]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F} (P\ p\ Z)\ (Call\ p)\ (Q\ p\ Z),(A\ p\ Z)$
⟨*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*: $\Gamma,\Theta\vdash_{/F} P\ c\ Q,A \Longrightarrow \Gamma,\Theta\models_{/F} P\ c\ Q,A$
⟨*proof*⟩

5.2 Completeness

lemma *MGT-valid*:
$\Gamma\models_{/F}\{s.\ s{=}Z \wedge \Gamma\vdash\langle c,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup\ \ Fault\ `\ (-F))\}\ c$
$\{t.\ \Gamma\vdash\langle c,Normal\ Z\rangle \Rightarrow\ Normal\ t\},\ \{t.\ \Gamma\vdash\langle c,Normal\ Z\rangle \Rightarrow\ Abrupt\ t\}$
⟨*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*: $\bigwedge s.\ s \in P \Longrightarrow s \in P'\ s \wedge (\forall t.\ t \in Q'\ s \longrightarrow t \in Q) \wedge$
$(\forall t.\ t \in A'\ s \longrightarrow t \in A)$
shows $\Gamma,\Theta \vdash_{/F} P\ c\ Q,A$
⟨*proof*⟩

lemma *Seq-NoFaultStuckD1*:
assumes *noabort*: $\Gamma \vdash \langle Seq\ c1\ c2,s\rangle \Rightarrow \notin(\{Stuck\} \cup Fault\ `\ F)$
shows $\Gamma \vdash \langle c1,s\rangle \Rightarrow \notin(\{Stuck\} \cup Fault\ `\ F)$
⟨*proof*⟩

lemma *Seq-NoFaultStuckD2*:
assumes *noabort*: $\Gamma \vdash \langle Seq\ c1\ c2,s\rangle \Rightarrow \notin(\{Stuck\} \cup Fault\ `\ F)$
shows $\forall t.\ \Gamma \vdash \langle c1,s\rangle \Rightarrow t \longrightarrow t \notin (\{Stuck\} \cup Fault\ `\ F) \longrightarrow$
$\Gamma \vdash \langle c2,t\rangle \Rightarrow \notin(\{Stuck\} \cup Fault\ `\ F)$
⟨*proof*⟩

lemma *MGT-implies-complete*:
assumes *MGT*: $\forall Z.\ \Gamma,\{\}\vdash_{/F} \{s.\ s=Z \wedge \Gamma \vdash \langle c,Normal\ s\rangle \Rightarrow \notin(\{Stuck\} \cup Fault\ `\ (-F))\}\ c$
$\{t.\ \Gamma \vdash \langle c,Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma \vdash \langle c,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
assumes *valid*: $\Gamma \models_{/F} P\ c\ Q,A$
shows $\Gamma,\{\} \vdash_{/F} P\ c\ Q,A$
⟨*proof*⟩

Equipped only with the classic consequence rule $[\![?\Gamma, ?\Theta\vdash_{/?F}\ ?P'\ ?c\ ?Q',?A';\ ?P \subseteq\ ?P';\ ?Q' \subseteq\ ?Q;\ ?A' \subseteq\ ?A]\!] \Longrightarrow\ ?\Gamma,?\Theta\vdash_{/?F}\ ?P\ ?c\ ?Q,?A$ 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*: $\forall Z.\ \Gamma,\{\}\vdash_{/F}$
$\{s.\ s=Z \wedge \Gamma \vdash \langle c,Normal\ s\rangle \Rightarrow \notin(\{Stuck\} \cup Fault\ `\ (-F))\}\ c$
$\{t.\ \Gamma \vdash \langle c,Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma \vdash \langle c,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
assumes *valid*: $\Gamma \models_{/F} P\ c\ Q,A$
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*⟩

Semantic equivalence of both kind of formulations

lemma *valid-involved-to-valid*:
assumes *valid*:
$\forall Z.\ \Gamma\models_{/F} \{s.\ s=Z \wedge s \in P\}\ c\ \{t.\ Z \in P \longrightarrow t \in Q\},\{t.\ Z \in P \longrightarrow t \in A\}$
shows $\Gamma \models_{/F} P\ c\ Q,A$
⟨*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*:
$\Gamma \models_{/F} P\ c\ Q,A \Longrightarrow$
$\Gamma\models_{/F} \{s.\ s=Z \wedge s \in P\}\ c\ \{t.\ Z \in P \longrightarrow t \in Q\},\{t.\ Z \in P \longrightarrow t \in A\}$
⟨*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*: $R \Longrightarrow \Gamma,\Theta\vdash_{/F} P\ c\ Q,A$
shows $\Gamma,\Theta\vdash_{/F} P\ c\ Q,A$
⟨*proof*⟩

lemma *MGT-lemma*:
assumes *MGT-Calls*:
$\forall p\in dom\ \Gamma.\ \forall Z.\ \Gamma,\Theta \vdash_{/F}$
$\{s.\ s=Z \wedge \Gamma\vdash\langle Call\ p,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F))\}$
$(Call\ p)$
$\{t.\ \Gamma\vdash\langle Call\ p,Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma\vdash\langle Call\ p,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
shows $\bigwedge Z.\ \Gamma,\Theta\vdash_{/F} \{s.\ s=Z \wedge \Gamma\vdash\langle c,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F))\}\ c$
$\{t.\ \Gamma\vdash\langle c,Normal\ Z\rangle \Rightarrow Normal\ t\},\{t.\ \Gamma\vdash\langle c,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
⟨*proof*⟩

lemma *MGT-Calls*:
$\forall p\in dom\ \Gamma.\ \forall Z.$
$\Gamma,\{\}\vdash_{/F}\{s.\ s=Z \wedge \Gamma\vdash\langle Call\ p,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F))\}$
$(Call\ p)$
$\{t.\ \Gamma\vdash\langle Call\ p,Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma\vdash\langle Call\ p,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
⟨*proof*⟩

theorem *hoare-complete*: $\Gamma\models_{/F} P\ c\ Q,A \Longrightarrow \Gamma,\{\}\vdash_{/F} P\ c\ Q,A$

⟨*proof*⟩

lemma *hoare-complete'*:
assumes *cvalid*: $\forall n.\ \Gamma,\Theta \models n{:}_{/F}\ P\ c\ Q,A$
shows $\Gamma,\Theta \vdash_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *hoare-strip-*Γ:
assumes *deriv*: $\Gamma,\{\}\vdash_{/F}\ P\ p\ Q,A$
assumes F': $F' \subseteq -F$
shows *strip* $F'\ \Gamma,\{\}\vdash_{/F}\ P\ p\ Q,A$
⟨*proof*⟩

5.3 And Now: Some Useful Rules

5.3.1 Consequence

lemma *LiberalConseq-sound*:
fixes $F{::}'f\ set$
assumes *cons*: $\forall s \in P.\ \forall (t{::}('s,'f)\ xstate).\ \exists P'\ Q'\ A'.\ (\forall n.\ \Gamma,\Theta \models n{:}_{/F}\ P'\ c\ Q',A')$
$\wedge$
$((s \in P' \longrightarrow t \in Normal\ `\ Q' \cup Abrupt\ `\ A')$
$\longrightarrow t \in Normal\ `\ Q \cup Abrupt\ `\ A)$
shows $\Gamma,\Theta \models n{:}_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *LiberalConseq*:
fixes $F{::}\ 'f\ set$
assumes *cons*: $\forall s \in P.\ \forall (t{::}('s,'f)\ xstate).\ \exists P'\ Q'\ A'.\ \Gamma,\Theta \vdash_{/F}\ P'\ c\ Q',A' \wedge$
$((s \in P' \longrightarrow t \in Normal\ `\ Q' \cup Abrupt\ `\ A')$
$\longrightarrow t \in Normal\ `\ Q \cup Abrupt\ `\ A)$
shows $\Gamma,\Theta \vdash_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma $\forall s \in P.\ \exists P'\ Q'\ A'.\ \Gamma,\Theta \vdash_{/F}\ P'\ c\ Q',A' \wedge s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A$
$\Longrightarrow \Gamma,\Theta \vdash_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma
fixes $F{::}\ 'f\ set$
assumes *cons*: $\forall s \in P.\ \exists P'\ Q'\ A'.\ \Gamma,\Theta \vdash_{/F}\ P'\ c\ Q',A' \wedge$
$(\forall (t{::}('s,'f)\ xstate).\ (s \in P' \longrightarrow t \in Normal\ `\ Q' \cup Abrupt\ `\ A')$
$\longrightarrow t \in Normal\ `\ Q \cup Abrupt\ `\ A)$
shows $\Gamma,\Theta \vdash_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *LiberalConseq'*:

fixes F:: $'f$ *set*
assumes *cons*: $\forall s \in P.\ \ \exists P'\ Q'\ A'.\ \Gamma,\Theta \vdash_{/F} P'\ c\ Q',A' \wedge$
$(\forall (t::('s,'f)\ xstate).\ (s \in P' \longrightarrow t \in Normal\ `\ Q' \cup Abrupt\ `\ A')$
$\longrightarrow t \in Normal\ `\ Q \cup Abrupt\ `\ A)$
shows $\Gamma,\Theta \vdash_{/F} P\ c\ Q,A$
$\langle proof \rangle$

lemma *LiberalConseq''*:
fixes F:: $'f$ *set*
assumes *spec*: $\forall Z.\ \Gamma,\Theta \vdash_{/F} (P'\ Z)\ c\ (Q'\ Z),(A'\ Z)$
assumes *cons*: $\forall s\ (t::('s,'f)\ xstate).$
$(\forall Z.\ s \in P'\ Z \longrightarrow t \in Normal\ `\ Q'\ Z \cup Abrupt\ `\ A'\ Z)$
$\longrightarrow (s \in P \longrightarrow t \in Normal\ `\ Q \cup Abrupt\ `\ A)$
shows $\Gamma,\Theta \vdash_{/F} P\ c\ Q,A$
$\langle proof \rangle$

primrec *procs*:: $('s,'p,'f)\ com \Rightarrow\ 'p\ set$
where
$procs\ Skip = \{\}\ |$
$procs\ (Basic\ f) = \{\}\ |$
$procs\ (Seq\ c_1\ c_2)\ = (procs\ c_1 \cup procs\ c_2)\ |$
$procs\ (Cond\ b\ c_1\ c_2) = (procs\ c_1 \cup procs\ c_2)\ |$
$procs\ (While\ b\ c) = procs\ c\ |$
$procs\ (Call\ p) = \{p\}\ |$
$procs\ (DynCom\ c) = (\bigcup s.\ procs\ (c\ s))\ |$
$procs\ (Guard\ f\ g\ c) = procs\ c\ |$
$procs\ Throw = \{\}\ |$
$procs\ (Catch\ c_1\ c_2) = (procs\ c_1 \cup procs\ c_2)$

primrec *noSpec*:: $('s,'p,'f)\ com \Rightarrow bool$
where
$noSpec\ Skip = True\ |$
$noSpec\ (Basic\ f) = True\ |$
$noSpec\ (Spec\ r) = False\ |$
$noSpec\ (Seq\ c_1\ c_2)\ = (noSpec\ c_1 \wedge noSpec\ c_2)\ |$
$noSpec\ (Cond\ b\ c_1\ c_2) = (noSpec\ c_1 \wedge noSpec\ c_2)\ |$
$noSpec\ (While\ b\ c) = noSpec\ c\ |$
$noSpec\ (Call\ p) = True\ |$
$noSpec\ (DynCom\ c) = (\forall s.\ noSpec\ (c\ s))\ |$
$noSpec\ (Guard\ f\ g\ c) = noSpec\ c\ |$
$noSpec\ Throw = True\ |$
$noSpec\ (Catch\ c_1\ c_2) = (noSpec\ c_1 \wedge noSpec\ c_2)$

lemma *exec-noSpec-no-Stuck*:
assumes *exec*: $\Gamma \vdash \langle c,s \rangle \Rightarrow t$
assumes *noSpec-c*: *noSpec c*
assumes *noSpec-*Γ: $\forall p \in dom\ \Gamma.\ noSpec\ (the\ (\Gamma\ p))$
assumes *procs-subset*: $procs\ c \subseteq dom\ \Gamma$
assumes *procs-subset-*Γ: $\forall p \in dom\ \Gamma.\ procs\ (the\ (\Gamma\ p)) \subseteq dom\ \Gamma$

assumes *s-no-Stuck*: $s \neq Stuck$
shows $t \neq Stuck$
⟨*proof*⟩

lemma *execn-noSpec-no-Stuck*:
assumes *exec*: $\Gamma\vdash\langle c,s\rangle =n\Rightarrow t$
assumes *noSpec-c*: *noSpec c*
assumes *noSpec*-Γ: $\forall p \in dom\ \Gamma.\ noSpec\ (the\ (\Gamma\ p))$
assumes *procs-subset*: $procs\ c \subseteq dom\ \Gamma$
assumes *procs-subset*-Γ: $\forall p \in dom\ \Gamma.\ procs\ (the\ (\Gamma\ p)) \subseteq dom\ \Gamma$
assumes *s-no-Stuck*: $s \neq Stuck$
shows $t \neq Stuck$
⟨*proof*⟩

lemma *LiberalConseq-noguards-nothrows-sound*:
assumes *spec*: $\forall Z.\ \forall n.\ \Gamma,\Theta\models n{:}_{/F}\ (P'\ Z)\ c\ (Q'\ Z),(A'\ Z)$
assumes *cons*: $\forall s\ t.\ (\forall Z.\ s \in P'\ Z \longrightarrow t \in\ Q'\ Z\ )$
$\longrightarrow (s \in P \longrightarrow t \in Q\ )$
assumes *noguards-c*: *noguards c*
assumes *noguards*-Γ: $\forall p \in dom\ \Gamma.\ noguards\ (the\ (\Gamma\ p))$
assumes *nothrows-c*: *nothrows c*
assumes *nothrows*-Γ: $\forall p \in dom\ \Gamma.\ nothrows\ (the\ (\Gamma\ p))$
assumes *noSpec-c*: *noSpec c*
assumes *noSpec*-Γ: $\forall p \in dom\ \Gamma.\ noSpec\ (the\ (\Gamma\ p))$
assumes *procs-subset*: $procs\ c \subseteq dom\ \Gamma$
assumes *procs-subset*-Γ: $\forall p \in dom\ \Gamma.\ procs\ (the\ (\Gamma\ p)) \subseteq dom\ \Gamma$
shows $\Gamma,\Theta\models n{:}_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *LiberalConseq-noguards-nothrows*:
assumes *spec*: $\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ c\ (Q'\ Z),(A'\ Z)$
assumes *cons*: $\forall s\ t.\ (\forall Z.\ s \in P'\ Z \longrightarrow t \in\ Q'\ Z\ )$
$\longrightarrow (s \in P \longrightarrow t \in Q\ )$
assumes *noguards-c*: *noguards c*
assumes *noguards*-Γ: $\forall p \in dom\ \Gamma.\ noguards\ (the\ (\Gamma\ p))$
assumes *nothrows-c*: *nothrows c*
assumes *nothrows*-Γ: $\forall p \in dom\ \Gamma.\ nothrows\ (the\ (\Gamma\ p))$
assumes *noSpec-c*: *noSpec c*
assumes *noSpec*-Γ: $\forall p \in dom\ \Gamma.\ noSpec\ (the\ (\Gamma\ p))$
assumes *procs-subset*: $procs\ c \subseteq dom\ \Gamma$
assumes *procs-subset*-Γ: $\forall p \in dom\ \Gamma.\ procs\ (the\ (\Gamma\ p)) \subseteq dom\ \Gamma$
shows $\Gamma,\Theta\vdash_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma
assumes *spec*: $\forall Z.\ \Gamma,\Theta\vdash_{/F}\{s.\ s=fst\ Z \wedge P\ s\ (snd\ Z)\}\ c\ \{t.\ Q\ (fst\ Z)\ (snd\ Z)\ t\},\{\}$

assumes *noguards-c*: *noguards c*
assumes *noguards-*Γ: $\forall p \in dom\ \Gamma.\ noguards\ (the\ (\Gamma\ p))$
assumes *nothrows-c*: *nothrows c*
assumes *nothrows-*Γ: $\forall p \in dom\ \Gamma.\ nothrows\ (the\ (\Gamma\ p))$
assumes *noSpec-c*: *noSpec c*
assumes *noSpec-*Γ: $\forall p \in dom\ \Gamma.\ noSpec\ (the\ (\Gamma\ p))$
assumes *procs-subset*: $procs\ c \subseteq dom\ \Gamma$
assumes *procs-subset-*Γ: $\forall p \in dom\ \Gamma.\ procs\ (the\ (\Gamma\ p)) \subseteq dom\ \Gamma$
shows $\forall \sigma.\ \Gamma,\Theta \vdash_{/F} \{s.\ s=\sigma\}\ c\ \{t.\ \forall l.\ P\ \sigma\ l \longrightarrow Q\ \sigma\ l\ t\},\{\}$
⟨*proof*⟩

5.3.2 Modify Return

lemma *Proc-exnModifyReturn-sound*:
  assumes *valid-call*: $\forall n.\ \Gamma,\Theta \models n{:}_{/F}\ P\ call\text{-}exn\ init\ p\ return'\ result\text{-}exn\ c\ Q,A$
  assumes *valid-modif*:
    $\forall \sigma.\ \forall n.\ \Gamma,\Theta \models n{:}_{/UNIV}\ \{\sigma\}\ Call\ p\ (Modif\ \sigma),(ModifAbr\ \sigma)$
  assumes *ret-modif*:
    $\forall s\ t.\ t \in Modif\ (init\ s)$
        $\longrightarrow return'\ s\ t = return\ s\ t$
  assumes *ret-modifAbr*: $\forall s\ t.\ t \in ModifAbr\ (init\ s)$
        $\longrightarrow result\text{-}exn\ (return'\ s\ t)\ t = result\text{-}exn\ (return\ s\ t)\ t$
  shows $\Gamma,\Theta \models n{:}_{/F}\ P\ (call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *ProcModifyReturn-sound*:
  assumes *valid-call*: $\forall n.\ \Gamma,\Theta \models n{:}_{/F}\ P\ call\ init\ p\ return'\ c\ Q,A$
  assumes *valid-modif*:
    $\forall \sigma.\ \forall n.\ \Gamma,\Theta \models n{:}_{/UNIV}\ \{\sigma\}\ Call\ p\ (Modif\ \sigma),(ModifAbr\ \sigma)$
  assumes *ret-modif*:
    $\forall s\ t.\ t \in Modif\ (init\ s)$
        $\longrightarrow return'\ s\ t = return\ s\ t$
  assumes *ret-modifAbr*: $\forall s\ t.\ t \in ModifAbr\ (init\ s)$
        $\longrightarrow return'\ s\ t = return\ s\ t$
  shows $\Gamma,\Theta \models n{:}_{/F}\ P\ (call\ init\ p\ return\ c)\ Q,A$
  ⟨*proof*⟩

lemma *Proc-exnModifyReturn*:
  assumes *spec*: $\Gamma,\Theta \vdash_{/F}\ P\ (call\text{-}exn\ init\ p\ return'\ result\text{-}exn\ c)\ Q,A$
  assumes *result-conform*:
    $\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow (return'\ s\ t) = (return\ s\ t)$
  assumes *return-conform*:
    $\forall s\ t.\ t \in ModifAbr\ (init\ s)$
        $\longrightarrow (result\text{-}exn\ (return'\ s\ t)\ t) = (result\text{-}exn\ (return\ s\ t)\ t)$
  assumes *modifies-spec*:
  $\forall \sigma.\ \Gamma,\Theta \vdash_{/UNIV}\ \{\sigma\}\ Call\ p\ (Modif\ \sigma),(ModifAbr\ \sigma)$
  shows $\Gamma,\Theta \vdash_{/F}\ P\ (call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *ProcModifyReturn*:
assumes *spec*: $\Gamma,\Theta\vdash_{/F}$ P (*call init p return′ c*) Q,A
assumes *result-conform*:
$\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow (return'\ s\ t) = (return\ s\ t)$
assumes *return-conform*:
$\forall s\ t.\ t \in ModifAbr\ (init\ s)$
$\longrightarrow (return'\ s\ t) = (return\ s\ t)$
assumes *modifies-spec*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/UNIV} \{\sigma\}\ Call\ p\ (Modif\ \sigma),(ModifAbr\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F}$ P (*call init p return c*) Q,A
⟨*proof*⟩

lemma *Proc-exnModifyReturnSameFaults-sound*:
assumes *valid-call*: $\forall n.\ \Gamma,\Theta \models n{:}_{/F}\ P$ *call-exn init p return′ result-exn c* Q,A
assumes *valid-modif*:
$\forall \sigma.\ \forall n.\ \Gamma,\Theta\models n{:}_{/F}\ \{\sigma\}\ Call\ p\ (Modif\ \sigma),(ModifAbr\ \sigma)$
assumes *ret-modif*:
$\forall s\ t.\ t \in Modif\ (init\ s)$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *ret-modifAbr*: $\forall s\ t.\ t \in ModifAbr\ (init\ s)$
$\longrightarrow result\text{-}exn\ (return'\ s\ t)\ t = result\text{-}exn\ (return\ s\ t)\ t$
shows $\Gamma,\Theta \models n{:}_{/F}\ P$ (*call-exn init p return result-exn c*) Q,A
⟨*proof*⟩

lemma *ProcModifyReturnSameFaults-sound*:
assumes *valid-call*: $\forall n.\ \Gamma,\Theta \models n{:}_{/F}\ P$ *call init p return′ c* Q,A
assumes *valid-modif*:
$\forall \sigma.\ \forall n.\ \Gamma,\Theta\models n{:}_{/F}\ \{\sigma\}\ Call\ p\ (Modif\ \sigma),(ModifAbr\ \sigma)$
assumes *ret-modif*:
$\forall s\ t.\ t \in Modif\ (init\ s)$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *ret-modifAbr*: $\forall s\ t.\ t \in ModifAbr\ (init\ s)$
$\longrightarrow return'\ s\ t = return\ s\ t$
shows $\Gamma,\Theta \models n{:}_{/F}\ P$ (*call init p return c*) Q,A
⟨*proof*⟩

lemma *Proc-exnModifyReturnSameFaults*:
assumes *spec*: $\Gamma,\Theta\vdash_{/F}$ P (*call-exn init p return′ result-exn c*) Q,A
assumes *result-conform*:
$\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow (return'\ s\ t) = (return\ s\ t)$
assumes *return-conform*:
$\forall s\ t.\ t \in ModifAbr\ (init\ s) \longrightarrow (result\text{-}exn\ (return'\ s\ t)\ t) = (result\text{-}exn\ (return\ s\ t)\ t)$
assumes *modifies-spec*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ Call\ p\ (Modif\ \sigma),(ModifAbr\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F}$ P (*call-exn init p return result-exn c*) Q,A

⟨*proof*⟩

lemma *ProcModifyReturnSameFaults*:
  assumes *spec*: $\Gamma,\Theta\vdash_{/F}$ *P* (*call init p return′ c*) *Q*,*A*
  assumes *result-conform*:
    $\forall s\ t.\ t \in$ *Modif* (*init s*) $\longrightarrow$ (*return′ s t*) = (*return s t*)
  assumes *return-conform*:
  $\forall s\ t.\ t \in$ *ModifAbr* (*init s*) $\longrightarrow$ (*return′ s t*) = (*return s t*)
  assumes *modifies-spec*:
  $\forall \sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}$ *Call p* (*Modif* σ),(*ModifAbr* σ)
shows $\Gamma,\Theta\vdash_{/F}$ *P* (*call init p return c*) *Q*,*A*
  ⟨*proof*⟩

5.3.3 DynCall

lemma *dynProc-exnModifyReturn-sound*:
assumes *valid-call*: $\bigwedge n.\ \Gamma,\Theta \models n{:}_{/F}$ *P dynCall-exn f g init p return′ result-exn c Q*,*A*
assumes *valid-modif*:
  $\forall s \in P.\ \forall \sigma.\ \forall n.$
    $\Gamma,\Theta\models n{:}_{/UNIV} \{\sigma\}$ *Call* (*p s*) (*Modif* σ),(*ModifAbr* σ)
assumes *ret-modif*:
  $\forall s\ t.\ t \in$ *Modif* (*init s*)
    $\longrightarrow$ *return′ s t* = *return s t*
assumes *ret-modifAbr*: $\forall s\ t.\ t \in$ *ModifAbr* (*init s*)
    $\longrightarrow$ *result-exn* (*return′ s t*) *t* = *result-exn* (*return s t*) *t*
shows $\Gamma,\Theta \models n{:}_{/F}$ *P* (*dynCall-exn f g init p return result-exn c*) *Q*,*A*
⟨*proof*⟩

lemma *dynProcModifyReturn-sound*:
assumes *valid-call*: $\bigwedge n.\ \Gamma,\Theta \models n{:}_{/F}$ *P dynCall init p return′ c Q*,*A*
assumes *valid-modif*:
  $\forall s \in P.\ \forall \sigma.\ \forall n.$
    $\Gamma,\Theta\models n{:}_{/UNIV} \{\sigma\}$ *Call* (*p s*) (*Modif* σ),(*ModifAbr* σ)
assumes *ret-modif*:
  $\forall s\ t.\ t \in$ *Modif* (*init s*)
    $\longrightarrow$ *return′ s t* = *return s t*
assumes *ret-modifAbr*: $\forall s\ t.\ t \in$ *ModifAbr* (*init s*)
    $\longrightarrow$ *return′ s t* = *return s t*
shows $\Gamma,\Theta \models n{:}_{/F}$ *P* (*dynCall init p return c*) *Q*,*A*
  ⟨*proof*⟩

lemma *dynProc-exnModifyReturn*:
assumes *dyn-call*: $\Gamma,\Theta\vdash_{/F}$ *P dynCall-exn f g init p return′ result-exn c Q*,*A*
assumes *ret-modif*:
  $\forall s\ t.\ t \in$ *Modif* (*init s*)

$\longrightarrow$ *return′ s t* = *return s t*
assumes *ret-modifAbr*: $\forall$ *s t. t* $\in$ *ModifAbr* (*init s*)
$\longrightarrow$ *result-exn* (*return′ s t*) *t* = *result-exn* (*return s t*) *t*
assumes *modif*:
$\forall s \in P. \forall \sigma.$
$\Gamma,\Theta\vdash_{/UNIV} \{\sigma\}$ *Call* (*p s*) (*Modif* σ),(*ModifAbr* σ)
shows $\Gamma,\Theta\vdash_{/F}$ *P* (*dynCall-exn f g init p return result-exn c*) *Q*,*A*
⟨*proof*⟩

lemma *dynProcModifyReturn*:
assumes *dyn-call*: $\Gamma,\Theta\vdash_{/F}$ *P dynCall init p return′ c Q*,*A*
assumes *ret-modif*:
$\forall$ *s t. t* $\in$ *Modif* (*init s*)
$\longrightarrow$ *return′ s t* = *return s t*
assumes *ret-modifAbr*: $\forall$ *s t. t* $\in$ *ModifAbr* (*init s*)
$\longrightarrow$ *return′ s t* = *return s t*
assumes *modif*:
$\forall s \in P. \forall \sigma.$
$\Gamma,\Theta\vdash_{/UNIV} \{\sigma\}$ *Call* (*p s*) (*Modif* σ),(*ModifAbr* σ)
shows $\Gamma,\Theta\vdash_{/F}$ *P* (*dynCall init p return c*) *Q*,*A*
⟨*proof*⟩

lemma *dynProc-exnModifyReturnSameFaults-sound*:
assumes *valid-call*: $\bigwedge n.\ \Gamma,\Theta \models n{:}_{/F}$ *P dynCall-exn f g init p return′ result-exn c Q*,*A*
assumes *valid-modif*:
$\forall s \in P. \forall \sigma. \forall n.$
$\Gamma,\Theta\models n{:}_{/F} \{\sigma\}$ *Call* (*p s*) (*Modif* σ),(*ModifAbr* σ)
assumes *ret-modif*:
$\forall$ *s t. t* $\in$ *Modif* (*init s*) $\longrightarrow$ *return′ s t* = *return s t*
assumes *ret-modifAbr*: $\forall$ *s t. t* $\in$ *ModifAbr* (*init s*) $\longrightarrow$ *result-exn* (*return′ s t*) *t* = *result-exn* (*return s t*) *t*
shows $\Gamma,\Theta \models n{:}_{/F}$ *P* (*dynCall-exn f g init p return result-exn c*) *Q*,*A*
⟨*proof*⟩

lemma *dynProcModifyReturnSameFaults-sound*:
assumes *valid-call*: $\bigwedge n.\ \Gamma,\Theta \models n{:}_{/F}$ *P dynCall init p return′ c Q*,*A*
assumes *valid-modif*:
$\forall s \in P. \forall \sigma. \forall n.$
$\Gamma,\Theta\models n{:}_{/F} \{\sigma\}$ *Call* (*p s*) (*Modif* σ),(*ModifAbr* σ)
assumes *ret-modif*:
$\forall$ *s t. t* $\in$ *Modif* (*init s*) $\longrightarrow$ *return′ s t* = *return s t*
assumes *ret-modifAbr*: $\forall$ *s t. t* $\in$ *ModifAbr* (*init s*) $\longrightarrow$ *return′ s t* = *return s t*
shows $\Gamma,\Theta \models n{:}_{/F}$ *P* (*dynCall init p return c*) *Q*,*A*
⟨*proof*⟩

lemma *dynProc-exnModifyReturnSameFaults*:

assumes *dyn-call*: $\Gamma,\Theta \vdash_{/F}$ *P dynCall-exn f g init p return′ result-exn c Q*,*A*
assumes *ret-modif*:
  ∀ *s t*. *t* ∈ *Modif* (*init s*)
    ⟶ *return′ s t* = *return s t*
assumes *ret-modifAbr*: ∀ *s t*. *t* ∈ *ModifAbr* (*init s*)
    ⟶ *result-exn* (*return′ s t*) *t* = *result-exn* (*return s t*) *t*
assumes *modif*:
  ∀ *s* ∈ *P*. ∀ σ. $\Gamma,\Theta \vdash_{/F}$ {σ} *Call* (*p s*) (*Modif* σ),(*ModifAbr* σ)
shows $\Gamma,\Theta \vdash_{/F}$ *P* (*dynCall-exn f g init p return result-exn c*) *Q*,*A*
⟨*proof*⟩

lemma *dynProcModifyReturnSameFaults*:
assumes *dyn-call*: $\Gamma,\Theta \vdash_{/F}$ *P dynCall init p return′ c Q*,*A*
assumes *ret-modif*:
  ∀ *s t*. *t* ∈ *Modif* (*init s*)
    ⟶ *return′ s t* = *return s t*
assumes *ret-modifAbr*: ∀ *s t*. *t* ∈ *ModifAbr* (*init s*)
    ⟶ *return′ s t* = *return s t*
assumes *modif*:
  ∀ *s* ∈ *P*. ∀ σ. $\Gamma,\Theta \vdash_{/F}$ {σ} *Call* (*p s*) (*Modif* σ),(*ModifAbr* σ)
  shows $\Gamma,\Theta \vdash_{/F}$ *P* (*dynCall init p return c*) *Q*,*A*
  ⟨*proof*⟩

5.3.4 Conjunction of Postcondition

lemma *PostConjI-sound*:
assumes *valid-Q*: ∀ *n*. $\Gamma,\Theta \models n{:}_{/F}$ *P c Q*,*A*
assumes *valid-R*: ∀ *n*. $\Gamma,\Theta \models n{:}_{/F}$ *P c R*,*B*
shows $\Gamma,\Theta \models n{:}_{/F}$ *P c* (*Q* ∩ *R*),(*A* ∩ *B*)
⟨*proof*⟩

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* ∩ *R*),(*A* ∩ *B*)
⟨*proof*⟩

lemma *Merge-PostConj-sound*:
  assumes *validF*: ∀ *n*. $\Gamma,\Theta \models n{:}_{/F}$ *P c Q*,*A*
  assumes *validG*: ∀ *n*. $\Gamma,\Theta \models n{:}_{/G}$ *P′ c R*,*X*
  assumes *F-G*: *F* ⊆ *G*
  assumes *P-P′*: *P* ⊆ *P′*
  shows $\Gamma,\Theta \models n{:}_{/F}$ *P c* (*Q* ∩ *R*),(*A* ∩ *X*)
⟨*proof*⟩

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*: $\forall n.\ \Gamma,\Theta\models n{:}_{/\{\}}\ P\ (strip\text{-}guards\ (-F)\ c)\ UNIV,UNIV$
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*: $\Gamma,\Theta\vdash_{/\{\}}\ P\ (strip\text{-}guards\ (-F)\ c)\ UNIV,UNIV$
shows $\Gamma,\Theta\vdash_{/\{\}}\ P\ c\ Q,A$

⟨*proof*⟩

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 $\Gamma,\Theta \models n:_{/\{\}}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *GuardsFlip*:
assumes *deriv*: $\Gamma,\Theta \vdash_{/F}\ P\ c\ Q,A$
assumes *derivFlip*: $\Gamma,\Theta \vdash_{/-F}\ P\ c\ UNIV,UNIV$
shows $\Gamma,\Theta \vdash_{/\{\}}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *MarkGuardsI-sound*:
assumes *valid*: $\forall n.\ \Gamma,\Theta \models n:_{/\{\}}\ P\ c\ Q,A$
shows $\Gamma,\Theta \models n:_{/\{\}}\ P\ \textit{mark-guards}\ f\ c\ Q,A$
⟨*proof*⟩

lemma *MarkGuardsI*:
assumes *deriv*: $\Gamma,\Theta \vdash_{/\{\}}\ P\ c\ Q,A$
shows $\Gamma,\Theta \vdash_{/\{\}}\ P\ \textit{mark-guards}\ f\ c\ Q,A$
⟨*proof*⟩

lemma *MarkGuardsD-sound*:
assumes *valid*: $\forall n.\ \Gamma,\Theta \models n:_{/\{\}}\ P\ \textit{mark-guards}\ f\ c\ Q,A$
shows $\Gamma,\Theta \models n:_{/\{\}}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *MarkGuardsD*:
assumes *deriv*: $\Gamma,\Theta \vdash_{/\{\}}\ P\ \textit{mark-guards}\ f\ c\ Q,A$
shows $\Gamma,\Theta \vdash_{/\{\}}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *MergeGuardsI-sound*:
assumes *valid*: $\forall n.\ \Gamma,\Theta \models n:_{/F}\ P\ c\ Q,A$
shows $\Gamma,\Theta \models n:_{/F}\ P\ \textit{merge-guards}\ c\ Q,A$
⟨*proof*⟩

lemma *MergeGuardsI*:
assumes *deriv*: $\Gamma,\Theta \vdash_{/F}\ P\ c\ Q,A$
shows $\Gamma,\Theta \vdash_{/F}\ P\ \textit{merge-guards}\ c\ Q,A$
⟨*proof*⟩

lemma *MergeGuardsD-sound*:
assumes *valid*: $\forall n.\ \Gamma,\Theta \models n:_{/F}\ P\ \textit{merge-guards}\ c\ Q,A$

shows $\Gamma,\Theta \models n{:}_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *MergeGuardsD*:
assumes *deriv*: $\Gamma,\Theta \vdash_{/F} P\ \mathit{merge\text{-}guards}\ c\ Q,A$
shows $\Gamma,\Theta \vdash_{/F} P\ c\ Q,A$
⟨*proof*⟩

lemma *SubsetGuards-sound*:
assumes *c-c′*: $c \subseteq_g c'$
assumes *valid*: $\forall n.\ \Gamma,\Theta \models n{:}_{/\{\}}\ P\ c'\ Q,A$
shows $\Gamma,\Theta \models n{:}_{/\{\}}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *SubsetGuards*:
assumes *c-c′*: $c \subseteq_g c'$
assumes *deriv*: $\Gamma,\Theta \vdash_{/\{\}} P\ c'\ Q,A$
shows $\Gamma,\Theta \vdash_{/\{\}} P\ c\ Q,A$
⟨*proof*⟩

lemma *NormalizeD-sound*:
assumes *valid*: $\forall n.\ \Gamma,\Theta \models n{:}_{/F}\ P\ (\mathit{normalize}\ c)\ Q,A$
shows $\Gamma,\Theta \models n{:}_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *NormalizeD*:
assumes *deriv*: $\Gamma,\Theta \vdash_{/F} P\ (\mathit{normalize}\ c)\ Q,A$
shows $\Gamma,\Theta \vdash_{/F} P\ c\ Q,A$
⟨*proof*⟩

lemma *NormalizeI-sound*:
assumes *valid*: $\forall n.\ \Gamma,\Theta \models n{:}_{/F}\ P\ c\ Q,A$
shows $\Gamma,\Theta \models n{:}_{/F}\ P\ (\mathit{normalize}\ c)\ Q,A$
⟨*proof*⟩

lemma *NormalizeI*:
assumes *deriv*: $\Gamma,\Theta \vdash_{/F} P\ c\ Q,A$
shows $\Gamma,\Theta \vdash_{/F} P\ (\mathit{normalize}\ c)\ Q,A$
⟨*proof*⟩

5.3.7 Restricting the Procedure Environment

lemma *nvalid-restrict-to-nvalid*:
assumes *valid-c*: $\Gamma|_M \models n{:}_{/F}\ P\ c\ Q,A$
shows $\Gamma \models n{:}_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *valid-restrict-to-valid*:
assumes *valid-c*: $\Gamma|_M \models_{/F} P\ c\ Q,A$
shows $\Gamma \models_{/F} P\ c\ Q,A$
⟨*proof*⟩

lemma *augment-procs*:
assumes *deriv-c*: $\Gamma|_M,\{\}\vdash_{/F} P\ c\ Q,A$
shows $\Gamma,\{\}\vdash_{/F} P\ c\ Q,A$
⟨*proof*⟩

lemma *augment-Faults*:
assumes *deriv-c*: $\Gamma,\{\}\vdash_{/F} P\ c\ Q,A$
assumes *F*: $F \subseteq F'$
shows $\Gamma,\{\}\vdash_{/F'} P\ c\ Q,A$
⟨*proof*⟩

end

6 Derived Hoare Rules for Partial Correctness

theory *HoarePartial* **imports** *HoarePartialProps* **begin**

lemma *conseq-no-aux*:
$[\![\Gamma,\Theta\vdash_{/F} P'\ c\ Q',A';$
$\forall s.\ s \in P \longrightarrow (s{\in}P' \wedge (Q' \subseteq Q) \wedge (A' \subseteq A))]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F} P\ c\ Q,A$
⟨*proof*⟩

lemma *conseq-exploit-pre*:
$[\![\forall s \in P.\ \Gamma,\Theta\vdash_{/F} (\{s\} \cap P)\ c\ Q,A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F} P\ c\ Q,A$
⟨*proof*⟩

lemma *conseq*:$[\![\forall Z.\ \Gamma,\Theta\vdash_{/F} (P'\ Z)\ c\ (Q'\ Z),(A'\ Z);$
$\forall s.\ s \in P \longrightarrow (\exists\ Z.\ s{\in}P'\ Z \wedge (Q'\ Z \subseteq Q) \wedge (A'\ Z \subseteq A))]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F} P\ c\ Q,A$
⟨*proof*⟩

lemma *Lem*: $[\![\forall Z.\ \Gamma,\Theta\vdash_{/F} (P'\ Z)\ c\ (Q'\ Z),(A'\ Z);$
$P \subseteq \{s.\ \exists\ Z.\ s{\in}P'\ Z \wedge (Q'\ Z \subseteq Q) \wedge (A'\ Z \subseteq A)\}]\!]$
$\Longrightarrow$

$\Gamma,\Theta\vdash_{/F}\ P\ (lem\ x\ c)\ Q,A$
⟨*proof*⟩

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*: $\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ c\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (lem\ x\ c)\ Q,A$
⟨*proof*⟩

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*: $\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ c\ (Q'\ Z),\{\}$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (lem\ x\ c)\ Q,\{\}$
⟨*proof*⟩

lemma *TrivPost*: $\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ c\ (Q'\ Z),(A'\ Z)$
$\Longrightarrow$
$\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ c\ UNIV,UNIV$
⟨*proof*⟩

lemma *TrivPostNoAbr*: $\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ c\ (Q'\ Z),\{\}$
$\Longrightarrow$
$\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ c\ UNIV,\{\}$
⟨*proof*⟩

lemma *conseq-under-new-pre*:$[\![\Gamma,\Theta\vdash_{/F}\ P'\ c\ Q',A';$
$\forall s \in P.\ s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *conseq-Kleymann*:$[\![\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ c\ (Q'\ Z),(A'\ Z);$
$\forall s \in P.\ (\exists Z.\ s{\in}P'\ Z \wedge (Q'\ Z \subseteq Q) \wedge (A'\ Z \subseteq A))]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *DynComConseq*:
assumes $P \subseteq \{s.\ \exists P'\ Q'\ A'.\ \ \Gamma,\Theta\vdash_{/F}\ P'\ (c\ s)\ Q',A' \wedge P \subseteq P' \wedge Q' \subseteq Q \wedge A'$
$\subseteq A\}$
shows $\Gamma,\Theta\vdash_{/F}\ P\ DynCom\ c\ Q,A$
⟨*proof*⟩

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*: $\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ (c\ Z)\ (Q'\ Z),(A'\ Z)$
assumes *bdy-constant*: $\ \forall Z.\ c\ Z = c\ undefined$

shows $\Gamma,\Theta\vdash_{/F}$ P (*specAnno* P' c Q' A') Q,A
⟨*proof*⟩

lemma *SpecAnno'*:
$[\![P \subseteq \{s.\ \exists\ Z.\ s{\in}P'\ Z\ \wedge$
$(\forall\,t.\ t \in Q'\ Z \longrightarrow\ t \in Q) \wedge (\forall\,t.\ t \in A'\ Z \longrightarrow t \in\ A)\};$
$\forall\,Z.\ \Gamma,\Theta\vdash_{/F} (P'\ Z)\ (c\ Z)\ (Q'\ Z),(A'\ Z);$
$\forall\,Z.\ c\ Z = c$ *undefined*
$]\!] \Longrightarrow$
$\Gamma,\Theta\vdash_{/F}$ P (*specAnno* P' c Q' A') Q,A
⟨*proof*⟩

lemma *SpecAnnoNoAbrupt*:
$[\![P \subseteq \{s.\ \exists\ Z.\ s{\in}P'\ Z\ \wedge$
$(\forall\,t.\ t \in Q'\ Z \longrightarrow\ t \in Q)\};$
$\forall\,Z.\ \Gamma,\Theta\vdash_{/F} (P'\ Z)\ (c\ Z)\ (Q'\ Z),\{\};$
$\forall\,Z.\ c\ Z = c$ *undefined*
$]\!] \Longrightarrow$
$\Gamma,\Theta\vdash_{/F}$ P (*specAnno* P' c Q' $(\lambda s.\ \{\})$) Q,A
⟨*proof*⟩

lemma *Skip*: $P \subseteq Q \Longrightarrow \Gamma,\Theta\vdash_{/F} P$ *Skip* Q,A
⟨*proof*⟩

lemma *Basic*: $P \subseteq \{s.\ (f\ s) \in Q\} \Longrightarrow \Gamma,\Theta\vdash_{/F} P$ (*Basic* f) Q,A
⟨*proof*⟩

lemma *BasicCond*:
$[\![P \subseteq \{s.\ (b\ s \longrightarrow f\ s{\in}Q) \wedge (\neg\ b\ s \longrightarrow g\ s{\in}Q)\}]\!] \Longrightarrow$
$\Gamma,\Theta\vdash_{/F} P$ *Basic* ($\lambda s.$ *if* b s *then* f s *else* g s) Q,A
⟨*proof*⟩

lemma *Spec*: $P \subseteq \{s.\ (\forall\,t.\ (s,t) \in r \longrightarrow t \in Q) \wedge (\exists\,t.\ (s,t) \in r)\}$
$\Longrightarrow \Gamma,\Theta\vdash_{/F} P$ (*Spec* r) Q,A
⟨*proof*⟩

lemma *SpecIf*:
$[\![P \subseteq \{s.\ (b\ s \longrightarrow f\ s \in Q) \wedge (\neg\ b\ s \longrightarrow g\ s \in Q \wedge h\ s \in Q)\}]\!] \Longrightarrow$
$\Gamma,\Theta\vdash_{/F} P$ *Spec* (*if-rel* b f g h) Q,A
⟨*proof*⟩

lemma *Seq* [*trans*, *intro?*]:
$[\![\Gamma,\Theta\vdash_{/F} P\ c_1\ R,A;\ \Gamma,\Theta\vdash_{/F} R\ c_2\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F} P$ (*Seq* c_1 c_2) Q,A
⟨*proof*⟩

lemma *SeqSame*:
$[\![\Gamma,\Theta\vdash_{/F}\ P\ c_1\ Q,A;\ \Gamma,\Theta\vdash_{/F}\ Q\ c_2\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (Seq\ c_1\ c_2)\ Q,A$
⟨*proof*⟩

lemma *SeqSwap*:
$[\![\Gamma,\Theta\vdash_{/F}\ R\ c2\ Q,A;\ \Gamma,\Theta\vdash_{/F}\ P\ c1\ R,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (Seq\ c1\ c2)\ Q,A$
⟨*proof*⟩

lemma *BSeq*:
$[\![\Gamma,\Theta\vdash_{/F}\ P\ c_1\ R,A;\ \Gamma,\Theta\vdash_{/F}\ R\ c_2\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (bseq\ c_1\ c_2)\ Q,A$
⟨*proof*⟩

lemma *BSeqSame*:
$[\![\Gamma,\Theta\vdash_{/F}\ P\ c_1\ Q,A;\ \Gamma,\Theta\vdash_{/F}\ Q\ c_2\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (bseq\ c_1\ c_2)\ Q,A$
⟨*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*: $\Gamma,\Theta\vdash_{/F}\ P_1\ c_1\ Q,A$
assumes *deriv-c2*: $\Gamma,\Theta\vdash_{/F}\ P_2\ c_2\ Q,A$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (Cond\ b\ c_1\ c_2)\ Q,A$
⟨*proof*⟩

lemma *CondSwap*:
$[\![\Gamma,\Theta\vdash_{/F}\ P1\ c1\ Q,A;\ \Gamma,\Theta\vdash_{/F}\ P2\ c2\ Q,A;\ P \subseteq \{s.\ (s\in b \longrightarrow s\in P1) \wedge (s\notin b \longrightarrow s\in P2)\}]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\ P\ (Cond\ b\ c1\ c2)\ Q,A$
⟨*proof*⟩

lemma *Cond'*:
$[\![P \subseteq \{s.\ (b \subseteq P1) \wedge (-\ b \subseteq P2)\};\Gamma,\Theta\vdash_{/F}\ P1\ c1\ Q,A;\ \Gamma,\Theta\vdash_{/F}\ P2\ c2\ Q,A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\ P\ (Cond\ b\ c1\ c2)\ Q,A$
⟨*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*: $\Gamma,\Theta\vdash_{/F}\ P_1\ c_1\ Q,A$
assumes *deriv-c2*: $\Gamma,\Theta\vdash_{/F}\ P_2\ c_2\ Q,A$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (Cond\ b\ c_1\ c_2)\ Q,A$
⟨*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 $\Gamma,\Theta\vdash_{/F}\ P\ (Cond\ b\ c_1\ c_2)\ Q,A$
⟨*proof*⟩

lemma *switchNil*:
$P \subseteq Q \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (switch\ v\ [])\ Q,A$
⟨*proof*⟩

lemma *switchCons*:
$[\![P \subseteq \{s.\ (v\ s \in V \longrightarrow s \in P_1) \wedge (v\ s \notin V \longrightarrow s \in P_2)\};$
$\Gamma,\Theta\vdash_{/F}\ P_1\ c\ Q,A;$
$\Gamma,\Theta\vdash_{/F}\ P_2\ (switch\ v\ vs)\ Q,A]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (switch\ v\ ((V,c)\#vs))\ Q,A$
⟨*proof*⟩

lemma *Guard*:
$[\![P \subseteq g \cap R;\ \Gamma,\Theta\vdash_{/F}\ R\ c\ Q,A]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (Guard\ f\ g\ c)\ Q,A$
⟨*proof*⟩

lemma *GuardSwap*:
$[\![\ \Gamma,\Theta\vdash_{/F}\ R\ c\ Q,A;\ P \subseteq g \cap R]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (Guard\ f\ g\ c)\ Q,A$
⟨*proof*⟩

lemma *Guarantee*:
$[\![P \subseteq \{s.\ s \in g \longrightarrow s \in R\};\ \Gamma,\Theta\vdash_{/F}\ R\ c\ Q,A;\ f \in F]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (Guard\ f\ g\ c)\ Q,A$
⟨*proof*⟩

lemma *GuaranteeSwap*:
$[\![\ \Gamma,\Theta\vdash_{/F}\ R\ c\ Q,A;\ P \subseteq \{s.\ s \in g \longrightarrow s \in R\};\ f \in F]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (Guard\ f\ g\ c)\ Q,A$
⟨*proof*⟩

lemma *GuardStrip*:
$[\![P \subseteq R;\ \Gamma,\Theta\vdash_{/F}\ R\ c\ Q,A;\ f \in F]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (Guard\ f\ g\ c)\ Q,A$
⟨*proof*⟩

lemma *GuardStripSame*:

$[\![\Gamma,\Theta\vdash_{/F}\ P\ c\ Q,A;\ f \in F]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (\mathit{Guard}\ f\ g\ c)\ Q,A$
$\langle\mathit{proof}\rangle$

lemma *GuardStripSwap*:
$[\![\Gamma,\Theta\vdash_{/F}\ R\ c\ Q,A;\ P \subseteq R;\ f \in F]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (\mathit{Guard}\ f\ g\ c)\ Q,A$
$\langle\mathit{proof}\rangle$

lemma *GuaranteeStrip*:
$[\![P \subseteq R;\ \Gamma,\Theta\vdash_{/F}\ R\ c\ Q,A;\ f \in F]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (\mathit{guaranteeStrip}\ f\ g\ c)\ Q,A$
$\langle\mathit{proof}\rangle$

lemma *GuaranteeStripSwap*:
$[\![\Gamma,\Theta\vdash_{/F}\ R\ c\ Q,A;\ P \subseteq R;\ f \in F]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (\mathit{guaranteeStrip}\ f\ g\ c)\ Q,A$
$\langle\mathit{proof}\rangle$

lemma *GuaranteeAsGuard*:
$[\![P \subseteq g \cap R;\ \Gamma,\Theta\vdash_{/F}\ R\ c\ Q,A]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (\mathit{guaranteeStrip}\ f\ g\ c)\ Q,A$
$\langle\mathit{proof}\rangle$

lemma *GuaranteeAsGuardSwap*:
$[\![\ \Gamma,\Theta\vdash_{/F}\ R\ c\ Q,A;\ P \subseteq g \cap R]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (\mathit{guaranteeStrip}\ f\ g\ c)\ Q,A$
$\langle\mathit{proof}\rangle$

lemma *GuardsNil*:
$\Gamma,\Theta\vdash_{/F}\ P\ c\ Q,A \Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\ P\ (\mathit{guards}\ []\ c)\ Q,A$
$\langle\mathit{proof}\rangle$

lemma *GuardsCons*:
$\Gamma,\Theta\vdash_{/F}\ P\ \mathit{Guard}\ f\ g\ (\mathit{guards}\ \mathit{gs}\ c)\ Q,A \Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\ P\ (\mathit{guards}\ ((f,g)\#\mathit{gs})\ c)\ Q,A$
$\langle\mathit{proof}\rangle$

lemma *GuardsConsGuaranteeStrip*:
$\Gamma,\Theta\vdash_{/F}\ P\ \mathit{guaranteeStrip}\ f\ g\ (\mathit{guards}\ \mathit{gs}\ c)\ Q,A \Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\ P\ (\mathit{guards}\ (\mathit{guaranteeStripPair}\ f\ g\#\mathit{gs})\ c)\ Q,A$
$\langle\mathit{proof}\rangle$

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 $\Gamma,\Theta\vdash_{/F} P\ (\mathit{whileAnno}\ b\ I\ V\ c)\ Q,A$
⟨*proof*⟩

J will be instantiated by tactic with $gs' \cap I$ for those guards that are not stripped.

lemma *WhileAnnoG*:
$\Gamma,\Theta\vdash_{/F} P\ (\mathit{guards}\ gs$
$(\mathit{whileAnno}\ \ b\ J\ V\ (\mathit{Seq}\ c\ (\mathit{guards}\ gs\ \mathit{Skip}))))\ Q,A$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F} P\ (\mathit{whileAnnoG}\ gs\ b\ I\ V\ c)\ Q,A$
⟨*proof*⟩

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 $\Gamma,\Theta\vdash_{/F} P\ (\mathit{whileAnno}\ b\ I\ V\ (\mathit{Seq}\ c\ \mathit{Skip}))\ Q,A$
⟨*proof*⟩

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*: $\forall Z.\ c\ Z = c\ \mathit{undefined}$
shows $\Gamma,\Theta\vdash_{/F} P\ (\mathit{whileAnnoFix}\ b\ I\ V\ c)\ Q,A$
⟨*proof*⟩

lemma *WhileAnnoFix'*:
assumes *consequence*: $P \subseteq \{s.\ (\exists\ Z.\ s{\in}I\ Z \wedge$
$(\forall t.\ t \in I\ Z \cap -b \longrightarrow t \in Q))\ \}$
assumes *bdy*: $\forall Z.\ \Gamma,\Theta\vdash_{/F} (I\ Z \cap b)\ (c\ Z)\ (I\ Z),A$
assumes *bdy-constant*: $\forall Z.\ c\ Z = c\ \mathit{undefined}$
shows $\Gamma,\Theta\vdash_{/F} P\ (\mathit{whileAnnoFix}\ b\ I\ V\ c)\ Q,A$
⟨*proof*⟩

lemma *WhileAnnoGFix*:
assumes *whileAnnoFix*:
$\Gamma,\Theta\vdash_{/F} P\ (\mathit{guards}\ gs$
$(\mathit{whileAnnoFix}\ \ b\ J\ V\ (\lambda Z.\ (\mathit{Seq}\ (c\ Z)\ (\mathit{guards}\ gs\ \mathit{Skip})))))\ Q,A$
shows $\Gamma,\Theta\vdash_{/F} P\ (\mathit{whileAnnoGFix}\ gs\ b\ I\ V\ c)\ Q,A$
⟨*proof*⟩

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 $\Gamma,\Theta\vdash_{/F} P\ (bind\ e\ c)\ Q,A$
⟨*proof*⟩

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 $\Gamma,\Theta\vdash_{/F} P\ (block\text{-}exn\ init\ bdy\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

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 $\Gamma,\Theta\vdash_{/F} P\ (block\ init\ bdy\ return\ c)\ Q,A$
⟨*proof*⟩

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 $\Gamma,\Theta\vdash_{/F} P\ (block\ init\ bdy\ return\ c)\ Q,A$
⟨*proof*⟩

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 $\Gamma,\Theta\vdash_{/F} P\ (block\text{-}exn\ init\ bdy\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

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 $\Gamma,\Theta\vdash_{/F} P\ (block\ init\ bdy\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *Throw*: $P \subseteq A \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ Throw\ Q,A$
⟨*proof*⟩

lemmas *Catch* = *hoarep.Catch*
lemma *CatchSwap*: $[\![\Gamma,\Theta\vdash_{/F}\ R\ c_2\ Q,A;\ \Gamma,\Theta\vdash_{/F}\ P\ c_1\ Q,R]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ Catch\ c_1\ c_2\ Q,A$
⟨*proof*⟩

lemma *CatchSame*: $[\![\Gamma,\Theta\vdash_{/F}\ P\ c_1\ Q,A;\ \Gamma,\Theta\vdash_{/F}\ A\ c_2\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ Catch\ c_1\ c_2\ Q,A$
⟨*proof*⟩

lemma *raise*: $P \subseteq \{s.\ f\ s \in A\} \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ raise\ f\ Q,A$
⟨*proof*⟩

lemma *condCatch*: $[\![\Gamma,\Theta\vdash_{/F}\ P\ c_1\ Q,((b \cap R) \cup (-b \cap A));\Gamma,\Theta\vdash_{/F}\ R\ c_2\ Q,A]\!]$
$\Longrightarrow\ \Gamma,\Theta\vdash_{/F}P\ condCatch\ c_1\ b\ c_2\ Q,A$
⟨*proof*⟩

lemma *condCatchSwap*: $[\![\Gamma,\Theta\vdash_{/F}\ R\ c_2\ Q,A;\Gamma,\Theta\vdash_{/F}\ P\ c_1\ Q,((b \cap R) \cup (-b \cap A))]\!]$
$\Longrightarrow\ \Gamma,\Theta\vdash_{/F}P\ condCatch\ c_1\ b\ c_2\ Q,A$
⟨*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 $\Gamma,\Theta\vdash_{/F}P\ condCatch\ c_1\ b\ c_2\ Q,A$
⟨*proof*⟩

lemma *ProcSpec*:
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 *p*: $\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ Call\ p\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (call\ init\ p\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *Proc-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 *p*: $\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ Call\ p\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$

⟨*proof*⟩

lemma *ProcSpec'*:
  assumes *adapt*: $P \subseteq \{s.\ \exists Z.\ init\ s \in P'\ Z \wedge$
  $(\forall t \in Q'\ Z.\ return\ s\ t \in R\ s\ t) \wedge$
  $(\forall t \in A'\ Z.\ return\ s\ t \in A)\}$
  assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
  assumes *p*: $\forall Z.\ \Gamma,\Theta\vdash_{/F} (P'\ Z)\ Call\ p\ (Q'\ Z),(A'\ Z)$
  shows $\Gamma,\Theta\vdash_{/F} P\ (call\ init\ p\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *Proc-exnSpecNoAbrupt*:
  assumes *adapt*: $P \subseteq \{s.\ \exists Z.\ init\ s \in P'\ Z \wedge$
  $(\forall t.\ t \in Q'\ Z \longrightarrow return\ s\ t \in R\ s\ t)\}$
  assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
  assumes *p*: $\forall Z.\ \Gamma,\Theta\vdash_{/F} (P'\ Z)\ Call\ p\ (Q'\ Z),\{\}$
  shows $\Gamma,\Theta\vdash_{/F} P\ (call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *ProcSpecNoAbrupt*:
  assumes *adapt*: $P \subseteq \{s.\ \exists Z.\ init\ s \in P'\ Z \wedge$
  $(\forall t.\ t \in Q'\ Z \longrightarrow return\ s\ t \in R\ s\ t)\}$
  assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
  assumes *p*: $\forall Z.\ \Gamma,\Theta\vdash_{/F} (P'\ Z)\ Call\ p\ (Q'\ Z),\{\}$
  shows $\Gamma,\Theta\vdash_{/F} P\ (call\ init\ p\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *FCall*:
$\Gamma,\Theta\vdash_{/F} P\ (call\ init\ p\ return\ (\lambda s\ t.\ c\ (result\ t)))\ Q,A$
$\Longrightarrow \Gamma,\Theta\vdash_{/F} P\ (fcall\ init\ p\ return\ result\ c)\ Q,A$
  ⟨*proof*⟩

lemma *ProcRec*:
  assumes *deriv-bodies*:
  $\forall p{\in}Procs.$
  $\forall Z.\ \Gamma,\Theta\cup(\bigcup p{\in}Procs.\ \bigcup Z.\ \{(P\ p\ Z,p,Q\ p\ Z,A\ p\ Z)\})$
  $\vdash_{/F} (P\ p\ Z)\ (the\ (\Gamma\ p))\ (Q\ p\ Z),(A\ p\ Z)$
  assumes *Procs-defined*: $Procs \subseteq dom\ \Gamma$
  shows $\forall p{\in}Procs.\ \forall Z.\ \Gamma,\Theta\vdash_{/F}(P\ p\ Z)\ Call\ p\ (Q\ p\ Z),(A\ p\ Z)$
  ⟨*proof*⟩

lemma *ProcRec'*:
  assumes *ctxt*: $\Theta' = \Theta\cup(\bigcup p{\in}Procs.\ \bigcup Z.\ \{(P\ p\ Z,p,Q\ p\ Z,A\ p\ Z)\})$
  assumes *deriv-bodies*:
  $\forall p{\in}Procs.\ \forall Z.\ \Gamma,\Theta'\vdash_{/F} (P\ p\ Z)\ (the\ (\Gamma\ p))\ (Q\ p\ Z),(A\ p\ Z)$
  assumes *Procs-defined*: $Procs \subseteq dom\ \Gamma$

shows $\forall p{\in}Procs.\ \forall Z.\ \Gamma,\Theta\vdash_{/F}(P\ p\ Z)\ Call\ p\ (Q\ p\ Z),(A\ p\ Z)$
⟨*proof*⟩

lemma *ProcRecList*:
assumes *deriv-bodies*:
$\forall p{\in}set\ Procs.$
$\forall Z.\ \Gamma,\Theta\cup(\bigcup p{\in}set\ Procs.\ \bigcup Z.\ \{(P\ p\ Z,p,Q\ p\ Z,A\ p\ Z)\})$
$\vdash_{/F}(P\ p\ Z)\ (the\ (\Gamma\ p))\ (Q\ p\ Z),(A\ p\ Z)$
assumes *dist*: *distinct Procs*
assumes *Procs-defined*: $set\ Procs \subseteq dom\ \Gamma$
shows $\forall p{\in}set\ Procs.\ \forall Z.\ \Gamma,\Theta\vdash_{/F}(P\ p\ Z)\ Call\ p\ (Q\ p\ Z),(A\ p\ Z)$
⟨*proof*⟩

lemma *ProcRecSpecs*:
$[\![\forall (P,p,Q,A) \in Specs.\ \Gamma,\Theta\cup Specs\vdash_{/F} P\ (the\ (\Gamma\ p))\ Q,A;$
$\forall (P,p,Q,A) \in Specs.\ p \in dom\ \Gamma]\!]$
$\Longrightarrow \forall (P,p,Q,A) \in Specs.\ \Gamma,\Theta\vdash_{/F} P\ (Call\ p)\ Q,A$
⟨*proof*⟩

lemma *ProcRec1*:
assumes *deriv-body*:
$\forall Z.\ \Gamma,\Theta\cup(\bigcup Z.\ \{(P\ Z,p,Q\ Z,A\ Z)\})\vdash_{/F}(P\ Z)\ (the\ (\Gamma\ p))\ (Q\ Z),(A\ Z)$
assumes *p-defined*: $p \in dom\ \Gamma$
shows $\forall Z.\ \Gamma,\Theta\vdash_{/F}(P\ Z)\ Call\ p\ (Q\ Z),(A\ Z)$
⟨*proof*⟩

lemma *ProcNoRec1*:
assumes *deriv-body*:
$\forall Z.\ \Gamma,\Theta\vdash_{/F}(P\ Z)\ (the\ (\Gamma\ p))\ (Q\ Z),(A\ Z)$
assumes *p-def*: $p \in dom\ \Gamma$
shows $\forall Z.\ \Gamma,\Theta\vdash_{/F}(P\ Z)\ Call\ p\ (Q\ Z),(A\ Z)$
⟨*proof*⟩

lemma *ProcBody*:
assumes *WP*: $P \subseteq P'$
assumes *deriv-body*: $\Gamma,\Theta\vdash_{/F} P'\ body\ Q,A$
assumes *body*: $\Gamma\ p = Some\ body$
shows $\Gamma,\Theta\vdash_{/F} P\ Call\ p\ Q,A$
⟨*proof*⟩

lemma *CallBody*:
assumes *adapt*: $P \subseteq \{s.\ init\ s \in P'\ s\}$
assumes *bdy*: $\forall s.\ \Gamma,\Theta\vdash_{/F}(P'\ s)\ body\ \{t.\ return\ s\ t \in R\ s\ t\},\{t.\ return\ s\ t \in A\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{/F}(R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *body*: $\Gamma\ p = Some\ body$

shows $\Gamma,\Theta\vdash_{/F}$ P (*call init p return c*) Q,A
⟨*proof*⟩

lemma *Call-exnBody*:
assumes *adapt*: $P \subseteq \{s.\ init\ s \in P'\ s\}$
assumes *bdy*: $\forall s.\ \Gamma,\Theta\vdash_{/F} (P'\ s)\ body\ \{t.\ return\ s\ t \in R\ s\ t\},\{t.\ result\text{-}exn\ (return\ s\ t)\ t \in A\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *body*: $\Gamma\ p = Some\ body$
shows $\Gamma,\Theta\vdash_{/F}$ P (*call-exn init p return result-exn c*) Q,A
⟨*proof*⟩

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 (*call-exn init p return' result-exn c*) Q,A
assumes *result-conform*:
$\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow (return'\ s\ t) = (return\ s\ t)$
assumes *modifies-spec*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/UNIV} \{\sigma\}\ Call\ p\ (Modif\ \sigma),\{\}$
shows $\Gamma,\Theta\vdash_{/F}$ P (*call-exn init p return result-exn c*) Q,A
⟨*proof*⟩

lemma *ProcModifyReturnNoAbr*:
assumes *spec*: $\Gamma,\Theta\vdash_{/F}$ P (*call init p return' c*) Q,A
assumes *result-conform*:
$\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow (return'\ s\ t) = (return\ s\ t)$
assumes *modifies-spec*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/UNIV} \{\sigma\}\ Call\ p\ (Modif\ \sigma),\{\}$
shows $\Gamma,\Theta\vdash_{/F}$ P (*call init p return c*) Q,A
⟨*proof*⟩

lemma *Proc-exnModifyReturnNoAbrSameFaults*:
assumes *spec*: $\Gamma,\Theta\vdash_{/F}$ P (*call-exn init p return' result-exn c*) Q,A
assumes *result-conform*:
$\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow (return'\ s\ t) = (return\ s\ t)$
assumes *modifies-spec*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ Call\ p\ (Modif\ \sigma),\{\}$
shows $\Gamma,\Theta\vdash_{/F}$ P (*call-exn init p return result-exn c*) Q,A
⟨*proof*⟩

lemma *ProcModifyReturnNoAbrSameFaults*:
assumes *spec*: $\Gamma,\Theta\vdash_{/F}$ P (*call init p return' c*) Q,A
assumes *result-conform*:

$\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow (return'\ s\ t) = (return\ s\ t)$
assumes *modifies-spec*:
$\forall \sigma.\ \Gamma,\Theta \vdash_{/F} \{\sigma\}\ Call\ p\ (Modif\ \sigma),\{\}$
shows $\Gamma,\Theta \vdash_{/F} P\ (call\ init\ p\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *DynProc-exn*:
assumes *adapt*: $P \subseteq \{s.\ \exists Z.\ init\ s \in P'\ s\ Z \wedge$
$(\forall t.\ t \in Q'\ s\ Z \longrightarrow\ return\ s\ t \in R\ s\ t) \wedge$
$(\forall t.\ t \in A'\ s\ Z \longrightarrow result\text{-}exn\ (return\ s\ t)\ t \in A)\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta \vdash_{/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *p*: $\forall s \in P.\ \forall Z.\ \Gamma,\Theta \vdash_{/F} (P'\ s\ Z)\ Call\ (p\ s)\ (Q'\ s\ Z),(A'\ s\ Z)$
shows $\Gamma,\Theta \vdash_{/F} P\ dynCall\text{-}exn\ f\ UNIV\ init\ p\ return\ result\text{-}exn\ c\ Q,A$
⟨*proof*⟩

lemma *DynProc-exn-guards-cons*:
assumes *p*: $\Gamma,\Theta \vdash_{/F} P\ dynCall\text{-}exn\ f\ UNIV\ init\ p\ return\ result\text{-}exn\ c\ Q,A$
shows $\Gamma,\Theta \vdash_{/F} (g \cap P)\ dynCall\text{-}exn\ f\ g\ init\ p\ return\ result\text{-}exn\ c\ Q,A$
⟨*proof*⟩

lemma *DynProc*:
assumes *adapt*: $P \subseteq \{s.\ \exists Z.\ init\ s \in P'\ s\ Z \wedge$
$(\forall t.\ t \in Q'\ s\ Z \longrightarrow\ return\ s\ t \in R\ s\ t) \wedge$
$(\forall t.\ t \in A'\ s\ Z \longrightarrow return\ s\ t \in A)\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta \vdash_{/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *p*: $\forall s \in P.\ \forall Z.\ \Gamma,\Theta \vdash_{/F} (P'\ s\ Z)\ Call\ (p\ s)\ (Q'\ s\ Z),(A'\ s\ Z)$
shows $\Gamma,\Theta \vdash_{/F} P\ dynCall\ init\ p\ return\ c\ Q,A$
⟨*proof*⟩

lemma *DynProc-exn'*:
assumes *adapt*: $P \subseteq \{s.\ \exists Z.\ init\ s \in P'\ s\ Z \wedge$
$(\forall t \in Q'\ s\ Z.\ return\ s\ t \in R\ s\ t) \wedge$
$(\forall t \in A'\ s\ Z.\ result\text{-}exn\ (return\ s\ t)\ t \in A)\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta \vdash_{/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *p*: $\forall s \in P.\ \forall Z.\ \Gamma,\Theta \vdash_{/F} (P'\ s\ Z)\ Call\ (p\ s)\ (Q'\ s\ Z),(A'\ s\ Z)$
shows $\Gamma,\Theta \vdash_{/F} P\ dynCall\text{-}exn\ f\ UNIV\ init\ p\ return\ result\text{-}exn\ c\ Q,A$
⟨*proof*⟩

lemma *DynProc'*:
assumes *adapt*: $P \subseteq \{s.\ \exists Z.\ init\ s \in P'\ s\ Z \wedge$
$(\forall t \in Q'\ s\ Z.\ return\ s\ t \in R\ s\ t) \wedge$
$(\forall t \in A'\ s\ Z.\ return\ s\ t \in A)\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta \vdash_{/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *p*: $\forall s \in P.\ \forall Z.\ \Gamma,\Theta \vdash_{/F} (P'\ s\ Z)\ Call\ (p\ s)\ (Q'\ s\ Z),(A'\ s\ Z)$
shows $\Gamma,\Theta \vdash_{/F} P\ dynCall\ init\ p\ return\ c\ Q,A$
⟨*proof*⟩

lemma *DynProc-exnStaticSpec*:
assumes *adapt*: $P \subseteq \{s.\ s \in S \wedge (\exists Z.\ init\ s \in P'\ Z\ \wedge$
$(\forall \tau.\ \tau \in Q'\ Z \longrightarrow return\ s\ \tau \in R\ s\ \tau) \wedge$
$(\forall \tau.\ \tau \in A'\ Z \longrightarrow result\text{-}exn\ (return\ s\ \tau)\ \tau \in A))\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{/F}\ (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *spec*: $\forall s{\in}S.\ \forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ Call\ (p\ s)\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (dynCall\text{-}exn\ f\ UNIV\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *DynProcStaticSpec*:
assumes *adapt*: $P \subseteq \{s.\ s \in S \wedge (\exists Z.\ init\ s \in P'\ Z\ \wedge$
$(\forall \tau.\ \tau \in Q'\ Z \longrightarrow return\ s\ \tau \in R\ s\ \tau) \wedge$
$(\forall \tau.\ \tau \in A'\ Z \longrightarrow return\ s\ \tau \in A))\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{/F}\ (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *spec*: $\forall s{\in}S.\ \forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ Call\ (p\ s)\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (dynCall\ init\ p\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *DynProc-exnProcPar*:
assumes *adapt*: $P \subseteq \{s.\ p\ s = q \wedge (\exists Z.\ init\ s \in P'\ Z\ \wedge$
$(\forall \tau.\ \tau \in Q'\ Z \longrightarrow return\ s\ \tau \in R\ s\ \tau) \wedge$
$(\forall \tau.\ \tau \in A'\ Z \longrightarrow result\text{-}exn\ (return\ s\ \tau)\ \tau \in A))\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{/F}\ (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *spec*: $\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ Call\ q\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (dynCall\text{-}exn\ f\ UNIV\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *DynProcProcPar*:
assumes *adapt*: $P \subseteq \{s.\ p\ s = q \wedge (\exists Z.\ init\ s \in P'\ Z\ \wedge$
$(\forall \tau.\ \tau \in Q'\ Z \longrightarrow return\ s\ \tau \in R\ s\ \tau) \wedge$
$(\forall \tau.\ \tau \in A'\ Z \longrightarrow return\ s\ \tau \in A))\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{/F}\ (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *spec*: $\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ Call\ q\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (dynCall\ init\ p\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *DynProc-exnProcParNoAbrupt*:
assumes *adapt*: $P \subseteq \{s.\ p\ s = q \wedge (\exists Z.\ init\ s \in P'\ Z\ \wedge$
$(\forall \tau.\ \tau \in Q'\ Z \longrightarrow return\ s\ \tau \in R\ s\ \tau))\}$
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 $\Gamma,\Theta\vdash_{/F}\ P\ (dynCall\text{-}exn\ f\ UNIV\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *DynProcProcParNoAbrupt*:
assumes *adapt*: $P \subseteq \{s.\ p\ s = q \wedge (\exists Z.\ init\ s \in P'\ Z\ \wedge$
$(\forall \tau.\ \tau \in Q'\ Z \longrightarrow return\ s\ \tau \in R\ s\ \tau))\}$

assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{/F}\ (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *spec*: $\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ \mathit{Call}\ q\ (Q'\ Z),\{\}$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (\mathit{dynCall\ init\ p\ return\ c})\ Q,A$
⟨*proof*⟩

lemma *DynProc-exnModifyReturnNoAbr*:
assumes *to-prove*: $\Gamma,\Theta\vdash_{/F}\ P\ (\mathit{dynCall\text{-}exn\ f\ g\ init\ p\ return'\ result\text{-}exn\ c})\ Q,A$
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (\mathit{Modif}\ (\mathit{init}\ s))$
$\longrightarrow \mathit{return'}\ s\ t = \mathit{return}\ s\ t$
assumes *modif-clause*:
$\forall s \in P.\ \forall \sigma.\ \Gamma,\Theta\vdash_{/UNIV}\ \{\sigma\}\ \mathit{Call}\ (p\ s)\ \ (\mathit{Modif}\ \sigma),\{\}$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (\mathit{dynCall\text{-}exn\ f\ g\ init\ p\ return\ result\text{-}exn\ c})\ Q,A$
⟨*proof*⟩

lemma *DynProcModifyReturnNoAbr*:
assumes *to-prove*: $\Gamma,\Theta\vdash_{/F}\ P\ (\mathit{dynCall\ init\ p\ return'\ c})\ Q,A$
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (\mathit{Modif}\ (\mathit{init}\ s))$
$\longrightarrow \mathit{return'}\ s\ t = \mathit{return}\ s\ t$
assumes *modif-clause*:
$\forall s \in P.\ \forall \sigma.\ \Gamma,\Theta\vdash_{/UNIV}\ \{\sigma\}\ \mathit{Call}\ (p\ s)\ \ (\mathit{Modif}\ \sigma),\{\}$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (\mathit{dynCall\ init\ p\ return\ c})\ Q,A$
⟨*proof*⟩

lemma *ProcDyn-exnModifyReturnNoAbrSameFaults*:
assumes *to-prove*: $\Gamma,\Theta\vdash_{/F}\ P\ (\mathit{dynCall\text{-}exn\ f\ g\ init\ p\ return'\ result\text{-}exn\ c})\ Q,A$
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (\mathit{Modif}\ (\mathit{init}\ s))$
$\longrightarrow \mathit{return'}\ s\ t = \mathit{return}\ s\ t$
assumes *modif-clause*:
$\forall s \in P.\ \forall \sigma.\ \Gamma,\Theta\vdash_{/F}\ \{\sigma\}\ (\mathit{Call}\ (p\ s))\ (\mathit{Modif}\ \sigma),\{\}$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (\mathit{dynCall\text{-}exn\ f\ g\ init\ p\ return\ result\text{-}exn\ c})\ Q,A$
⟨*proof*⟩

lemma *ProcDynModifyReturnNoAbrSameFaults*:
assumes *to-prove*: $\Gamma,\Theta\vdash_{/F}\ P\ (\mathit{dynCall\ init\ p\ return'\ c})\ Q,A$
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (\mathit{Modif}\ (\mathit{init}\ s))$
$\longrightarrow \mathit{return'}\ s\ t = \mathit{return}\ s\ t$
assumes *modif-clause*:
$\forall s \in P.\ \forall \sigma.\ \Gamma,\Theta\vdash_{/F}\ \{\sigma\}\ (\mathit{Call}\ (p\ s))\ (\mathit{Modif}\ \sigma),\{\}$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (\mathit{dynCall\ init\ p\ return\ c})\ Q,A$
⟨*proof*⟩

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*: $\Gamma,\Theta\vdash_{/F}\ P'\ (\mathit{dynCall\text{-}exn\ f\ g\ init\ p\ return'\ result\text{-}exn\ c})\ Q,A$

assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (Modif\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *ret-abr-modif*: $\forall s\ t.\ t \in (ModifAbr\ (init\ s))$
$\longrightarrow result\text{-}exn\ (return'\ s\ t)\ t = result\text{-}exn\ (return\ s\ t)\ t$
assumes *modif-clause*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/UNIV}\ \{\sigma\}\ (Call\ q)\ (Modif\ \sigma),(ModifAbr\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (dynCall\text{-}exn\ f\ g\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*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*: $\Gamma,\Theta\vdash_{/F}\ P'\ (dynCall\ init\ p\ return'\ c)\ Q,A$
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (Modif\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *ret-abr-modif*: $\forall s\ t.\ t \in (ModifAbr\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *modif-clause*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/UNIV}\ \{\sigma\}\ (Call\ q)\ (Modif\ \sigma),(ModifAbr\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (dynCall\ init\ p\ return\ c)\ Q,A$
⟨*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*: $\Gamma,\Theta\vdash_{/F}\ P'\ (dynCall\text{-}exn\ f\ g\ init\ p\ return'\ result\text{-}exn\ c)\ Q,A$
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (Modif\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *ret-abr-modif*: $\forall s\ t.\ t \in (ModifAbr\ (init\ s))$
$\longrightarrow result\text{-}exn\ (return'\ s\ t)\ t = result\text{-}exn\ (return\ s\ t)\ t$
assumes *modif-clause*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/F}\ \{\sigma\}\ Call\ q\ (Modif\ \sigma),(ModifAbr\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (dynCall\text{-}exn\ f\ g\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*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*: $\Gamma,\Theta\vdash_{/F}\ P'\ (dynCall\ init\ p\ return'\ c)\ Q,A$
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (Modif\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *ret-abr-modif*: $\forall s\ t.\ t \in (ModifAbr\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *modif-clause*:

$\forall \sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ Call\ q\ (Modif\ \sigma),(ModifAbr\ \sigma)$
shows $\Gamma,\Theta\vdash_{/F} P\ (dynCall\ init\ p\ return\ c)\ Q,A$
$\langle proof\rangle$

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*: $\Gamma,\Theta\vdash_{/F} P'\ (dynCall\text{-}exn\ f\ g\ init\ p\ return'\ result\text{-}exn\ c)\ Q,A$
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (Modif\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *modif-clause*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/UNIV} \{\sigma\}\ (Call\ q)\ (Modif\ \sigma),\{\}$
shows $\Gamma,\Theta\vdash_{/F} P\ (dynCall\text{-}exn\ f\ g\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
$\langle proof\rangle$

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*: $\Gamma,\Theta\vdash_{/F} P'\ (dynCall\ init\ p\ return'\ c)\ Q,A$
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (Modif\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *modif-clause*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/UNIV} \{\sigma\}\ (Call\ q)\ (Modif\ \sigma),\{\}$
shows $\Gamma,\Theta\vdash_{/F} P\ (dynCall\ init\ p\ return\ c)\ Q,A$
$\langle proof\rangle$

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*: $\Gamma,\Theta\vdash_{/F} P'\ (dynCall\text{-}exn\ f\ g\ init\ p\ return'\ result\text{-}exn\ c)\ Q,A$
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (Modif\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *modif-clause*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ (Call\ q)\ (Modif\ \sigma),\{\}$
shows $\Gamma,\Theta\vdash_{/F} P\ (dynCall\text{-}exn\ f\ g\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
$\langle proof\rangle$

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*: $\Gamma,\Theta\vdash_{/F} P'\ (dynCall\ init\ p\ return'\ c)\ Q,A$
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (Modif\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$

assumes *modif-clause*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/F}\ \{\sigma\}\ (Call\ q)\ (Modif\ \sigma),\{\}$
shows $\Gamma,\Theta\vdash_{/F}\ P\ (dynCall\ init\ p\ return\ c)\ Q,A$
$\langle proof \rangle$

lemma *MergeGuards-iff*: $\Gamma,\Theta\vdash_{/F}\ P\ merge\text{-}guards\ c\ Q,A = \Gamma,\Theta\vdash_{/F}\ P\ c\ Q,A$
$\langle proof \rangle$

lemma *CombineStrip'*:
assumes *deriv*: $\Gamma,\Theta\vdash_{/F}\ P\ c'\ Q,A$
assumes *deriv-strip-triv*: $\Gamma,\{\}\vdash_{/\{\}}\ P\ c''\ UNIV,UNIV$
assumes *c''*: $c''= mark\text{-}guards\ False\ (strip\text{-}guards\ (-F)\ c')$
assumes *c*: $merge\text{-}guards\ c = merge\text{-}guards\ (mark\text{-}guards\ False\ c')$
shows $\Gamma,\Theta\vdash_{/\{\}}\ P\ c\ Q,A$
$\langle proof \rangle$

lemma *CombineStrip''*:
assumes *deriv*: $\Gamma,\Theta\vdash_{/\{True\}}\ P\ c'\ Q,A$
assumes *deriv-strip-triv*: $\Gamma,\{\}\vdash_{/\{\}}\ P\ c''\ UNIV,UNIV$
assumes *c''*: $c''= mark\text{-}guards\ False\ (strip\text{-}guards\ (\{False\})\ c')$
assumes *c*: $merge\text{-}guards\ c = merge\text{-}guards\ (mark\text{-}guards\ False\ c')$
shows $\Gamma,\Theta\vdash_{/\{\}}\ P\ c\ Q,A$
$\langle proof \rangle$

lemma *AsmUN*:
$(\bigcup Z.\ \{(P\ Z,\ p,\ Q\ Z,A\ Z)\}) \subseteq \Theta$
$\Longrightarrow$
$\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P\ Z)\ (Call\ p)\ (Q\ Z),(A\ Z)$
$\langle proof \rangle$

lemma *augment-context'*:
$[\![\Theta \subseteq \Theta';\ \forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P\ Z)\ \ p\ (Q\ Z),(A\ Z)]\!]$
$\Longrightarrow \forall Z.\ \Gamma,\Theta'\vdash_{/F}\ (P\ Z)\ p\ (Q\ Z),(A\ Z)$
$\langle proof \rangle$

lemma *hoarep-strip*:
$[\![\forall Z.\ \Gamma,\{\}\vdash_{/F}\ (P\ Z)\ p\ (Q\ Z),(A\ Z);\ F' \subseteq -F]\!] \Longrightarrow$
$\forall Z.\ strip\ F'\ \Gamma,\{\}\vdash_{/F}\ (P\ Z)\ p\ (Q\ Z),(A\ Z)$
$\langle proof \rangle$

lemma *augment-emptyFaults*:
$[\![\forall Z.\ \Gamma,\{\}\vdash_{/\{\}}\ (P\ Z)\ p\ (Q\ Z),(A\ Z)]\!] \Longrightarrow$
$\forall Z.\ \Gamma,\{\}\vdash_{/F}\ (P\ Z)\ p\ (Q\ Z),(A\ Z)$
$\langle proof \rangle$

lemma *augment-FaultsUNIV*:
$[\![\forall Z.\ \Gamma,\{\}\vdash_{/F} (P\ Z)\ p\ (Q\ Z),(A\ Z)]\!] \Longrightarrow$
$\forall Z.\ \Gamma,\{\}\vdash_{/UNIV} (P\ Z)\ p\ (Q\ Z),(A\ Z)$
⟨*proof*⟩

lemma *PostConjI* [*trans*]:
$[\![\Gamma,\Theta\vdash_{/F}\ P\ c\ Q,A;\ \Gamma,\Theta\vdash_{/F}\ P\ c\ R,B]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ c\ (Q \cap R),(A \cap B)$
⟨*proof*⟩

lemma *PostConjI'* :
$[\![\Gamma,\Theta\vdash_{/F}\ P\ c\ Q,A;\ \Gamma,\Theta\vdash_{/F}\ P\ c\ Q,A \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ c\ R,B]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ c\ (Q \cap R),(A \cap B)$
⟨*proof*⟩

lemma *PostConjE* [*consumes 1*]:
assumes *conj*: $\Gamma,\Theta\vdash_{/F}\ P\ c\ (Q \cap R),(A \cap B)$
assumes *E*: $[\![\Gamma,\Theta\vdash_{/F}\ P\ c\ Q,A;\ \Gamma,\Theta\vdash_{/F}\ P\ c\ R,B]\!] \Longrightarrow S$
shows *S*
⟨*proof*⟩

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*]:
$[\![\Gamma,\Theta\vdash_{/F}P\ anno\ Q,A;\ c = anno]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F}P\ c\ Q,A$
⟨*proof*⟩

lemma *annotate-normI*:
assumes *deriv-anno*: $\Gamma,\Theta\vdash_{/F}P\ anno\ Q,A$
assumes *norm-eq*: *normalize c = normalize anno*
shows $\Gamma,\Theta\vdash_{/F}P\ c\ Q,A$
⟨*proof*⟩

lemma *annotateWhile*:
$[\![\Gamma,\Theta\vdash_{/F}\ P\ (whileAnnoG\ gs\ b\ I\ V\ c)\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (while\ gs\ b\ c)\ Q,A$
⟨*proof*⟩

lemma *reannotateWhile*:
$[\![\Gamma,\Theta\vdash_{/F}\ P\ (whileAnnoG\ gs\ b\ I\ V\ c)\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (whileAnnoG\ gs\ b\ J\ V\ c)\ Q,A$

⟨*proof*⟩

lemma *reannotateWhileNoGuard*:
$[\![\Gamma,\Theta\vdash_{/F}\ P\ (whileAnno\ b\ I\ V\ c)\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (whileAnno\ b\ J\ V\ c)\ Q,A$
⟨*proof*⟩

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

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

lemma [*trans*]:
$\Gamma,\Theta\vdash_{/F}\ \{s.\ P\ s\}\ c\ Q,A \Longrightarrow (\bigwedge s.\ P'\ s \longrightarrow P\ s) \Longrightarrow \Gamma,\Theta\vdash_{/F}\ \{s.\ P'\ s\}\ c\ Q,A$
⟨*proof*⟩

lemma [*trans*]:
$(\bigwedge s.\ P'\ s \longrightarrow P\ s) \Longrightarrow \Gamma,\Theta\vdash_{/F}\ \{s.\ P\ s\}\ c\ Q,A \Longrightarrow \Gamma,\Theta\vdash_{/F}\ \{s.\ P'\ s\}\ c\ Q,A$
⟨*proof*⟩

lemma [*trans*]:
$\Gamma,\Theta\vdash_{/F}P\ c\ \{s.\ Q\ s\},A \Longrightarrow (\bigwedge s.\ Q\ s \longrightarrow Q'\ s) \Longrightarrow \Gamma,\Theta\vdash_{/F}P\ c\ \{s.\ Q'\ s\},A$
⟨*proof*⟩

lemma [*trans*]:
$(\bigwedge s.\ Q\ s \longrightarrow Q'\ s) \Longrightarrow \Gamma,\Theta\vdash_{/F}P\ c\ \{s.\ Q\ s\},A \Longrightarrow \Gamma,\Theta\vdash_{/F}P\ c\ \{s.\ Q'\ s\},A$
⟨*proof*⟩

lemma [*intro?*]: $\Gamma,\Theta\vdash_{/F}\ P\ Skip\ P,A$
⟨*proof*⟩

lemma *CondInt* [*trans,intro?*]:
$[\![\Gamma,\Theta\vdash_{/F}\ (P \cap b)\ c1\ Q,A;\ \Gamma,\Theta\vdash_{/F}\ (P \cap -\ b)\ c2\ Q,A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\ P\ (Cond\ b\ c1\ c2)\ Q,A$
⟨*proof*⟩

lemma *CondConj* [*trans, intro?*]:
$[\![\Gamma,\Theta\vdash_{/F}\ \{s.\ P\ s \wedge b\ s\}\ c1\ Q,A;\ \Gamma,\Theta\vdash_{/F}\ \{s.\ P\ s \wedge \neg\ b\ s\}\ c2\ Q,A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\ \{s.\ P\ s\}\ (Cond\ \{s.\ b\ s\}\ c1\ c2)\ Q,A$
⟨*proof*⟩

lemma *WhileInvInt* [*intro?*]:
$\Gamma,\Theta\vdash_{/F}\ (P \cap b)\ c\ P,A \Longrightarrow \Gamma,\Theta\vdash_{/F}\ P\ (whileAnno\ b\ P\ V\ c)\ (P \cap -b),A$
⟨*proof*⟩

lemma *WhileInt* [*intro?*]:
$\Gamma,\Theta\vdash_{/F}$ $(P \cap b)$ c P,A
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}$ P (*whileAnno* b $\{s.$ *undefined*$\}$ V c) $(P \cap -b),A$
⟨*proof*⟩

lemma *WhileInvConj* [*intro?*]:
$\Gamma,\Theta\vdash_{/F}$ $\{s.\ P\ s \wedge b\ s\}$ c $\{s.\ P\ s\},A$
$\Longrightarrow \Gamma,\Theta\vdash_{/F}$ $\{s.\ P\ s\}$ (*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 $\{s.\ P\ s\},A$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}$ $\{s.\ P\ s\}$ (*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* ⇒ ('s,'p,'f) *com* ⇒ ('s,'f) *xstate* ⇒ *bool*
(‹-⊢- ↓ -› [*60*,*20*,*60*] *89*)
for Γ::('s,'p,'f) *body*
where
Skip: Γ⊢*Skip* ↓(*Normal s*)

| *Basic*: Γ⊢*Basic f* ↓(*Normal s*)

| *Spec*: Γ⊢*Spec r* ↓(*Normal s*)

| *Guard*: ⟦*s*∈*g*; Γ⊢*c*↓(*Normal s*)⟧
⟹
Γ⊢*Guard f g c*↓(*Normal s*)

| *GuardFault*: *s*∉*g*
⟹
Γ⊢*Guard f g c*↓(*Normal s*)

| *Fault* [*intro*,*simp*]: Γ⊢*c*↓*Fault f*

| *Seq*: $[\![\Gamma\vdash c_1 \downarrow Normal\ s;\ \forall s'.\ \Gamma\vdash\langle c_1, Normal\ s\rangle \Rightarrow s' \longrightarrow \Gamma\vdash c_2\downarrow s']\!]$
$\Longrightarrow$
$\Gamma\vdash Seq\ c_1\ \ c_2\downarrow(Normal\ s)$

| *CondTrue*: $[\![s \in b;\ \Gamma\vdash c_1\downarrow(Normal\ s)]\!]$
$\Longrightarrow$
$\Gamma\vdash Cond\ b\ c_1\ \ c_2\downarrow(Normal\ s)$

| *CondFalse*: $[\![s \notin b;\ \Gamma\vdash c_2\downarrow(Normal\ s)]\!]$
$\Longrightarrow$
$\Gamma\vdash Cond\ b\ c_1\ \ c_2\downarrow(Normal\ s)$

| *WhileTrue*: $[\![s \in b;\ \Gamma\vdash c\downarrow(Normal\ s);$
$\forall s'.\ \Gamma\vdash\langle c, Normal\ s\ \rangle \Rightarrow s' \longrightarrow \Gamma\vdash While\ b\ c\downarrow s']\!]$
$\Longrightarrow$
$\Gamma\vdash While\ b\ c\downarrow(Normal\ s)$

| *WhileFalse*: $[\![s \notin b]\!]$
$\Longrightarrow$
$\Gamma\vdash While\ b\ c\downarrow(Normal\ s)$

| *Call*: $[\![\Gamma\ p{=}Some\ bdy;\Gamma\vdash bdy\downarrow(Normal\ s)]\!]$
$\Longrightarrow$
$\Gamma\vdash Call\ p\downarrow(Normal\ s)$

| *CallUndefined*: $[\![\Gamma\ p = None]\!]$
$\Longrightarrow$
$\Gamma\vdash Call\ p\downarrow(Normal\ s)$

| *Stuck* [*intro*,*simp*]: $\Gamma\vdash c\downarrow Stuck$

| *DynCom*: $[\![\Gamma\vdash(c\ s)\downarrow(Normal\ s)]\!]$
$\Longrightarrow$
$\Gamma\vdash DynCom\ c\downarrow(Normal\ s)$

| *Throw*: $\Gamma\vdash Throw\downarrow(Normal\ s)$

| *Abrupt* [*intro*,*simp*]: $\Gamma\vdash c\downarrow Abrupt\ s$

| *Catch*: $[\![\Gamma\vdash c_1\downarrow Normal\ s;$
$\forall s'.\ \Gamma\vdash\langle c_1, Normal\ s\ \rangle \Rightarrow Abrupt\ s' \longrightarrow \Gamma\vdash c_2\downarrow Normal\ s']\!]$
$\Longrightarrow$
$\Gamma\vdash Catch\ c_1\ \ c_2\downarrow Normal\ s$

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

$\Gamma\vdash$ *Skip* $\downarrow$ *s*
$\Gamma\vdash$ *Guard f g c* $\downarrow$ *s*
$\Gamma\vdash$ *Basic f* $\downarrow$ *s*
$\Gamma\vdash$ *Spec r* $\downarrow$ *s*
$\Gamma\vdash$ *Seq c1 c2* $\downarrow$ *s*
$\Gamma\vdash$ *Cond b c1 c2* $\downarrow$ *s*
$\Gamma\vdash$ *While b c* $\downarrow$ *s*
$\Gamma\vdash$ *Call p* $\downarrow$ *s*
$\Gamma\vdash$ *DynCom c* $\downarrow$ *s*
$\Gamma\vdash$ *Throw* $\downarrow$ *s*
$\Gamma\vdash$ *Catch c1 c2* $\downarrow$ *s*

inductive-cases *terminates-Normal-elim-cases* [*cases set*]:
$\Gamma\vdash$ *Skip* $\downarrow$ *Normal s*
$\Gamma\vdash$ *Guard f g c* $\downarrow$ *Normal s*
$\Gamma\vdash$ *Basic f* $\downarrow$ *Normal s*
$\Gamma\vdash$ *Spec r* $\downarrow$ *Normal s*
$\Gamma\vdash$ *Seq c1 c2* $\downarrow$ *Normal s*
$\Gamma\vdash$ *Cond b c1 c2* $\downarrow$ *Normal s*
$\Gamma\vdash$ *While b c* $\downarrow$ *Normal s*
$\Gamma\vdash$ *Call p* $\downarrow$ *Normal s*
$\Gamma\vdash$ *DynCom c* $\downarrow$ *Normal s*
$\Gamma\vdash$ *Throw* $\downarrow$ *Normal s*
$\Gamma\vdash$ *Catch c1 c2* $\downarrow$ *Normal s*

lemma *terminates-Skip'*: $\Gamma\vdash$ *Skip* $\downarrow$ *s*
$\langle proof \rangle$

lemma *terminates-Call-body*:
Γ *p=Some bdy* $\Longrightarrow\Gamma\vdash$ *Call p* $\downarrow$*s* = $\Gamma\vdash$(*the* (Γ *p*))$\downarrow$*s*
$\langle proof \rangle$

lemma *terminates-Normal-Call-body*:
p $\in$ *dom* $\Gamma \Longrightarrow$
$\Gamma\vdash$ *Call p* $\downarrow$*Normal s* = $\Gamma\vdash$(*the* (Γ *p*))$\downarrow$*Normal s*
$\langle proof \rangle$

lemma *terminates-implies-exec*:
assumes *terminates*: $\Gamma\vdash c\downarrow s$
shows $\exists\, t.\ \Gamma\vdash\langle c,s\rangle \Rightarrow t$
$\langle proof \rangle$

lemma *terminates-block-exn*:
$[\![\Gamma\vdash$ *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*)$]\!]$
$\Longrightarrow \Gamma\vdash$ *block-exn init bdy return result-exn c* $\downarrow$ *Normal s*
$\langle proof \rangle$

lemma *terminates-block*:

⟦Γ⊢*bdy* ↓ *Normal* (*init s*);
 ∀ *t*. Γ⊢⟨*bdy*,*Normal* (*init s*)⟩ ⇒ *Normal t* ⟶ Γ⊢*c s t* ↓ *Normal* (*return s t*)⟧
⟹ Γ⊢*block init bdy return c* ↓ *Normal s*
 ⟨*proof*⟩

lemma *terminates-block-exn-elim* [*cases set*, *consumes 1*]:
assumes *termi*: Γ⊢*block-exn init bdy return result-exn c* ↓ *Normal s*
assumes *e*: ⟦Γ⊢*bdy* ↓ *Normal* (*init s*);
 ∀ *t*. Γ⊢⟨*bdy*,*Normal* (*init s*)⟩ ⇒ *Normal t* ⟶ Γ⊢*c s t* ↓ *Normal* (*return s t*)
 ⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *terminates-block-elim* [*cases set*, *consumes 1*]:
assumes *termi*: Γ⊢*block init bdy return c* ↓ *Normal s*
assumes *e*: ⟦Γ⊢*bdy* ↓ *Normal* (*init s*);
 ∀ *t*. Γ⊢⟨*bdy*,*Normal* (*init s*)⟩ ⇒ *Normal t* ⟶ Γ⊢*c s t* ↓ *Normal* (*return s t*)
 ⟧ ⟹ *P*
shows *P*
 ⟨*proof*⟩

lemma *terminates-call*:
⟦Γ *p* = *Some bdy*; Γ⊢*bdy* ↓ *Normal* (*init s*);
 ∀ *t*. Γ⊢⟨*bdy*,*Normal* (*init s*)⟩ ⇒ *Normal t* ⟶ Γ⊢*c s t* ↓ *Normal* (*return s t*)⟧
⟹ Γ⊢*call init p return c* ↓ *Normal s*
 ⟨*proof*⟩

lemma *terminates-call-exn*:
⟦Γ *p* = *Some bdy*; Γ⊢*bdy* ↓ *Normal* (*init s*);
 ∀ *t*. Γ⊢⟨*bdy*,*Normal* (*init s*)⟩ ⇒ *Normal t* ⟶ Γ⊢*c s t* ↓ *Normal* (*return s t*)⟧
⟹ Γ⊢*call-exn init p return result-exn c* ↓ *Normal s*
 ⟨*proof*⟩

lemma *terminates-callUndefined*:
⟦Γ *p* = *None*⟧
⟹ Γ⊢*call init p return result* ↓ *Normal s*
 ⟨*proof*⟩

lemma *terminates-call-exnUndefined*:
⟦Γ *p* = *None*⟧
⟹ Γ⊢*call-exn init p return result-exn result* ↓ *Normal s*
 ⟨*proof*⟩

lemma *terminates-call-exn-elim* [*cases set*, *consumes 1*]:
assumes *termi*: Γ⊢*call-exn init p return result-exn c* ↓ *Normal s*
assumes *bdy*: ⋀*bdy*. ⟦Γ *p* = *Some bdy*; Γ⊢*bdy* ↓ *Normal* (*init s*);

$\forall t.\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow Normal\ t \longrightarrow \Gamma\vdash c\ s\ t \downarrow Normal\ (return\ s\ t)]\!]$
$\Longrightarrow P$
assumes *undef*: $[\![\Gamma\ p = None]\!] \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *terminates-call-elim* [*cases set*, *consumes 1*]:
assumes *termi*: $\Gamma\vdash call\ init\ p\ return\ c \downarrow Normal\ s$
assumes *bdy*: $\bigwedge bdy.\ [\![\Gamma\ p = Some\ bdy;\ \Gamma\vdash bdy \downarrow Normal\ (init\ s);$
$\forall t.\ \Gamma\vdash\langle bdy,Normal\ (init\ s)\rangle \Rightarrow Normal\ t \longrightarrow \Gamma\vdash c\ s\ t \downarrow Normal\ (return\ s\ t)]\!]$
$\Longrightarrow P$
assumes *undef*: $[\![\Gamma\ p = None]\!] \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *terminates-dynCall*:
$[\![\Gamma\vdash call\ init\ (p\ s)\ return\ c \downarrow Normal\ s]\!]$
$\Longrightarrow \Gamma\vdash dynCall\ init\ p\ return\ c \downarrow Normal\ s$
⟨*proof*⟩

lemma *terminates-guards*: $\Gamma\vdash c \downarrow Normal\ s \Longrightarrow \Gamma\vdash guards\ gs\ c \downarrow Normal\ s$
⟨*proof*⟩

lemma *terminates-guards-Fault*: $find\ (\lambda(f,\ g).\ s \notin g)\ gs = Some\ (f,\ g) \Longrightarrow \Gamma\vdash guards\ gs\ c \downarrow Normal\ s$
⟨*proof*⟩

lemma *terminates-maybe-guard-Fault*: $s \notin g \Longrightarrow \Gamma\vdash maybe\text{-}guard\ f\ g\ c \downarrow Normal\ s$
⟨*proof*⟩

lemma *terminates-guards-DynCom*: $\Gamma\vdash(c\ s) \downarrow Normal\ s \Longrightarrow \Gamma\vdash guards\ gs\ (DynCom\ c) \downarrow Normal\ s$
⟨*proof*⟩

lemma *terminates-maybe-guard-DynCom*: $\Gamma\vdash(c\ s) \downarrow Normal\ s \Longrightarrow \Gamma\vdash maybe\text{-}guard\ f\ g\ (DynCom\ c) \downarrow Normal\ s$
⟨*proof*⟩

lemma *terminates-dynCall-exn*:
$[\![\Gamma\vdash call\text{-}exn\ init\ (p\ s)\ return\ result\text{-}exn\ c \downarrow Normal\ s]\!]$
$\Longrightarrow \Gamma\vdash dynCall\text{-}exn\ f\ g\ init\ p\ return\ result\text{-}exn\ c \downarrow Normal\ s$
⟨*proof*⟩

lemma *terminates-dynCall-elim* [*cases set*, *consumes 1*]:
assumes *termi*: $\Gamma\vdash dynCall\ init\ p\ return\ c \downarrow Normal\ s$
assumes $[\![\Gamma\vdash call\ init\ (p\ s)\ return\ c \downarrow Normal\ s]\!] \Longrightarrow P$
shows P

⟨*proof*⟩

lemma *terminates-guards-elim* [*cases set*, *consumes 1*, *case-names noFault someFault*]:
  assumes *termi*: $\Gamma\vdash$*guards gs c* $\downarrow$ *Normal s*
  assumes *noFault*: $[\![\forall f\ g.\ (f,\ g) \in set\ gs \longrightarrow s \in g; \Gamma\vdash c \downarrow Normal\ s]\!] \Longrightarrow P$
  assumes *someFault*: $\bigwedge f\ g.\ find\ (\lambda(f,g).\ s \notin g)\ gs = Some\ (f,\ g) \Longrightarrow P$
  shows *P*
  ⟨*proof*⟩

lemma *terminates-maybe-guard-elim* [*cases set*, *consumes 1*, *case-names noFault someFault*]:
  assumes *termi*: $\Gamma\vdash$*maybe-guard f g c* $\downarrow$ *Normal s*
  assumes *noFault*: $[\![s \in g; \Gamma\vdash c \downarrow Normal\ s]\!] \Longrightarrow P$
  assumes *someFault*: $s \notin g \Longrightarrow P$
  shows *P*
  ⟨*proof*⟩

lemma *terminates-dynCall-exn-elim* [*cases set*, *consumes 1*, *case-names noFault someFault*]:
assumes *termi*: $\Gamma\vdash$*dynCall-exn f g init p return result-exn c* $\downarrow$ *Normal s*
assumes *noFault*: $[\![s \in g;$
$\Gamma\vdash$*call-exn init (p s) return result-exn c* $\downarrow$ *Normal s*$]\!] \Longrightarrow P$
assumes *someFault*: $s \notin g \Longrightarrow P$
shows *P*
⟨*proof*⟩

7.2 Lemmas about *sequence*, *flatten* and *Language.normalize*

lemma *terminates-sequence-app*:
  $\bigwedge s.$ $[\![\Gamma\vdash$*sequence Seq xs* $\downarrow$ *Normal s*;
      $\forall s'.\ \Gamma\vdash\langle$*sequence Seq xs*,*Normal s* $\rangle \Rightarrow s' \longrightarrow\ \Gamma\vdash$*sequence Seq ys* $\downarrow s']\!]$
$\Longrightarrow \Gamma\vdash$*sequence Seq (xs @ ys)* $\downarrow$ *Normal s*
⟨*proof*⟩

lemma *terminates-sequence-appD*:
  $\bigwedge s.$ $\Gamma\vdash$*sequence Seq (xs @ ys)* $\downarrow$ *Normal s*
   $\Longrightarrow \Gamma\vdash$*sequence Seq xs* $\downarrow$ *Normal s* $\wedge$
     $(\forall s'.\ \Gamma\vdash\langle$*sequence Seq xs*,*Normal s* $\rangle \Rightarrow s' \longrightarrow\ \Gamma\vdash$*sequence Seq ys* $\downarrow s')$
⟨*proof*⟩

lemma *terminates-sequence-appE* [*consumes 1*]:
  $[\![\Gamma\vdash$*sequence Seq (xs @ ys)* $\downarrow$ *Normal s*;
   $[\![\Gamma\vdash$*sequence Seq xs* $\downarrow$ *Normal s*;
    $\forall s'.\ \Gamma\vdash\langle$*sequence Seq xs*,*Normal s* $\rangle \Rightarrow s' \longrightarrow\ \Gamma\vdash$*sequence Seq ys* $\downarrow s']\!] \Longrightarrow P]\!]$
  $\Longrightarrow P$
  ⟨*proof*⟩

lemma *terminates-to-terminates-sequence-flatten*:

assumes *termi*: $\Gamma\vdash c\downarrow s$
shows $\Gamma\vdash$*sequence Seq* (*flatten c*)$\downarrow s$
⟨*proof*⟩

lemma *terminates-to-terminates-normalize*:
assumes *termi*: $\Gamma\vdash c\downarrow s$
shows $\Gamma\vdash$*normalize c*$\downarrow s$
⟨*proof*⟩

lemma *terminates-sequence-flatten-to-terminates*:
shows $\bigwedge s.$ $\Gamma\vdash$*sequence Seq* (*flatten c*)$\downarrow s \Longrightarrow \Gamma\vdash c\downarrow s$
⟨*proof*⟩

lemma *terminates-normalize-to-terminates*:
shows $\bigwedge s.$ $\Gamma\vdash$*normalize c*$\downarrow s \Longrightarrow \Gamma\vdash c\downarrow s$
⟨*proof*⟩

lemma *terminates-iff-terminates-normalize*:
$\Gamma\vdash$*normalize c*$\downarrow s = \Gamma\vdash c\downarrow s$
⟨*proof*⟩

7.3 Lemmas about *strip-guards*

lemma *terminates-strip-guards-to-terminates*: $\bigwedge s.$ $\Gamma\vdash$*strip-guards F c*$\downarrow s \Longrightarrow \Gamma\vdash c\downarrow s$
⟨*proof*⟩

lemma *terminates-strip-to-terminates*:
assumes *termi-strip*: *strip F* $\Gamma\vdash c\downarrow s$
shows $\Gamma\vdash c\downarrow s$
⟨*proof*⟩

7.4 Lemmas about $c_1 \cap_g c_2$

lemma *inter-guards-terminates*:
$\bigwedge c\ c2\ s.$ ⟦($c1 \cap_g c2$) = *Some c*; $\Gamma\vdash c1\downarrow s$ ⟧
$\Longrightarrow \Gamma\vdash c\downarrow s$
⟨*proof*⟩

lemma *inter-guards-terminates'*:
assumes *c*: ($c1 \cap_g c2$) = *Some c*
assumes *termi-c2*: $\Gamma\vdash c2\downarrow s$
shows $\Gamma\vdash c\downarrow s$
⟨*proof*⟩

7.5 Lemmas about *mark-guards*

lemma *terminates-to-terminates-mark-guards*:
assumes *termi*: $\Gamma\vdash c\downarrow s$
shows $\Gamma\vdash$*mark-guards f c*$\downarrow s$
⟨*proof*⟩

lemma *terminates-mark-guards-to-terminates-Normal*:
$\bigwedge s.\ \Gamma \vdash \textit{mark-guards } f\ c \downarrow \textit{Normal } s \Longrightarrow \Gamma \vdash c \downarrow \textit{Normal } s$
⟨*proof*⟩

lemma *terminates-mark-guards-to-terminates*:
$\Gamma \vdash \textit{mark-guards } f\ c \downarrow s \Longrightarrow \Gamma \vdash c \downarrow\ s$
⟨*proof*⟩

7.6 Lemmas about *merge-guards*

lemma *terminates-to-terminates-merge-guards*:
assumes *termi*: $\Gamma \vdash c \downarrow s$
shows $\Gamma \vdash \textit{merge-guards } c \downarrow s$
⟨*proof*⟩

lemma *terminates-merge-guards-to-terminates-Normal*:
shows $\bigwedge s.\ \Gamma \vdash \textit{merge-guards } c \downarrow \textit{Normal } s \Longrightarrow \Gamma \vdash c \downarrow \textit{Normal } s$
⟨*proof*⟩

lemma *terminates-merge-guards-to-terminates*:
$\Gamma \vdash \textit{merge-guards } c \downarrow\ s \Longrightarrow \Gamma \vdash c \downarrow\ s$
⟨*proof*⟩

theorem *terminates-iff-terminates-merge-guards*:
$\Gamma \vdash c \downarrow\ s = \Gamma \vdash \textit{merge-guards } c \downarrow\ s$
⟨*proof*⟩

7.7 Lemmas about $c_1 \subseteq_g c_2$

lemma *terminates-fewer-guards-Normal*:
shows $\bigwedge c\ s.\ [\![\Gamma \vdash c' \downarrow \textit{Normal } s;\ c \subseteq_g c';\ \Gamma \vdash \langle c', \textit{Normal } s\ \rangle \Rightarrow \notin \textit{Fault } `\ \textit{UNIV}]\!]$
$\Longrightarrow \Gamma \vdash c \downarrow \textit{Normal } s$
⟨*proof*⟩

theorem *terminates-fewer-guards*:
shows $[\![\Gamma \vdash c' \downarrow s;\ c \subseteq_g c';\ \Gamma \vdash \langle c', s\ \rangle \Rightarrow \notin \textit{Fault } `\ \textit{UNIV}]\!]$
$\Longrightarrow \Gamma \vdash c \downarrow s$
⟨*proof*⟩

lemma *terminates-noFault-strip-guards*:
assumes *termi*: $\Gamma \vdash c \downarrow \textit{Normal } s$
shows $[\![\Gamma \vdash \langle c, \textit{Normal } s\ \rangle \Rightarrow \notin \textit{Fault } `\ F]\!] \Longrightarrow \Gamma \vdash \textit{strip-guards } F\ c \downarrow \textit{Normal } s$
⟨*proof*⟩

7.8 Lemmas about *strip-guards*

lemma *terminates-noFault-strip*:
assumes *termi*: $\Gamma \vdash c \downarrow \textit{Normal } s$
shows $[\![\Gamma \vdash \langle c, \textit{Normal } s\ \rangle \Rightarrow \notin \textit{Fault } `\ F]\!] \Longrightarrow \textit{strip } F\ \Gamma \vdash c \downarrow \textit{Normal } s$

⟨*proof*⟩

7.9 Miscellaneous

lemma *terminates-while-lemma*:
assumes *termi*: $\Gamma\vdash w\downarrow fk$
shows $\bigwedge k\ b\ c.\ [\![fk = Normal\ (f\ k);\ w{=}While\ b\ c;$
$\forall\, i.\ \Gamma\vdash\langle c,Normal\ (f\ i)\ \rangle \Rightarrow Normal\ (f\ (Suc\ i))]\!]$
$\Longrightarrow \exists\, i.\ f\ i \notin b$
⟨*proof*⟩

lemma *terminates-while*:
$[\![\Gamma\vdash(While\ b\ c)\downarrow Normal\ (f\ k);$
$\forall\, i.\ \Gamma\vdash\langle c,Normal\ (f\ i)\ \rangle \Rightarrow Normal\ (f\ (Suc\ i))]\!]$
$\Longrightarrow \exists\, i.\ f\ i \notin b$
⟨*proof*⟩

lemma *wf-terminates-while*:
$wf\ \{(t,s).\ \Gamma\vdash(While\ b\ c)\downarrow Normal\ s \wedge s{\in}b \wedge$
$\Gamma\vdash\langle c,Normal\ s\ \rangle \Rightarrow Normal\ t\}$
⟨*proof*⟩

lemma *terminates-restrict-to-terminates*:
assumes *terminates-res*: $\Gamma|_M\vdash\ c \downarrow s$
assumes *not-Stuck*: $\Gamma|_M\vdash\langle c,s\ \rangle \Rightarrow\notin\{Stuck\}$
shows $\Gamma\vdash\ c \downarrow s$
⟨*proof*⟩

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*:: $('s,'p,'f)com \Rightarrow ('s,'p,'f)com$
where
redex Skip = Skip |
redex (*Basic f*) = (*Basic f*) |
redex (*Spec r*) = (*Spec r*) |
redex (*Seq* c_1 c_2) = *redex* c_1 |
redex (*Cond b* c_1 c_2) = (*Cond b* c_1 c_2) |
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 $step::[('s,'p,'f)\ body,('s,'p,'f)\ config,('s,'p,'f)\ config] \Rightarrow bool$
(‹-⊢ (- →/ -)› [*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*: $[\![s\in b]\!]$
$\Longrightarrow$
$\Gamma\vdash(While\ b\ c,Normal\ s) \rightarrow (Seq\ c\ (While\ b\ c),Normal\ s)$

| *WhileFalse*: $[\![s\notin b]\!]$
$\Longrightarrow$
$\Gamma\vdash(While\ b\ c,Normal\ s) \rightarrow (Skip,Normal\ s)$

| *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*: $[\![\Gamma\vdash(c_1,s) \rightarrow (c_1{}',s')]\!]$
$\Longrightarrow$
$\Gamma\vdash(Catch\ c_1\ c_2,s) \rightarrow (Catch\ c_1{}'\ c_2,s')$

| *CatchThrow*: $\Gamma\vdash(\mathit{Catch\ Throw}\ c_2,\mathit{Normal}\ s) \rightarrow (c_2,\mathit{Normal}\ s)$
| *CatchSkip*: $\Gamma\vdash(\mathit{Catch\ Skip}\ c_2,s) \rightarrow (\mathit{Skip},s)$

| *FaultProp*: $[\![c\neq \mathit{Skip};\ \mathit{redex}\ c = c]\!] \Longrightarrow \Gamma\vdash(c,\mathit{Fault}\ f) \rightarrow (\mathit{Skip},\mathit{Fault}\ f)$
| *StuckProp*: $[\![c\neq \mathit{Skip};\ \mathit{redex}\ c = c]\!] \Longrightarrow \Gamma\vdash(c,\mathit{Stuck}) \rightarrow (\mathit{Skip},\mathit{Stuck})$
| *AbruptProp*: $[\![c\neq \mathit{Skip};\ \mathit{redex}\ c = c]\!] \Longrightarrow \Gamma\vdash(c,\mathit{Abrupt}\ f) \rightarrow (\mathit{Skip},\mathit{Abrupt}\ f)$

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*]:
$\Gamma\vdash(\mathit{Skip},s) \rightarrow u$
$\Gamma\vdash(\mathit{Guard}\ f\ g\ c,s) \rightarrow u$
$\Gamma\vdash(\mathit{Basic}\ f,s) \rightarrow u$
$\Gamma\vdash(\mathit{Spec}\ r,s) \rightarrow u$
$\Gamma\vdash(\mathit{Seq}\ c1\ c2,s) \rightarrow u$
$\Gamma\vdash(\mathit{Cond}\ b\ c1\ c2,s) \rightarrow u$
$\Gamma\vdash(\mathit{While}\ b\ c,s) \rightarrow u$
$\Gamma\vdash(\mathit{Call}\ p,s) \rightarrow u$
$\Gamma\vdash(\mathit{DynCom}\ c,s) \rightarrow u$
$\Gamma\vdash(\mathit{Throw},s) \rightarrow u$
$\Gamma\vdash(\mathit{Catch}\ c1\ c2,s) \rightarrow u$

inductive-cases *step-Normal-elim-cases* [*cases set*]:
$\Gamma\vdash(\mathit{Skip},\mathit{Normal}\ s) \rightarrow u$
$\Gamma\vdash(\mathit{Guard}\ f\ g\ c,\mathit{Normal}\ s) \rightarrow u$
$\Gamma\vdash(\mathit{Basic}\ f,\mathit{Normal}\ s) \rightarrow u$
$\Gamma\vdash(\mathit{Spec}\ r,\mathit{Normal}\ s) \rightarrow u$
$\Gamma\vdash(\mathit{Seq}\ c1\ c2,\mathit{Normal}\ s) \rightarrow u$
$\Gamma\vdash(\mathit{Cond}\ b\ c1\ c2,\mathit{Normal}\ s) \rightarrow u$
$\Gamma\vdash(\mathit{While}\ b\ c,\mathit{Normal}\ s) \rightarrow u$
$\Gamma\vdash(\mathit{Call}\ p,\mathit{Normal}\ s) \rightarrow u$
$\Gamma\vdash(\mathit{DynCom}\ c,\mathit{Normal}\ s) \rightarrow u$
$\Gamma\vdash(\mathit{Throw},\mathit{Normal}\ s) \rightarrow u$
$\Gamma\vdash(\mathit{Catch}\ c1\ c2,\mathit{Normal}\ s) \rightarrow u$

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* ⇒ *bool* **where**
$\mathit{final}\ \mathit{cfg} = (\mathit{fst}\ \mathit{cfg}{=}\mathit{Skip} \lor (\mathit{fst}\ \mathit{cfg}{=}\mathit{Throw} \land (\exists s.\ \mathit{snd}\ \mathit{cfg}{=}\mathit{Normal}\ s)))$

abbreviation
step-rtrancl :: $[('s,'p,'f)$ *body*$,('s,'p,'f)$ *config*$,('s,'p,'f)$ *config*$] \Rightarrow$ *bool*
(‹-⊢ (- $\rightarrow^*$/ -)› [*81*,*81*,*81*] *100*)
where
$\Gamma\vdash cf0 \rightarrow^* cf1 \equiv (CONST\ step\ \Gamma)^{**}\ cf0\ cf1$
abbreviation
step-trancl :: $[('s,'p,'f)$ *body*$,('s,'p,'f)$ *config*$,('s,'p,'f)$ *config*$] \Rightarrow$ *bool*
(‹-⊢ (- $\rightarrow^+$/ -)› [*81*,*81*,*81*] *100*)
where
$\Gamma\vdash cf0 \rightarrow^+ cf1 \equiv (CONST\ step\ \Gamma)^{++}\ cf0\ cf1$

8.2 Structural Properties of Small Step Computations

lemma *redex-not-Seq*: $redex\ c = Seq\ c1\ c2 \Longrightarrow P$
⟨*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*: $\Gamma\vdash cfg \rightarrow cfg'$
shows $final\ cfg \Longrightarrow P$
⟨*proof*⟩

lemma *step-Abrupt*:
assumes *step*: $\Gamma\vdash (c,\ s) \rightarrow (c',\ s')$
shows $\bigwedge x.\ s{=}Abrupt\ x \Longrightarrow s'{=}Abrupt\ x$
⟨*proof*⟩

lemma *step-Fault*:
assumes *step*: $\Gamma\vdash (c,\ s) \rightarrow (c',\ s')$
shows $\bigwedge f.\ s{=}Fault\ f \Longrightarrow s'{=}Fault\ f$
⟨*proof*⟩

lemma *step-Stuck*:
assumes *step*: $\Gamma\vdash (c,\ s) \rightarrow (c',\ s')$
shows $\bigwedge f.\ s{=}Stuck \Longrightarrow s'{=}Stuck$
⟨*proof*⟩

lemma *SeqSteps*:
assumes *steps*: $\Gamma\vdash cfg_1 \rightarrow^* cfg_2$
shows $\bigwedge\ c_1\ s\ c_1'\ s'.\ [\![cfg_1 = (c_1,s);cfg_2{=}(c_1',s')]\!]$
$\Longrightarrow \Gamma\vdash(Seq\ c_1\ c_2,s) \rightarrow^* (Seq\ c_1'\ c_2,\ s')$
⟨*proof*⟩

lemma *CatchSteps*:
  assumes *steps*: $\Gamma\vdash cfg_1 \rightarrow^* cfg_2$
  shows $\bigwedge c_1\ s\ c_1'\ s'.\ [\![cfg_1 = (c_1,s);\ cfg_2=(c_1',s')]\!]$
  $\Longrightarrow \Gamma\vdash(Catch\ c_1\ c_2,s) \rightarrow^* (Catch\ c_1'\ c_2,\ s')$
⟨*proof*⟩

lemma *steps-Fault*: $\Gamma\vdash (c,\ Fault\ f) \rightarrow^* (Skip,\ Fault\ f)$
⟨*proof*⟩

lemma *steps-Stuck*: $\Gamma\vdash (c,\ Stuck) \rightarrow^* (Skip,\ Stuck)$
⟨*proof*⟩

lemma *steps-Abrupt*: $\Gamma\vdash (c,\ Abrupt\ s) \rightarrow^* (Skip,\ Abrupt\ s)$
⟨*proof*⟩

lemma *step-Fault-prop*:
  assumes *step*: $\Gamma\vdash (c,\ s) \rightarrow (c',\ s')$
  shows $\bigwedge f.\ s=Fault\ f \Longrightarrow s'=Fault\ f$
⟨*proof*⟩

lemma *step-Abrupt-prop*:
  assumes *step*: $\Gamma\vdash (c,\ s) \rightarrow (c',\ s')$
  shows $\bigwedge x.\ s=Abrupt\ x \Longrightarrow s'=Abrupt\ x$
⟨*proof*⟩

lemma *step-Stuck-prop*:
  assumes *step*: $\Gamma\vdash (c,\ s) \rightarrow (c',\ s')$
  shows $s=Stuck \Longrightarrow s'=Stuck$
⟨*proof*⟩

lemma *steps-Fault-prop*:
  assumes *step*: $\Gamma\vdash (c,\ s) \rightarrow^* (c',\ s')$
  shows $s=Fault\ f \Longrightarrow s'=Fault\ f$
⟨*proof*⟩

lemma *steps-Abrupt-prop*:
  assumes *step*: $\Gamma\vdash (c,\ s) \rightarrow^* (c',\ s')$
  shows $s=Abrupt\ t \Longrightarrow s'=Abrupt\ t$
⟨*proof*⟩

lemma *steps-Stuck-prop*:
  assumes *step*: $\Gamma\vdash (c,\ s) \rightarrow^* (c',\ s')$
  shows $s=Stuck \Longrightarrow s'=Stuck$
⟨*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* $c'=Throw \wedge t'=Normal\ x$
| - $\Rightarrow$ $c'=Skip \wedge t'=t$)
⟨*proof*⟩

corollary *exec-impl-steps-Normal*:
assumes *exec*: $\Gamma\vdash\langle c,s\rangle \Rightarrow Normal\ t$
shows $\Gamma\vdash(c,s) \rightarrow^* (Skip,\ Normal\ t)$
⟨*proof*⟩

corollary *exec-impl-steps-Normal-Abrupt*:
assumes *exec*: $\Gamma\vdash\langle c,Normal\ s\rangle \Rightarrow Abrupt\ t$
shows $\Gamma\vdash(c,Normal\ s) \rightarrow^* (Throw,\ Normal\ t)$
⟨*proof*⟩

corollary *exec-impl-steps-Abrupt-Abrupt*:
assumes *exec*: $\Gamma\vdash\langle c,Abrupt\ t\rangle \Rightarrow Abrupt\ t$
shows $\Gamma\vdash(c,Abrupt\ t) \rightarrow^* (Skip,\ Abrupt\ t)$
⟨*proof*⟩

corollary *exec-impl-steps-Fault*:
assumes *exec*: $\Gamma\vdash\langle c,s\rangle \Rightarrow Fault\ f$
shows $\Gamma\vdash(c,s) \rightarrow^* (Skip,\ Fault\ f)$
⟨*proof*⟩

corollary *exec-impl-steps-Stuck*:
assumes *exec*: $\Gamma\vdash\langle c,s\rangle \Rightarrow Stuck$
shows $\Gamma\vdash(c,s) \rightarrow^* (Skip,\ Stuck)$
⟨*proof*⟩

lemma *step-Abrupt-end*:
assumes *step*: $\Gamma\vdash (c_1,\ s) \rightarrow (c_1',\ s')$
shows $s'=Abrupt\ x \Longrightarrow s=Abrupt\ x$
⟨*proof*⟩

lemma *step-Stuck-end*:
assumes *step*: $\Gamma\vdash (c_1,\ s) \rightarrow (c_1',\ s')$
shows $s'=Stuck \Longrightarrow$
$s=Stuck \vee$
$(\exists\, r\ x.\ redex\ c_1 = Spec\ r \wedge s=Normal\ x \wedge (\forall\, t.\ (x,t)\notin r)) \vee$
$(\exists\, p\ x.\ redex\ c_1=Call\ p \wedge s=Normal\ x \wedge \Gamma\ p = None)$
⟨*proof*⟩

lemma *step-Fault-end*:
assumes *step*: $\Gamma\vdash (c_1,\ s) \rightarrow (c_1',\ s')$
shows $s'=Fault\ f \Longrightarrow$
$s=Fault\ f \vee$

$(\exists\, g\ c\ x.\ redex\ c_1 = Guard\ f\ g\ c \land s{=}Normal\ x \land x \notin g)$
⟨*proof*⟩

lemma *exec-redex-Stuck*:
$\Gamma\vdash\langle redex\ c,s\rangle \Rightarrow Stuck \Longrightarrow \Gamma\vdash\langle c,s\rangle \Rightarrow Stuck$
⟨*proof*⟩

lemma *exec-redex-Fault*:
$\Gamma\vdash\langle redex\ c,s\rangle \Rightarrow Fault\ f \Longrightarrow \Gamma\vdash\langle c,s\rangle \Rightarrow Fault\ f$
⟨*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*: $\Gamma\vdash(c,s) \rightarrow^* (Skip,t)$
shows $\Gamma\vdash\langle c,s\rangle \Rightarrow t$
⟨*proof*⟩

theorem *steps-Throw-impl-exec*:
assumes *steps*: $\Gamma\vdash(c,s) \rightarrow^* (Throw,Normal\ t)$
shows $\Gamma\vdash\langle c,s\rangle \Rightarrow Abrupt\ t$
⟨*proof*⟩

8.4 Infinite Computations: $\Gamma\vdash(c,\ s) \rightarrow \ldots(\infty)$

definition *inf*:: $('s,'p,'f)\ body \Rightarrow ('s,'p,'f)\ config \Rightarrow bool$
(‹-⊢ - → …′(∞′)› [*60*,*80*] *100*) **where**
$\Gamma\vdash\ cfg \rightarrow \ldots(\infty) \equiv (\exists f.\ f\ (0{::}nat) = cfg \land (\forall i.\ \Gamma\vdash f\ i \rightarrow f\ (i{+}1)))$

lemma *not-infI*: $[\![\bigwedge f.\ [\![f\ 0 = cfg;\ \bigwedge i.\ \Gamma\vdash f\ i \rightarrow f\ (Suc\ i)]\!] \Longrightarrow False]\!]$
$\Longrightarrow \neg\Gamma\vdash\ cfg \rightarrow \ldots(\infty)$
⟨*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*⟩

⟨*ML*⟩

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*⟩

definition *head-com*:: $('s,'p,'f)$ *com* ⇒ $('s,'p,'f)$ *com*
where
head-com c =
  (*case c of*
     Seq c_1 c_2 ⇒ c_1
   | *Catch* c_1 c_2 ⇒ c_1
   | - ⇒ *c*)

definition *head*:: $('s,'p,'f)$ *config* ⇒ $('s,'p,'f)$ *config*
  where *head cfg* = (*head-com* (*fst cfg*), *snd cfg*)

lemma *le-Suc-cases*: $[\![\bigwedge i.\ [\![i < k]\!] \Longrightarrow P\ i;\ P\ k]\!] \Longrightarrow \forall i<(Suc\ k).\ P\ i$
  ⟨*proof*⟩

lemma *redex-Seq-False*: $\bigwedge c'\ c''.\ (redex\ c = Seq\ c''\ c') = False$
  ⟨*proof*⟩

lemma *redex-Catch-False*: $\bigwedge c'\ c''.\ (redex\ c = Catch\ c''\ c') = False$
  ⟨*proof*⟩

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*: $\forall i<k.\ \neg\ final\ (head\ (f\ i))$
  shows $\forall i<k.\ (\exists c'\ s'.\ f\ (i + 1) = (Seq\ c'\ c_2,\ s')) \wedge$
        $\Gamma\vdash head\ (f\ i) \rightarrow head\ (f\ (i+1))$
     (**is** $\forall i<k.\ ?P\ i$)
⟨*proof*⟩

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*: $\forall i<k.\ \neg\ final\ (head\ (f\ i))$
  shows $\forall i<k.\ (\exists c'\ s'.\ f\ (i + 1) = (Catch\ c'\ c_2,\ s')) \wedge$
        $\Gamma\vdash head\ (f\ i) \rightarrow head\ (f\ (i+1))$
     (**is** $\forall i<k.\ ?P\ i$)
⟨*proof*⟩

lemma *no-inf-Throw*: $\neg\ \Gamma\vdash(\mathit{Throw},s) \to \ldots(\infty)$
$\langle proof \rangle$

lemma *split-inf-Seq*:
assumes *inf-comp*: $\Gamma\vdash(\mathit{Seq}\ c_1\ c_2,s) \to \ldots(\infty)$
shows $\Gamma\vdash(c_1,s) \to \ldots(\infty) \lor$
$(\exists s'.\ \Gamma\vdash(c_1,s) \to^* (\mathit{Skip},s') \land \Gamma\vdash(c_2,s') \to \ldots(\infty))$
$\langle proof \rangle$

lemma *split-inf-Catch*:
assumes *inf-comp*: $\Gamma\vdash(\mathit{Catch}\ c_1\ c_2,s) \to \ldots(\infty)$
shows $\Gamma\vdash(c_1,s) \to \ldots(\infty) \lor$
$(\exists s'.\ \Gamma\vdash(c_1,s) \to^* (\mathit{Throw},\mathit{Normal}\ s') \land \Gamma\vdash(c_2,\mathit{Normal}\ s') \to \ldots(\infty))$
$\langle proof \rangle$

lemma *Skip-no-step*: $\Gamma\vdash(\mathit{Skip},s) \to \mathit{cfg} \Longrightarrow P$
$\langle proof \rangle$

lemma *not-inf-Stuck*: $\neg\ \Gamma\vdash(c,\mathit{Stuck}) \to \ldots(\infty)$
$\langle proof \rangle$

lemma *not-inf-Fault*: $\neg\ \Gamma\vdash(c,\mathit{Fault}\ x) \to \ldots(\infty)$
$\langle proof \rangle$

lemma *not-inf-Abrupt*: $\neg\ \Gamma\vdash(c,\mathit{Abrupt}\ s) \to \ldots(\infty)$
$\langle proof \rangle$

theorem *terminates-impl-no-infinite-computation*:
assumes *termi*: $\Gamma\vdash c \downarrow s$
shows $\neg\ \Gamma\vdash(c,s) \to \ldots(\infty)$
$\langle proof \rangle$

definition
termi-call-steps :: $('s,'p,'f)\ \mathit{body} \Rightarrow (('s \times 'p) \times ('s \times 'p))\mathit{set}$
where
termi-call-steps $\Gamma =$
$\{((t,q),(s,p)).\ \Gamma\vdash \mathit{Call}\ p{\downarrow}\mathit{Normal}\ s \land$
$(\exists c.\ \Gamma\vdash(\mathit{Call}\ p,\mathit{Normal}\ s) \to^+ (c,\mathit{Normal}\ t) \land \mathit{redex}\ c = \mathit{Call}\ q)\}$

primrec *subst-redex*:: $('s,'p,'f)\mathit{com} \Rightarrow ('s,'p,'f)\mathit{com} \Rightarrow ('s,'p,'f)\mathit{com}$
where
subst-redex Skip $c = c$ |
subst-redex $(\mathit{Basic}\ f)\ c = c$ |
subst-redex $(\mathit{Spec}\ r)\ c = c$ |
subst-redex $(\mathit{Seq}\ c_1\ c_2)\ c = \mathit{Seq}\ (\mathit{subst\text{-}redex}\ c_1\ c)\ c_2$ |
subst-redex $(\mathit{Cond}\ b\ c_1\ c_2)\ c = c$ |

subst-redex (*While b c'*) *c* = *c* |
subst-redex (*Call p*) *c* = *c* |
subst-redex (*DynCom d*) *c* = *c* |
subst-redex (*Guard f b c'*) *c* = *c* |
subst-redex (*Throw*) *c* = *c* |
subst-redex (*Catch* c_1 c_2) *c* = *Catch* (*subst-redex* c_1 *c*) c_2

lemma *subst-redex-redex*:
subst-redex c (*redex c*) = *c*
⟨*proof*⟩

lemma *redex-subst-redex*: *redex* (*subst-redex c r*) = *redex r*
⟨*proof*⟩

lemma *step-redex'*:
shows Γ⊢(*redex c*,*s*) → (*r'*,*s'*) ⟹ Γ⊢(*c*,*s*) → (*subst-redex c r'*,*s'*)
⟨*proof*⟩

lemma *step-redex*:
shows Γ⊢(*r*,*s*) → (*r'*,*s'*) ⟹ Γ⊢(*subst-redex c r*,*s*) → (*subst-redex c r'*,*s'*)
⟨*proof*⟩

lemma *steps-redex*:
assumes *steps*: Γ⊢ (*r*, *s*) $\rightarrow^*$ (*r'*, *s'*)
shows ⋀*c*. Γ⊢(*subst-redex c r*,*s*) $\rightarrow^*$ (*subst-redex c r'*,*s'*)
⟨*proof*⟩

⟨*ML*⟩

lemma *steps-redex'*:
assumes *steps*: Γ⊢ (*r*, *s*) $\rightarrow^+$ (*r'*, *s'*)
shows ⋀*c*. Γ⊢(*subst-redex c r*,*s*) $\rightarrow^+$ (*subst-redex c r'*,*s'*)
⟨*proof*⟩

primrec *seq*:: (*nat* ⇒ (*'s*,*'p*,*'f*)*com*) ⇒ *'p* ⇒ *nat* ⇒ (*'s*,*'p*,*'f*)*com*
where
seq c p 0 = *Call p* |
seq c p (*Suc i*) = *subst-redex* (*seq c p i*) (*c i*)

lemma *renumber'*:
assumes *f*: ∀ *i*. (*a*,*f i*) ∈ r^* ∧ (*f i*,*f*(*Suc i*)) ∈ *r*
assumes *a-b*: (*a*,*b*) ∈ r^*
shows *b* = *f 0* ⟹ (∃ *f*. *f 0* = *a* ∧ (∀ *i*. (*f i*, *f*(*Suc i*)) ∈ *r*))
⟨*proof*⟩

lemma *renumber*:
∀ *i*. (*a*,*f i*) ∈ r^* ∧ (*f i*,*f*(*Suc i*)) ∈ *r*

$\Longrightarrow \exists f.\ f\ 0 = a \wedge (\forall i.\ (f\ i,\ f(Suc\ i)) \in r)$
⟨proof⟩

lemma *lem*:
$\forall y.\ r^{++}\ a\ y \longrightarrow P\ a \longrightarrow P\ y$
$\Longrightarrow ((b,a) \in \{(y,x).\ P\ x \wedge r\ x\ y\}^{+}) = ((b,a) \in \{(y,x).\ P\ x \wedge r^{++}\ x\ y\})$
⟨proof⟩

corollary *terminates-impl-no-infinite-trans-computation*:
assumes *terminates*: $\Gamma \vdash c \downarrow s$
shows $\neg(\exists f.\ f\ 0 = (c,s) \wedge (\forall i.\ \Gamma \vdash f\ i \rightarrow^{+} f(Suc\ i)))$
⟨proof⟩

theorem *wf-termi-call-steps*: *wf* (*termi-call-steps* Γ)
⟨proof⟩

lemma *no-infinite-computation-implies-wf*:
assumes *not-inf*: $\neg\ \Gamma \vdash (c,\ s) \rightarrow \ldots(\infty)$
shows *wf* $\{(c2,c1).\ \Gamma \vdash (c,s) \rightarrow^{*} c1 \wedge \Gamma \vdash c1 \rightarrow c2\}$
⟨proof⟩

lemma *not-final-Stuck-step*: $\neg$ *final* (*c*,*Stuck*) $\Longrightarrow \exists c'\ s'.\ \Gamma \vdash (c,\ Stuck) \rightarrow (c',s')$
⟨proof⟩

lemma *not-final-Abrupt-step*:
$\neg$ *final* (*c*,*Abrupt s*) $\Longrightarrow \exists c'\ s'.\ \Gamma \vdash (c,\ Abrupt\ s) \rightarrow (c',s')$
⟨proof⟩

lemma *not-final-Fault-step*:
$\neg$ *final* (*c*,*Fault f*) $\Longrightarrow \exists c'\ s'.\ \Gamma \vdash (c,\ Fault\ f) \rightarrow (c',s')$
⟨proof⟩

lemma *not-final-Normal-step*:
$\neg$ *final* (*c*,*Normal s*) $\Longrightarrow \exists c'\ s'.\ \Gamma \vdash (c,\ Normal\ s) \rightarrow (c',s')$
⟨proof⟩

lemma *final-termi*:
final (*c*,*s*) $\Longrightarrow \Gamma \vdash c \downarrow s$
⟨proof⟩

lemma *split-computation*:
assumes *steps*: $\Gamma \vdash (c,\ s) \rightarrow^{*} (c_f,\ s_f)$
assumes *not-final*: $\neg$ *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*: $\bigwedge c'\ s'.\ \Gamma\vdash\ (c,\ \mathit{Normal}\ s) \rightarrow (c',\ s') \Longrightarrow \Gamma\vdash c' \downarrow s'$
shows $\Gamma\vdash c \downarrow \mathit{Normal}\ s$
⟨*proof*⟩

lemma *wf-implies-termi-reach*:
assumes *wf*: $\mathit{wf}\ \{(\mathit{cfg2},\mathit{cfg1}).\ \Gamma \vdash (c,s) \rightarrow^* \mathit{cfg1} \wedge \Gamma \vdash \mathit{cfg1} \rightarrow \mathit{cfg2}\}$
shows $\bigwedge \mathit{c1}\ \mathit{s1}.\ [\![\Gamma \vdash (c,s) \rightarrow^* \mathit{cfg1};\ \ \mathit{cfg1}{=}(\mathit{c1},\mathit{s1})]\!]\Longrightarrow \Gamma\vdash \mathit{c1}\downarrow \mathit{s1}$
⟨*proof*⟩

theorem *no-infinite-computation-impl-terminates*:
assumes *not-inf*: $\neg\ \Gamma\vdash\ (c,\ s) \rightarrow \ldots(\infty)$
shows $\Gamma\vdash c\downarrow s$
⟨*proof*⟩

corollary *terminates-iff-no-infinite-computation*:
$\Gamma\vdash c\downarrow s = (\neg\ \Gamma\vdash\ (c,\ s) \rightarrow \ldots(\infty))$
⟨*proof*⟩

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*:: $('s,'p,'f)\mathit{com} \Rightarrow ('s,'p,'f)\mathit{com}\ \mathit{set}$
where
$\mathit{redexes}\ \mathit{Skip} = \{\mathit{Skip}\}$ |
$\mathit{redexes}\ (\mathit{Basic}\ f) = \{\mathit{Basic}\ f\}$ |
$\mathit{redexes}\ (\mathit{Spec}\ r) = \{\mathit{Spec}\ r\}$ |
$\mathit{redexes}\ (\mathit{Seq}\ c_1\ c_2) = \{\mathit{Seq}\ c_1\ c_2\} \cup \mathit{redexes}\ c_1$ |
$\mathit{redexes}\ (\mathit{Cond}\ b\ c_1\ c_2) = \{\mathit{Cond}\ b\ c_1\ c_2\}$ |
$\mathit{redexes}\ (\mathit{While}\ b\ c) = \{\mathit{While}\ b\ c\}$ |
$\mathit{redexes}\ (\mathit{Call}\ p) = \{\mathit{Call}\ p\}$ |
$\mathit{redexes}\ (\mathit{DynCom}\ d) = \{\mathit{DynCom}\ d\}$ |
$\mathit{redexes}\ (\mathit{Guard}\ f\ b\ c) = \{\mathit{Guard}\ f\ b\ c\}$ |
$\mathit{redexes}\ (\mathit{Throw}) = \{\mathit{Throw}\}$ |
$\mathit{redexes}\ (\mathit{Catch}\ c_1\ c_2) = \{\mathit{Catch}\ c_1\ c_2\} \cup \mathit{redexes}\ c_1$

lemma *root-in-redexes*: $c \in \mathit{redexes}\ c$
⟨*proof*⟩

lemma *redex-in-redexes*: $\mathit{redex}\ c \in \mathit{redexes}\ c$
⟨*proof*⟩

lemma *redex-redexes*: $\bigwedge c'.\ [\![c' \in \mathit{redexes}\ c;\ \mathit{redex}\ c' = c']\!] \Longrightarrow \mathit{redex}\ c = c'$
⟨*proof*⟩

lemma *step-redexes*:
shows $\bigwedge r\ r'.\ [\![\Gamma\vdash(r,s) \rightarrow (r',s');\ r \in \mathit{redexes}\ c]\!]$

$\Longrightarrow \exists\, c'.\ \Gamma\vdash(c,s) \rightarrow (c',s') \wedge r' \in \mathit{redexes}\ c'$
⟨*proof*⟩

lemma *steps-redexes*:
assumes *steps*: $\Gamma\vdash\ (r,\ s) \rightarrow^*\ (r',\ s')$
shows $\bigwedge c.\ r \in \mathit{redexes}\ c \Longrightarrow \exists\, c'.\ \Gamma\vdash(c,s) \rightarrow^*\ (c',s') \wedge r' \in \mathit{redexes}\ c'$
⟨*proof*⟩

lemma *steps-redexes'*:
assumes *steps*: $\Gamma\vdash\ (r,\ s) \rightarrow^+\ (r',\ s')$
shows $\bigwedge c.\ r \in \mathit{redexes}\ c \Longrightarrow \exists\, c'.\ \Gamma\vdash(c,s) \rightarrow^+\ (c',s') \wedge r' \in \mathit{redexes}\ c'$
⟨*proof*⟩

lemma *step-redexes-Seq*:
assumes *step*: $\Gamma\vdash(r,s) \rightarrow (r',s')$
assumes *Seq*: $\mathit{Seq}\ r\ c_2 \in \mathit{redexes}\ c$
shows $\exists\, c'.\ \Gamma\vdash(c,s) \rightarrow (c',s') \wedge \mathit{Seq}\ r'\ c_2 \in \mathit{redexes}\ c'$
⟨*proof*⟩

lemma *steps-redexes-Seq*:
assumes *steps*: $\Gamma\vdash\ (r,\ s) \rightarrow^*\ (r',\ s')$
shows $\bigwedge c.\ \mathit{Seq}\ r\ c_2 \in \mathit{redexes}\ c \Longrightarrow$
$\exists\, c'.\ \Gamma\vdash(c,s) \rightarrow^*\ (c',s') \wedge \mathit{Seq}\ r'\ c_2 \in \mathit{redexes}\ c'$
⟨*proof*⟩

lemma *steps-redexes-Seq'*:
assumes *steps*: $\Gamma\vdash\ (r,\ s) \rightarrow^+\ (r',\ s')$
shows $\bigwedge c.\ \mathit{Seq}\ r\ c_2 \in \mathit{redexes}\ c$
$\Longrightarrow \exists\, c'.\ \Gamma\vdash(c,s) \rightarrow^+\ (c',s') \wedge \mathit{Seq}\ r'\ c_2 \in \mathit{redexes}\ c'$
⟨*proof*⟩

lemma *step-redexes-Catch*:
assumes *step*: $\Gamma\vdash(r,s) \rightarrow (r',s')$
assumes *Catch*: $\mathit{Catch}\ r\ c_2 \in \mathit{redexes}\ c$
shows $\exists\, c'.\ \Gamma\vdash(c,s) \rightarrow (c',s') \wedge \mathit{Catch}\ r'\ c_2 \in \mathit{redexes}\ c'$
⟨*proof*⟩

lemma *steps-redexes-Catch*:
assumes *steps*: $\Gamma\vdash\ (r,\ s) \rightarrow^*\ (r',\ s')$
shows $\bigwedge c.\ \mathit{Catch}\ r\ c_2 \in \mathit{redexes}\ c \Longrightarrow$
$\exists\, c'.\ \Gamma\vdash(c,s) \rightarrow^*\ (c',s') \wedge \mathit{Catch}\ r'\ c_2 \in \mathit{redexes}\ c'$
⟨*proof*⟩

lemma *steps-redexes-Catch'*:
assumes *steps*: $\Gamma\vdash\ (r,\ s) \rightarrow^+\ (r',\ s')$
shows $\bigwedge c.\ \mathit{Catch}\ r\ c_2 \in \mathit{redexes}\ c$
$\Longrightarrow \exists\, c'.\ \Gamma\vdash(c,s) \rightarrow^+\ (c',s') \wedge \mathit{Catch}\ r'\ c_2 \in \mathit{redexes}\ c'$

⟨*proof*⟩

lemma *redexes-subset*: $\bigwedge c'.\ c' \in redexes\ c \Longrightarrow redexes\ c' \subseteq redexes\ c$
⟨*proof*⟩

lemma *redexes-preserves-termination*:
assumes *termi*: $\Gamma\vdash c\downarrow s$
shows $\bigwedge c'.\ c' \in redexes\ c \Longrightarrow \Gamma\vdash c'\downarrow s$
⟨*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 :: $[('s,'p,'f)\ body,'f\ set,'s\ assn,('s,'p,'f)\ com,'s\ assn,'s\ assn] \Rightarrow bool$
(‹-$\models_{t'/\_}$/ - - -,-› [61,60,1000, 20, 1000,1000] 60)
where
$\Gamma\models_{t/F}\ P\ c\ Q,A \equiv \Gamma\models_{/F}\ P\ c\ Q,A \wedge (\forall s \in Normal\ `\ P.\ \Gamma\vdash c\downarrow s)$

definition
cvalidt::
$[('s,'p,'f)\ body,('s,'p)\ quadruple\ set,'f\ set,$
$'s\ assn,('s,'p,'f)\ com,'s\ assn,'s\ assn] \Rightarrow bool$
(‹-,-$\models_{t'/\_}$/ - - -,-› [61,60, 60,1000, 20, 1000,1000] 60)
where
$\Gamma,\Theta\models_{t/F}\ P\ c\ Q,A \equiv (\forall (P,p,Q,A)\in\Theta.\ \Gamma\models_{t/F}\ P\ (Call\ p)\ Q,A) \longrightarrow \Gamma\ \models_{t/F}\ P\ c\ Q,A$

notation (*ASCII*)
validt (‹-|=t'/-/ - - -,-› [61,60,1000, 20, 1000,1000] 60) **and**
cvalidt (‹-,-|=t'/- / - - -,-› [61,60,60,1000, 20, 1000,1000] 60)

9.2 Properties of Validity

lemma *validtI*:
$[\![\bigwedge s\ t.\ [\![\Gamma\vdash\langle c,Normal\ s\rangle \Rightarrow t;s \in P;t \notin Fault\ `\ F]\!] \Longrightarrow t \in Normal\ `\ Q \cup Abrupt\ `\ A;$
$\bigwedge s.\ s \in P \Longrightarrow \Gamma\vdash\ c\downarrow(Normal\ s)\ ]\!]$
$\Longrightarrow \Gamma\models_{t/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *cvalidtI*:
$\llbracket\bigwedge s\ t.\ \llbracket\forall (P,p,Q,A)\in\Theta.\ \Gamma\models_{t/F}\ P\ (Call\ p)\ Q,A;\Gamma\vdash\langle c,Normal\ s\rangle \Rightarrow t;s \in P;$
$\quad t \notin Fault\ `\ F\rrbracket$
$\quad \Longrightarrow t \in Normal\ `\ Q \cup Abrupt\ `\ A;$
$\bigwedge s.\ \llbracket\forall (P,p,Q,A)\in\Theta.\ \Gamma\models_{t/F}\ P\ (Call\ p)\ Q,A;\ s\in P\rrbracket \Longrightarrow\ \Gamma\vdash c\downarrow(Normal\ s)\rrbracket$
$\Longrightarrow \Gamma,\Theta\models_{t/F}\ P\ c\ Q,A$
$\langle proof\rangle$

lemma *cvalidt-postD*:
$\llbracket\Gamma,\Theta\models_{t/F}\ P\ c\ Q,A;\ \forall (P,p,Q,A)\in\Theta.\ \Gamma\models_{t/F}\ P\ (Call\ p)\ Q,A;\Gamma\vdash\langle c,Normal\ s\ \rangle \Rightarrow$
$t;$
$s \in P;t \notin Fault\ `\ F\rrbracket$
$\Longrightarrow t \in Normal\ `\ Q \cup Abrupt\ `\ A$
$\langle proof\rangle$

lemma *cvalidt-termD*:
$\llbracket\Gamma,\Theta\models_{t/F}\ P\ c\ Q,A;\ \forall (P,p,Q,A)\in\Theta.\ \Gamma\models_{t/F}\ P\ (Call\ p)\ Q,A;s \in P\rrbracket$
$\Longrightarrow \Gamma\vdash c\downarrow(Normal\ s)$
$\langle proof\rangle$

lemma *validt-augment-Faults*:
assumes *valid*:$\Gamma\models_{t/F}\ P\ c\ Q,A$
assumes F': $F \subseteq F'$
shows $\Gamma\models_{t/F'}\ P\ c\ Q,A$
$\langle proof\rangle$

9.3 The Hoare Rules: $\Gamma,\Theta\vdash_{t/F}\ P\ c\ Q,A$

inductive *hoaret*::[('s,'p,'f) body,('s,'p) quadruple set,'f set,
'*s assn*,('s,'p,'f) *com*,'s assn,'s assn]
=> *bool*
(‹(3-,-/⊢$_{t'/-}$ (-/ (-)/ -,-))› [61,60,60,1000,20,1000,1000]60)
for Γ::('s,'p,'f) *body*
where
Skip: $\Gamma,\Theta\vdash_{t/F}\ Q\ Skip\ Q,A$

| *Basic*: $\Gamma,\Theta\vdash_{t/F}\ \{s.\ f\ s \in Q\}\ (Basic\ f)\ Q,A$

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

| *Seq*: $\llbracket\Gamma,\Theta\vdash_{t/F}\ P\ c_1\ R,A;\ \Gamma,\Theta\vdash_{t/F}\ R\ c_2\ Q,A\rrbracket$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F}\ P\ Seq\ c_1\ c_2\ Q,A$

| *Cond*: $\llbracket\Gamma,\Theta\vdash_{t/F}\ (P \cap b)\ c_1\ Q,A;\ \Gamma,\Theta\vdash_{t/F}\ (P \cap -\ b)\ c_2\ Q,A\rrbracket$
$\Longrightarrow$

$\Gamma,\Theta\vdash_{t/F} P\ (Cond\ b\ c_1\ c_2)\ Q,A$

| *While*: $[\![wf\ r;\ \forall\sigma.\ \Gamma,\Theta\vdash_{t/F} (\{\sigma\} \cap P \cap b)\ c\ (\{t.\ (t,\sigma)\in r\} \cap P),A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F} P\ (While\ b\ c)\ (P \cap -\ b),A$

| *Guard*: $\Gamma,\Theta\vdash_{t/F} (g \cap P)\ c\ Q,A$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F} (g \cap P)\ Guard\ f\ g\ c\ Q,A$

| *Guarantee*: $[\![f \in F;\ \Gamma,\Theta\vdash_{t/F} (g \cap P)\ c\ Q,A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F} P\ (Guard\ f\ g\ c)\ Q,A$

| *CallRec*:
$[\![(P,p,Q,A) \in Specs;$
$wf\ r;$
$Specs\text{-}wf = (\lambda p\ \sigma.\ (\lambda(P,q,Q,A).\ (P \cap \{s.\ ((s,q),(\sigma,p)) \in r\},q,Q,A))\ `\ Specs);$
$\forall (P,p,Q,A)\in Specs.$
$p \in dom\ \Gamma \wedge (\forall\sigma.\ \Gamma,\Theta \cup Specs\text{-}wf\ p\ \sigma\vdash_{t/F} (\{\sigma\} \cap P)\ (the\ (\Gamma\ p))\ Q,A)$
$]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F} P\ (Call\ p)\ Q,A$

| *DynCom*: $\forall s \in P.\ \Gamma,\Theta\vdash_{t/F} P\ (c\ s)\ Q,A$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F} P\ (DynCom\ c)\ Q,A$

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

| *Catch*: $[\![\Gamma,\Theta\vdash_{t/F} P\ c_1\ Q,R;\ \Gamma,\Theta\vdash_{t/F} R\ c_2\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F} P\ Catch\ c_1\ c_2\ Q,A$

| *Conseq*: $\forall s \in P.\ \exists P'\ Q'\ A'.\ \Gamma,\Theta\vdash_{t/F} P'\ c\ Q',A' \wedge s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A$
$\Longrightarrow \Gamma,\Theta\vdash_{t/F} P\ c\ Q,A$

| *Asm*: $(P,p,Q,A) \in \Theta$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F} P\ (Call\ p)\ Q,A$

| *ExFalso*: $[\![\Gamma,\Theta\models_{t/F} P\ c\ Q,A;\ \neg\ \Gamma\models_{t/F} P\ c\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F} P\ c\ Q,A$
— This is a hack rule that enables us to derive completeness for an arbitrary context Θ, from completeness for an empty context.

Does not work, because of rule ExFalso, the context Θ 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 $\Gamma,\Theta \vdash_{/F} P\ p\ Q,A$
$\langle proof \rangle$

lemma *hoaret-augment-context*:
  assumes *deriv*: $\Gamma,\Theta \vdash_{t/F} P\ p\ Q,A$
  shows $\bigwedge \Theta'. \Theta \subseteq \Theta' \Longrightarrow \Gamma,\Theta' \vdash_{t/F} P\ p\ Q,A$
$\langle proof \rangle$

9.4 Some Derived Rules

lemma *Conseq'*: $\forall s.\ s \in P \longrightarrow$
$(\exists P'\ Q'\ A'.$
$(\forall\ Z.\ \Gamma,\Theta \vdash_{t/F} (P'\ Z)\ c\ (Q'\ Z),(A'\ Z)) \wedge$
$(\exists Z.\ s \in P'\ Z \wedge (Q'\ Z \subseteq Q) \wedge (A'\ Z \subseteq A)))$
$\Longrightarrow$
$\Gamma,\Theta \vdash_{t/F} P\ c\ Q,A$
$\langle proof \rangle$

lemma *conseq*: $[\![\forall Z.\ \Gamma,\Theta \vdash_{t/F} (P'\ Z)\ c\ (Q'\ Z),(A'\ Z);$
$\forall s.\ s \in P \longrightarrow (\exists\ Z.\ s{\in}P'\ Z \wedge (Q'\ Z \subseteq Q) \wedge (A'\ Z \subseteq A))]\!]$
$\Longrightarrow$
$\Gamma,\Theta \vdash_{t/F} P\ c\ Q,A$
$\langle proof \rangle$

theorem *conseqPrePost*:
$\Gamma,\Theta \vdash_{t/F} P'\ c\ Q',A' \Longrightarrow P \subseteq P' \Longrightarrow Q' \subseteq Q \Longrightarrow A' \subseteq A \Longrightarrow \Gamma,\Theta \vdash_{t/F} P\ c\ Q,A$
$\langle proof \rangle$

lemma *conseqPre*: $\Gamma,\Theta \vdash_{t/F} P'\ c\ Q,A \Longrightarrow P \subseteq P' \Longrightarrow \Gamma,\Theta \vdash_{t/F} P\ c\ Q,A$
$\langle proof \rangle$

lemma *conseqPost*: $\Gamma,\Theta \vdash_{t/F} P\ c\ Q',A' \Longrightarrow Q' \subseteq Q \Longrightarrow A' \subseteq A \Longrightarrow \Gamma,\Theta \vdash_{t/F} P\ c\ Q,A$
$\langle proof \rangle$

lemma *Spec-wf-conv*:
$(\lambda(P,\ q,\ Q,\ A).\ (P \cap \{s.\ ((s,\ q),\ \tau,\ p) \in r\},\ q,\ Q,\ A))$ ‘
$(\bigcup p{\in}Procs.\ \bigcup Z.\ \{(P\ p\ Z,\ p,\ Q\ p\ Z,\ A\ p\ Z)\}) =$
$(\bigcup q{\in}Procs.\ \bigcup Z.\ \{(P\ q\ Z \cap \{s.\ ((s,\ q),\ \tau,\ p) \in r\},\ q,\ Q\ q\ Z,\ A\ q\ Z)\})$
$\langle proof \rangle$

lemma *CallRec'*:
$[\![p{\in}Procs;\ Procs \subseteq dom\ \Gamma;$

$wf\ r$;
$\forall\, p{\in}Procs.\ \forall\,\tau\ Z.$
$\Gamma,\Theta{\cup}(\bigcup\, q{\in}Procs.\ \bigcup\, Z.$
$\{((P\ q\ Z) \cap \{s.\ ((s,q),(\tau,p)) \in r\},q,Q\ q\ Z,(A\ q\ Z))\})$
$\vdash_{t/F}\ (\{\tau\} \cap (P\ p\ Z))\ (the\ (\Gamma\ p))\ (Q\ p\ Z),(A\ p\ Z)]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F}\ (P\ p\ Z)\ (Call\ p)\ (Q\ p\ Z),(A\ p\ Z)$
⟨*proof*⟩

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 $\Gamma,\Theta\models_{t/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *hoaret-sound'*:
$\Gamma,\{\}\vdash_{t/F}\ P\ c\ Q,A \Longrightarrow \Gamma\models_{t/F}\ P\ c\ Q,A$
⟨*proof*⟩

theorem *total-to-partial*:
assumes *total*: $\Gamma,\{\}\vdash_{t/F}\ P\ c\ Q,A$ **shows** $\Gamma,\{\}\vdash_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

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
$\{t.\ \Gamma\vdash\langle c,Normal\ Z\rangle \Rightarrow\ Normal\ t\},\ \{t.\ \Gamma\vdash\langle c,Normal\ Z\rangle \Rightarrow\ Abrupt\ t\}$
⟨*proof*⟩

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*: $\bigwedge s.\ s \in P \Longrightarrow s \in P'\ s \wedge (\forall\, t.\ t \in Q'\ s \longrightarrow t \in Q) \wedge$
$(\forall\, t.\ t \in A'\ s \longrightarrow t \in A)$
shows $\Gamma,\Theta \vdash_{t/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *MGT-implies-complete*:
assumes *MGT*: $\forall Z.\ \Gamma,\{\}\vdash_{t/F} \{s.\ s=Z \wedge \Gamma\vdash\langle c,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault$ ‘ $(-F)) \wedge$
$\Gamma\vdash c\downarrow Normal\ s\}$
c
$\{t.\ \Gamma\vdash\langle c,Normal\ Z\rangle \Rightarrow Normal\ t\},$
$\{t.\ \Gamma\vdash\langle c,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
assumes *valid*: $\Gamma \models_{t/F} P\ c\ Q,A$
shows $\Gamma,\{\} \vdash_{t/F} P\ c\ Q,A$
⟨*proof*⟩

lemma *conseq-extract-state-indep-prop*:
assumes *state-indep-prop*:$\forall s \in P.\ R$
assumes *to-show*: $R \Longrightarrow \Gamma,\Theta\vdash_{t/F} P\ c\ Q,A$
shows $\Gamma,\Theta\vdash_{t/F} P\ c\ Q,A$
⟨*proof*⟩

lemma *MGT-lemma*:
assumes *MGT-Calls*:
$\forall p \in dom\ \Gamma.\ \forall Z.\ \Gamma,\Theta \vdash_{t/F}$
$\{s.\ s=Z \wedge \Gamma\vdash\langle Call\ p,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault$ ‘ $(-F)) \wedge$
$\Gamma\vdash(Call\ p)\downarrow Normal\ s\}$
$(Call\ p)$
$\{t.\ \Gamma\vdash\langle Call\ p,Normal\ Z\rangle \Rightarrow Normal\ t\},$
$\{t.\ \Gamma\vdash\langle Call\ p,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
shows $\bigwedge Z.\ \Gamma,\Theta\vdash_{t/F} \{s.\ s=Z \wedge \Gamma\vdash\langle c,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault$ ‘ $(-F)) \wedge$
$\Gamma\vdash c\downarrow Normal\ s\}$
c
$\{t.\ \Gamma\vdash\langle c,Normal\ Z\rangle \Rightarrow Normal\ t\},\{t.\ \Gamma\vdash\langle c,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
⟨*proof*⟩

lemma *Call-lemma′*:
assumes *Call-hyp*:
$\forall q\in dom\ \Gamma.\ \forall Z.\ \Gamma,\Theta\vdash_{t/F}\{s.\ s=Z \wedge \Gamma\vdash\langle Call\ q,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault$ ‘ $(-F)) \wedge$
$\Gamma\vdash Call\ q\downarrow Normal\ s \wedge ((s,q),(\sigma,p)) \in termi\text{-}call\text{-}steps\ \Gamma\}$
$(Call\ q)$
$\{t.\ \Gamma\vdash\langle Call\ q,Normal\ Z\rangle \Rightarrow Normal\ t\},$
$\{t.\ \Gamma\vdash\langle Call\ q,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
shows $\bigwedge Z.\ \Gamma,\Theta \vdash_{t/F}$
$\{s.\ s=Z \wedge \Gamma\vdash\langle c,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault$ ‘ $(-F)) \wedge \Gamma\vdash Call\ p\downarrow Normal\ \sigma \wedge$
$(\exists c'.\ \Gamma\vdash(Call\ p,Normal\ \sigma) \rightarrow^{+} (c',Normal\ s) \wedge c \in redexes\ c')\}$
c
$\{t.\ \Gamma\vdash\langle c,Normal\ Z\rangle \Rightarrow Normal\ t\},$
$\{t.\ \Gamma\vdash\langle c,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
⟨*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 *A*:
$\forall q \in dom\ \Gamma.\ \forall Z.\ \Gamma,\Theta \vdash_{t/F}$
$\{s.\ s{=}Z \wedge \Gamma\vdash\langle Call\ q,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F)) \wedge$
$\Gamma\vdash Call\ q\downarrow Normal\ s \wedge ((s,q),(\sigma,p)) \in termi\text{-}call\text{-}steps\ \Gamma\}$
$(Call\ q)$
$\{t.\ \Gamma\vdash\langle Call\ q,Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma\vdash\langle Call\ q,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
assumes *pdef*: $p \in dom\ \Gamma$
shows $\bigwedge Z.\ \Gamma,\Theta \vdash_{t/F}$
$(\{\sigma\} \cap \{s.\ s{=}Z \wedge\Gamma\vdash\langle the\ (\Gamma\ p),Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F))$
$\wedge$
$\Gamma\vdash the\ (\Gamma\ p)\downarrow Normal\ s\})$
$the\ (\Gamma\ p)$
$\{t.\ \Gamma\vdash\langle the\ (\Gamma\ p),Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma\vdash\langle the\ (\Gamma\ p),Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
⟨*proof*⟩

lemma *Call-lemma-switch-Call-body*:
assumes
call: $\forall q \in dom\ \Gamma.\ \forall Z.\ \Gamma,\Theta \vdash_{t/F}$
$\{s.\ s{=}Z \wedge \Gamma\vdash\langle Call\ q,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F)) \wedge$
$\Gamma\vdash Call\ q\downarrow Normal\ s \wedge ((s,q),(\sigma,p)) \in termi\text{-}call\text{-}steps\ \Gamma\}$
$(Call\ q)$
$\{t.\ \Gamma\vdash\langle Call\ q,Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma\vdash\langle Call\ q,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
assumes *p-defined*: $p \in dom\ \Gamma$
shows $\bigwedge Z.\ \Gamma,\Theta \vdash_{t/F}$
$(\{\sigma\} \cap \{s.\ s{=}Z \wedge \Gamma\vdash\langle Call\ p,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F)) \wedge$
$\Gamma\vdash Call\ p\downarrow Normal\ s\})$
$the\ (\Gamma\ p)$
$\{t.\ \Gamma\vdash\langle Call\ p,Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma\vdash\langle Call\ p,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
⟨*proof*⟩

lemma *MGT-Call*:
$\forall p \in dom\ \Gamma.\ \forall Z.$
$\Gamma,\Theta \vdash_{t/F} \{s.\ s{=}Z \wedge \Gamma\vdash\langle Call\ p,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F)) \wedge$
$\Gamma\vdash(Call\ p)\downarrow Normal\ s\}$
$(Call\ p)$
$\{t.\ \Gamma\vdash\langle Call\ p,Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma\vdash\langle Call\ p,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
⟨*proof*⟩

lemma *CollInt-iff*: $\{s.\ P\ s\} \cap \{s.\ Q\ s\} = \{s.\ P\ s \wedge Q\ s\}$
⟨*proof*⟩

lemma *image-Un-conv*: f ‘ $(\bigcup p{\in}dom\ \Gamma.\ \bigcup Z.\ \{x\ p\ Z\}) = (\bigcup p{\in}dom\ \Gamma.\ \bigcup Z.\ \{f\ (x\ p\ Z)\})$
⟨*proof*⟩

Another proof of *MGT-Call*, maybe a little more readable

lemma
$\forall p \in dom\ \Gamma.\ \forall Z.$
$\Gamma,\{\} \vdash_{t/F} \{s.\ s{=}Z \wedge \Gamma\vdash\langle Call\ p, Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ \text{‘}\ (-F)) \wedge$
$\Gamma\vdash(Call\ p)\downarrow Normal\ s\}$
$(Call\ p)$
$\{t.\ \Gamma\vdash\langle Call\ p, Normal\ Z\rangle \Rightarrow Normal\ t\},$
$\{t.\ \Gamma\vdash\langle Call\ p, Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
⟨*proof*⟩

theorem *hoaret-complete*: $\Gamma\models_{t/F} P\ c\ Q,A \Longrightarrow \Gamma,\{\}\vdash_{t/F} P\ c\ Q,A$
⟨*proof*⟩

lemma *hoaret-complete'*:
assumes *cvalid*: $\Gamma,\Theta\models_{t/F} P\ c\ Q,A$
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*: $\Gamma,\Theta \models_{t/F} P$ *call-exn init p return' result-exn c* Q,A
assumes *valid-modif*:
$\forall\sigma.\ \Gamma,\Theta \models_{/UNIV} \{\sigma\}\ (Call\ p)\ (Modif\ \sigma),(ModifAbr\ \sigma)$
assumes *res-modif*:
$\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow return'\ s\ t = return\ s\ t$
assumes *ret-modifAbr*:
$\forall s\ t.\ t \in ModifAbr\ (init\ s) \longrightarrow result\text{-}exn\ (return'\ s\ t)\ t = result\text{-}exn\ (return\ s\ t)\ t$
shows $\Gamma,\Theta \models_{t/F} P\ (call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *ProcModifyReturn-sound*:
assumes *valid-call*: $\Gamma,\Theta \models_{t/F} P$ *call init p return' c* Q,A
assumes *valid-modif*:
$\forall\sigma.\ \Gamma,\Theta \models_{/UNIV} \{\sigma\}\ (Call\ p)\ (Modif\ \sigma),(ModifAbr\ \sigma)$
assumes *res-modif*:
$\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow return'\ s\ t = return\ s\ t$

assumes *ret-modifAbr*:
$\forall s\ t.\ t \in ModifAbr\ (init\ s) \longrightarrow return'\ s\ t = return\ s\ t$
shows $\Gamma,\Theta \models_{t/F} P\ (call\ init\ p\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *Proc-exnModifyReturn*:
assumes *spec*: $\Gamma,\Theta\vdash_{t/F} P\ (call\text{-}exn\ init\ p\ return'\ result\text{-}exn\ c)\ Q,A$
assumes *res-modif*:
$\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow (return'\ s\ t) = (return\ s\ t)$
assumes *ret-modifAbr*:
$\forall s\ t.\ t \in ModifAbr\ (init\ s) \longrightarrow (result\text{-}exn\ (return'\ s\ t)\ t) = (result\text{-}exn\ (return\ s\ t)\ t)$
assumes *modifies-spec*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/UNIV} \{\sigma\}\ (Call\ p)\ (Modif\ \sigma),(ModifAbr\ \sigma)$
shows $\Gamma,\Theta\vdash_{t/F} P\ (call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *ProcModifyReturn*:
assumes *spec*: $\Gamma,\Theta\vdash_{t/F} P\ (call\ init\ p\ return'\ c)\ Q,A$
assumes *res-modif*:
$\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow (return'\ s\ t) = (return\ s\ t)$
assumes *ret-modifAbr*:
$\forall s\ t.\ t \in ModifAbr\ (init\ s) \longrightarrow (return'\ s\ t) = (return\ s\ t)$
assumes *modifies-spec*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/UNIV} \{\sigma\}\ (Call\ p)\ (Modif\ \sigma),(ModifAbr\ \sigma)$
shows $\Gamma,\Theta\vdash_{t/F} P\ (call\ init\ p\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *Proc-exnModifyReturnSameFaults-sound*:
assumes *valid-call*: $\Gamma,\Theta \models_{t/F} P\ call\text{-}exn\ init\ p\ return'\ result\text{-}exn\ c\ Q,A$
assumes *valid-modif*:
$\forall \sigma.\ \Gamma,\Theta \models_{/F} \{\sigma\}\ Call\ p\ (Modif\ \sigma),(ModifAbr\ \sigma)$
assumes *res-modif*:
$\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow return'\ s\ t = return\ s\ t$
assumes *ret-modifAbr*:
$\forall s\ t.\ t \in ModifAbr\ (init\ s) \longrightarrow result\text{-}exn\ (return'\ s\ t)\ t = result\text{-}exn\ (return\ s\ t)\ t$
shows $\Gamma,\Theta \models_{t/F} P\ (call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *ProcModifyReturnSameFaults-sound*:
assumes *valid-call*: $\Gamma,\Theta \models_{t/F} P\ call\ init\ p\ return'\ c\ Q,A$
assumes *valid-modif*:
$\forall \sigma.\ \Gamma,\Theta \models_{/F} \{\sigma\}\ Call\ p\ (Modif\ \sigma),(ModifAbr\ \sigma)$
assumes *res-modif*:
$\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow return'\ s\ t = return\ s\ t$

assumes *ret-modifAbr*:
$\forall s\ t.\ t \in \mathit{ModifAbr}\ (\mathit{init}\ s) \longrightarrow \mathit{return}'\ s\ t = \mathit{return}\ s\ t$
shows $\Gamma,\Theta \models_{t/F} P\ (\mathit{call}\ \mathit{init}\ p\ \mathit{return}\ c)\ Q,A$
⟨*proof*⟩

lemma *Proc-exnModifyReturnSameFaults*:
assumes *spec*: $\Gamma,\Theta\vdash_{t/F} P\ (\mathit{call\text{-}exn}\ \mathit{init}\ p\ \mathit{return}'\ \mathit{result\text{-}exn}\ c)\ Q,A$
assumes *res-modif*:
$\forall s\ t.\ t \in \mathit{Modif}\ (\mathit{init}\ s) \longrightarrow (\mathit{return}'\ s\ t) = (\mathit{return}\ s\ t)$
assumes *ret-modifAbr*:
$\forall s\ t.\ t \in \mathit{ModifAbr}\ (\mathit{init}\ s) \longrightarrow \mathit{result\text{-}exn}\ (\mathit{return}'\ s\ t)\ t = \mathit{result\text{-}exn}\ (\mathit{return}\ s\ t)\ t$
assumes *modifies-spec*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ (\mathit{Call}\ p)\ (\mathit{Modif}\ \sigma),(\mathit{ModifAbr}\ \sigma)$
shows $\Gamma,\Theta\vdash_{t/F} P\ (\mathit{call\text{-}exn}\ \mathit{init}\ p\ \mathit{return}\ \mathit{result\text{-}exn}\ c)\ Q,A$
⟨*proof*⟩

lemma *ProcModifyReturnSameFaults*:
assumes *spec*: $\Gamma,\Theta\vdash_{t/F} P\ (\mathit{call}\ \mathit{init}\ p\ \mathit{return}'\ c)\ Q,A$
assumes *res-modif*:
$\forall s\ t.\ t \in \mathit{Modif}\ (\mathit{init}\ s) \longrightarrow (\mathit{return}'\ s\ t) = (\mathit{return}\ s\ t)$
assumes *ret-modifAbr*:
$\forall s\ t.\ t \in \mathit{ModifAbr}\ (\mathit{init}\ s) \longrightarrow (\mathit{return}'\ s\ t) = (\mathit{return}\ s\ t)$
assumes *modifies-spec*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ (\mathit{Call}\ p)\ (\mathit{Modif}\ \sigma),(\mathit{ModifAbr}\ \sigma)$
shows $\Gamma,\Theta\vdash_{t/F} P\ (\mathit{call}\ \mathit{init}\ p\ \mathit{return}\ c)\ Q,A$
⟨*proof*⟩

10.3.2 DynCall

lemma *dynProc-exnModifyReturn-sound*:
assumes *valid-call*: $\Gamma,\Theta \models_{t/F} P\ \mathit{dynCall\text{-}exn}\ f\ g\ \mathit{init}\ p\ \mathit{return}'\ \mathit{result\text{-}exn}\ c\ Q,A$
assumes *valid-modif*:
$\forall s{\in}P.\ \forall \sigma.\ \Gamma,\Theta \models_{/UNIV} \{\sigma\}\ (\mathit{Call}\ (p\ s))\ (\mathit{Modif}\ \sigma),(\mathit{ModifAbr}\ \sigma)$
assumes *ret-modif*:
$\forall s\ t.\ t \in \mathit{Modif}\ (\mathit{init}\ s) \longrightarrow \mathit{return}'\ s\ t = \mathit{return}\ s\ t$
assumes *ret-modifAbr*: $\forall s\ t.\ t \in \mathit{ModifAbr}\ (\mathit{init}\ s) \longrightarrow \mathit{result\text{-}exn}\ (\mathit{return}'\ s\ t)\ t = \mathit{result\text{-}exn}\ (\mathit{return}\ s\ t)\ t$
shows $\Gamma,\Theta \models_{t/F} P\ (\mathit{dynCall\text{-}exn}\ f\ g\ \mathit{init}\ p\ \mathit{return}\ \mathit{result\text{-}exn}\ c)\ Q,A$
⟨*proof*⟩

lemma *dynProcModifyReturn-sound*:
assumes *valid-call*: $\Gamma,\Theta \models_{t/F} P\ \mathit{dynCall}\ \mathit{init}\ p\ \mathit{return}'\ c\ Q,A$
assumes *valid-modif*:
$\forall s{\in}P.\ \forall \sigma.\ \Gamma,\Theta \models_{/UNIV} \{\sigma\}\ (\mathit{Call}\ (p\ s))\ (\mathit{Modif}\ \sigma),(\mathit{ModifAbr}\ \sigma)$
assumes *ret-modif*:
$\forall s\ t.\ t \in \mathit{Modif}\ (\mathit{init}\ s) \longrightarrow \mathit{return}'\ s\ t = \mathit{return}\ s\ t$

assumes *ret-modifAbr*: $\forall s\ t.\ t \in \mathit{ModifAbr}\ (\mathit{init}\ s) \longrightarrow \mathit{return}'\ s\ t = \mathit{return}\ s\ t$
shows $\Gamma,\Theta \models_{t/F} P\ (\mathit{dynCall\ init\ p\ return\ c})\ Q,A$
⟨*proof*⟩

lemma *dynProc-exnModifyReturn*:
assumes *dyn-call*: $\Gamma,\Theta\vdash_{t/F} P\ \mathit{dynCall\text{-}exn}\ f\ g\ \mathit{init}\ p\ \mathit{return}'\ \mathit{result\text{-}exn}\ c\ Q,A$
assumes *ret-modif*:
$\forall s\ t.\ t \in \mathit{Modif}\ (\mathit{init}\ s)$
$\longrightarrow \mathit{return}'\ s\ t = \mathit{return}\ s\ t$
assumes *ret-modifAbr*: $\forall s\ t.\ t \in \mathit{ModifAbr}\ (\mathit{init}\ s)$
$\longrightarrow \mathit{result\text{-}exn}\ (\mathit{return}'\ s\ t)\ t = \mathit{result\text{-}exn}\ (\mathit{return}\ s\ t)\ t$
assumes *modif*:
$\forall s \in P.\ \forall \sigma.$
$\Gamma,\Theta\vdash_{/UNIV} \{\sigma\}\ \mathit{Call}\ (p\ s)\ (\mathit{Modif}\ \sigma),(\mathit{ModifAbr}\ \sigma)$
shows $\Gamma,\Theta \vdash_{t/F} P\ (\mathit{dynCall\text{-}exn}\ f\ g\ \mathit{init}\ p\ \mathit{return}\ \mathit{result\text{-}exn}\ c)\ Q,A$
⟨*proof*⟩

lemma *dynProcModifyReturn*:
assumes *dyn-call*: $\Gamma,\Theta\vdash_{t/F} P\ \mathit{dynCall\ init\ p\ return}'\ c\ Q,A$
assumes *ret-modif*:
$\forall s\ t.\ t \in \mathit{Modif}\ (\mathit{init}\ s)$
$\longrightarrow \mathit{return}'\ s\ t = \mathit{return}\ s\ t$
assumes *ret-modifAbr*: $\forall s\ t.\ t \in \mathit{ModifAbr}\ (\mathit{init}\ s)$
$\longrightarrow \mathit{return}'\ s\ t = \mathit{return}\ s\ t$
assumes *modif*:
$\forall s \in P.\ \forall \sigma.$
$\Gamma,\Theta\vdash_{/UNIV} \{\sigma\}\ \mathit{Call}\ (p\ s)\ (\mathit{Modif}\ \sigma),(\mathit{ModifAbr}\ \sigma)$
shows $\Gamma,\Theta \vdash_{t/F} P\ (\mathit{dynCall\ init\ p\ return\ c})\ Q,A$
⟨*proof*⟩

lemma *dynProc-exnModifyReturnSameFaults-sound*:
assumes *valid-call*: $\Gamma,\Theta \models_{t/F} P\ \mathit{dynCall\text{-}exn}\ f\ g\ \mathit{init}\ p\ \mathit{return}'\ \mathit{result\text{-}exn}\ c\ Q,A$
assumes *valid-modif*:
$\forall s{\in}P.\ \forall \sigma.\ \Gamma,\Theta \models_{/F} \{\sigma\}\ \mathit{Call}\ (p\ s)\ (\mathit{Modif}\ \sigma),(\mathit{ModifAbr}\ \sigma)$
assumes *ret-modif*:
$\forall s\ t.\ t \in \mathit{Modif}\ (\mathit{init}\ s) \longrightarrow \mathit{return}'\ s\ t = \mathit{return}\ s\ t$
assumes *ret-modifAbr*: $\forall s\ t.\ t \in \mathit{ModifAbr}\ (\mathit{init}\ s) \longrightarrow \mathit{result\text{-}exn}\ (\mathit{return}'\ s\ t)\ t = \mathit{result\text{-}exn}\ (\mathit{return}\ s\ t)\ t$
shows $\Gamma,\Theta \models_{t/F} P\ (\mathit{dynCall\text{-}exn}\ f\ g\ \mathit{init}\ p\ \mathit{return}\ \mathit{result\text{-}exn}\ c)\ Q,A$
⟨*proof*⟩

lemma *dynProcModifyReturnSameFaults-sound*:
assumes *valid-call*: $\Gamma,\Theta \models_{t/F} P\ \mathit{dynCall\ init\ p\ return}'\ c\ Q,A$
assumes *valid-modif*:
$\forall s{\in}P.\ \forall \sigma.\ \Gamma,\Theta \models_{/F} \{\sigma\}\ \mathit{Call}\ (p\ s)\ (\mathit{Modif}\ \sigma),(\mathit{ModifAbr}\ \sigma)$
assumes *ret-modif*:

$\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow return'\ s\ t = return\ s\ t$
assumes *ret-modifAbr*: $\forall s\ t.\ t \in ModifAbr\ (init\ s) \longrightarrow return'\ s\ t = return\ s\ t$
shows $\Gamma,\Theta \models_{t/F} P\ (dynCall\ init\ p\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *dynProc-exnModifyReturnSameFaults*:
assumes *dyn-call*: $\Gamma,\Theta\vdash_{t/F} P\ dynCall\text{-}exn\ f\ g\ init\ p\ return'\ result\text{-}exn\ c\ Q,A$
assumes *ret-modif*:
$\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow return'\ s\ t = return\ s\ t$
assumes *ret-modifAbr*: $\forall s\ t.\ t \in ModifAbr\ (init\ s) \longrightarrow result\text{-}exn\ (return'\ s\ t)\ t = result\text{-}exn\ (return\ s\ t)\ t$
assumes *modif*:
$\forall s \in P.\ \forall \sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ Call\ (p\ s)\ (Modif\ \sigma),(ModifAbr\ \sigma)$
shows $\Gamma,\Theta \vdash_{t/F} P\ (dynCall\text{-}exn\ f\ g\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *dynProcModifyReturnSameFaults*:
assumes *dyn-call*: $\Gamma,\Theta\vdash_{t/F} P\ dynCall\ init\ p\ return'\ c\ Q,A$
assumes *ret-modif*:
$\forall s\ t.\ t \in Modif\ (init\ s) \longrightarrow return'\ s\ t = return\ s\ t$
assumes *ret-modifAbr*: $\forall s\ t.\ t \in ModifAbr\ (init\ s) \longrightarrow return'\ s\ t = return\ s\ t$
assumes *modif*:
$\forall s \in P.\ \forall \sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ Call\ (p\ s)\ (Modif\ \sigma),(ModifAbr\ \sigma)$
shows $\Gamma,\Theta \vdash_{t/F} P\ (dynCall\ init\ p\ return\ c)\ Q,A$
⟨*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 $\Gamma,\Theta\models_{t/F} P\ c\ (Q \cap R),(A \cap X)$

⟨*proof*⟩

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 $(Q \cap R),(A \cap X)$
⟨*proof*⟩

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*: $\Gamma,\Theta\models_{/F}$ 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
⟨*proof*⟩

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*: $\Gamma,\Theta\vdash_{/F}$ P c_2 *UNIV*,*UNIV*
  shows $\Gamma,\Theta\vdash_{t/F}$ P c Q,A
⟨*proof*⟩

lemma *CombineStrip-sound*:
  assumes *valid*: $\Gamma,\Theta\models_{t/F}$ P c Q,A
  assumes *valid-strip*: $\Gamma,\Theta\models_{/\{\}}$ P (*strip-guards* $(-F)$ c) *UNIV*,*UNIV*
  shows $\Gamma,\Theta\models_{t/\{\}}$ P c Q,A
⟨*proof*⟩

lemma *CombineStrip*:
  assumes *deriv*: $\Gamma,\Theta\vdash_{t/F}$ P c Q,A
  assumes *deriv-strip*: $\Gamma,\Theta\vdash_{/\{\}}$ P (*strip-guards* $(-F)$ c) *UNIV*,*UNIV*
  shows $\Gamma,\Theta\vdash_{t/\{\}}$ P c Q,A
⟨*proof*⟩

lemma *GuardsFlip-sound*:
  assumes *valid*: $\Gamma,\Theta\models_{t/F}$ P c Q,A
  assumes *validFlip*: $\Gamma,\Theta\models_{/-F}$ P c *UNIV*,*UNIV*
  shows $\Gamma,\Theta\models_{t/\{\}}$ P c Q,A
⟨*proof*⟩

lemma *GuardsFlip*:
  assumes *deriv*: $\Gamma,\Theta\vdash_{t/F}$ *P c Q*,*A*
  assumes *derivFlip*: $\Gamma,\Theta\vdash_{/-F}$ *P c UNIV*,*UNIV*
  shows $\Gamma,\Theta\vdash_{t/\{\}}$ *P c Q*,*A*
⟨*proof*⟩

lemma *MarkGuardsI-sound*:
  assumes *valid*: $\Gamma,\Theta\models_{t/\{\}}$ *P c Q*,*A*
  shows $\Gamma,\Theta\models_{t/\{\}}$ *P mark-guards f c Q*,*A*
⟨*proof*⟩

lemma *MarkGuardsI*:
  assumes *deriv*: $\Gamma,\Theta\vdash_{t/\{\}}$ *P c Q*,*A*
  shows $\Gamma,\Theta\vdash_{t/\{\}}$ *P mark-guards f c Q*,*A*
⟨*proof*⟩

lemma *MarkGuardsD-sound*:
  assumes *valid*: $\Gamma,\Theta\models_{t/\{\}}$ *P mark-guards f c Q*,*A*
  shows $\Gamma,\Theta\models_{t/\{\}}$ *P c Q*,*A*
⟨*proof*⟩

lemma *MarkGuardsD*:
  assumes *deriv*: $\Gamma,\Theta\vdash_{t/\{\}}$ *P mark-guards f c Q*,*A*
  shows $\Gamma,\Theta\vdash_{t/\{\}}$ *P c Q*,*A*
⟨*proof*⟩

lemma *MergeGuardsI-sound*:
  assumes *valid*: $\Gamma,\Theta\models_{t/F}$ *P c Q*,*A*
  shows $\Gamma,\Theta\models_{t/F}$ *P merge-guards c Q*,*A*
⟨*proof*⟩

lemma *MergeGuardsI*:
  assumes *deriv*: $\Gamma,\Theta\vdash_{t/F}$ *P c Q*,*A*
  shows $\Gamma,\Theta\vdash_{t/F}$ *P merge-guards c Q*,*A*
⟨*proof*⟩

lemma *MergeGuardsD-sound*:
  assumes *valid*: $\Gamma,\Theta\models_{t/F}$ *P merge-guards c Q*,*A*
  shows $\Gamma,\Theta\models_{t/F}$ *P c Q*,*A*
⟨*proof*⟩

lemma *MergeGuardsD*:
  assumes *deriv*: $\Gamma,\Theta\vdash_{t/F}$ *P merge-guards c Q*,*A*
  shows $\Gamma,\Theta\vdash_{t/F}$ *P c Q*,*A*
⟨*proof*⟩

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*: $\Gamma,\Theta\models_{t/F}\ P\ (normalize\ c)\ Q,A$
  shows $\Gamma,\Theta\models_{t/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *NormalizeD*:
  assumes *deriv*: $\Gamma,\Theta\vdash_{t/F}\ P\ (normalize\ c)\ Q,A$
  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 $\Gamma,\Theta\models_{t/F}\ P\ (normalize\ c)\ Q,A$
⟨*proof*⟩

lemma *NormalizeI*:
  assumes *deriv*: $\Gamma,\Theta\vdash_{t/F}\ P\ c\ Q,A$
  shows $\Gamma,\Theta\vdash_{t/F}\ P\ (normalize\ c)\ Q,A$
⟨*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*: $\Gamma,\{\}\vdash_{t/F}$ $P\ c\ Q,A$
assumes *F*: $F \subseteq F'$
shows $\Gamma,\{\}\vdash_{t/F'}$ $P\ c\ Q,A$
⟨*proof*⟩

lemma *TerminationPartial-sound*:
assumes *termination*: $\forall s \in P.\ \Gamma\vdash c\downarrow Normal\ s$
assumes *partial-corr*: $\Gamma,\Theta\models_{/F}$ $P\ c\ Q,A$
shows $\Gamma,\Theta\models_{t/F}$ $P\ c\ Q,A$
⟨*proof*⟩

lemma *TerminationPartial*:
assumes *partial-deriv*: $\Gamma,\Theta\vdash_{/F}$ $P\ c\ Q,A$
assumes *termination*: $\forall s \in P.\ \Gamma\vdash c\downarrow Normal\ s$
shows $\Gamma,\Theta\vdash_{t/F}$ $P\ c\ Q,A$
⟨*proof*⟩

lemma *TerminationPartialStrip*:
assumes *partial-deriv*: $\Gamma,\Theta\vdash_{/F}$ $P\ c\ Q,A$
assumes *termination*: $\forall s \in P.\ strip\ F'\ \Gamma\vdash strip\text{-}guards\ F'\ c\downarrow Normal\ s$
shows $\Gamma,\Theta\vdash_{t/F}$ $P\ c\ Q,A$
⟨*proof*⟩

lemma *SplitTotalPartial*:
assumes *termi*: $\Gamma,\Theta\vdash_{t/F}$ $P\ c\ Q',A'$
assumes *part*: $\Gamma,\Theta\vdash_{/F}$ $P\ c\ Q,A$
shows $\Gamma,\Theta\vdash_{t/F}$ $P\ c\ Q,A$
⟨*proof*⟩

lemma *SplitTotalPartial'*:
assumes *termi*: $\Gamma,\Theta\vdash_{t/UNIV}$ $P\ c\ Q',A'$
assumes *part*: $\Gamma,\Theta\vdash_{/F}$ $P\ c\ Q,A$
shows $\Gamma,\Theta\vdash_{t/F}$ $P\ c\ Q,A$
⟨*proof*⟩

end

11 Derived Hoare Rules for Total Correctness

theory *HoareTotal* **imports** *HoareTotalProps* **begin**

lemma *conseq-no-aux*:
$[\![\Gamma,\Theta \vdash_{t/F} P'\ c\ Q',A';$
$\forall s.\ s \in P \longrightarrow (s{\in}P' \wedge (Q' \subseteq Q) \wedge (A' \subseteq A))]\!]$

$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F}\ P\ c\ Q,A$
⟨*proof*⟩

If for example a specification for a "procedure pointer" parameter is in the precondition we can extract it with this rule

lemma *conseq-exploit-pre*:
$[\![\forall s \in P.\ \Gamma,\Theta \vdash_{t/F} (\{s\} \cap P)\ c\ Q,A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *conseq*:$[\![\forall Z.\ \Gamma,\Theta \vdash_{t/F} (P'\ Z)\ c\ (Q'\ Z),(A'\ Z);$
$\forall s.\ s \in P \longrightarrow (\exists\ Z.\ s{\in}P'\ Z \wedge (Q'\ Z \subseteq Q) \wedge (A'\ Z \subseteq A))]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *Lem*:$[\![\forall Z.\ \Gamma,\Theta \vdash_{t/F} (P'\ Z)\ c\ (Q'\ Z),(A'\ Z);$
$P \subseteq \{s.\ \exists\ Z.\ s{\in}P'\ Z \wedge (Q'\ Z \subseteq Q) \wedge (A'\ Z \subseteq A)\}]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F}\ P\ (lem\ x\ c)\ Q,A$
⟨*proof*⟩

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*: $\forall Z.\ \Gamma,\Theta \vdash_{t/F} (P'\ Z)\ c\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{t/F}\ P\ (lem\ x\ c)\ Q,A$
⟨*proof*⟩

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*: $\forall Z.\ \Gamma,\Theta \vdash_{t/F} (P'\ Z)\ c\ (Q'\ Z),\{\}$
shows $\Gamma,\Theta\vdash_{t/F}\ P\ (lem\ x\ c)\ Q,\{\}$
⟨*proof*⟩

lemma *TrivPost*: $\forall Z.\ \Gamma,\Theta \vdash_{t/F} (P'\ Z)\ c\ (Q'\ Z),(A'\ Z)$
$\Longrightarrow$
$\forall Z.\ \Gamma,\Theta \vdash_{t/F} (P'\ Z)\ c\ UNIV,UNIV$
⟨*proof*⟩

lemma *TrivPostNoAbr*: $\forall Z.\ \Gamma,\Theta \vdash_{t/F} (P'\ Z)\ c\ (Q'\ Z),\{\}$
$\Longrightarrow$

$\forall Z.\ \Gamma,\Theta \vdash_{t/F} (P'\ Z)\ c\ UNIV,\{\}$
⟨*proof*⟩

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 $\Gamma,\Theta\vdash_{t/F} P\ DynCom\ c\ Q,A$
⟨*proof*⟩

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*: $\forall Z.\ c\ Z = c\ undefined$
shows $\Gamma,\Theta\vdash_{t/F} P\ (specAnno\ P'\ c\ Q'\ A')\ Q,A$
⟨*proof*⟩

lemma *SpecAnno'*:
$[\![P \subseteq \{s.\ \exists\ Z.\ s{\in}P'\ Z \wedge$
$(\forall t.\ t \in Q'\ Z \longrightarrow\ t \in Q) \wedge (\forall t.\ t \in A'\ Z \longrightarrow t \in\ A)\};$
$\forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ Z)\ (c\ Z)\ (Q'\ Z),(A'\ Z);$
$\forall Z.\ c\ Z = c\ undefined$
$]\!] \Longrightarrow$
$\Gamma,\Theta\vdash_{t/F} P\ (specAnno\ P'\ c\ Q'\ A')\ Q,A$
⟨*proof*⟩

lemma *SpecAnnoNoAbrupt*:
$[\![P \subseteq \{s.\ \exists\ Z.\ s{\in}P'\ Z \wedge$
$(\forall t.\ t \in Q'\ Z \longrightarrow\ t \in Q)\};$
$\forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ Z)\ (c\ Z)\ (Q'\ Z),\{\};$
$\forall Z.\ c\ Z = c\ undefined$
$]\!] \Longrightarrow$
$\Gamma,\Theta\vdash_{t/F} P\ (specAnno\ P'\ c\ Q'\ (\lambda s.\ \{\}))\ Q,A$
⟨*proof*⟩

lemma *Skip*: $P \subseteq Q \Longrightarrow \Gamma,\Theta\vdash_{t/F} P\ Skip\ Q,A$
⟨*proof*⟩

lemma *Basic*: $P \subseteq \{s.\ (f\ s) \in Q\} \Longrightarrow\ \Gamma,\Theta\vdash_{t/F} P\ (Basic\ f)\ Q,A$
⟨*proof*⟩

lemma *BasicCond*:
$[\![P \subseteq \{s.\ (b\ s \longrightarrow f\ s{\in}Q) \wedge (\neg\ b\ s \longrightarrow g\ s{\in}Q)\}]\!] \Longrightarrow$
$\Gamma,\Theta\vdash_{t/F} P\ Basic\ (\lambda s.\ if\ b\ s\ then\ f\ s\ else\ g\ s)\ Q,A$
⟨*proof*⟩

lemma *Spec*: $P \subseteq \{s. (\forall t. (s,t) \in r \longrightarrow t \in Q) \wedge (\exists t. (s,t) \in r)\}$
$\Longrightarrow \Gamma,\Theta\vdash_{t/F} P\ (Spec\ r)\ Q,A$
⟨*proof*⟩

lemma *SpecIf*:
$[\![P \subseteq \{s. (b\ s \longrightarrow f\ s \in Q) \wedge (\neg\ b\ s \longrightarrow g\ s \in Q \wedge h\ s \in Q)\}]\!] \Longrightarrow$
$\Gamma,\Theta\vdash_{t/F} P\ Spec\ (if\text{-}rel\ b\ f\ g\ h)\ Q,A$
⟨*proof*⟩

lemma *Seq* [*trans*, *intro?*]:
$[\![\Gamma,\Theta\vdash_{t/F} P\ c_1\ R,A; \Gamma,\Theta\vdash_{t/F} R\ c_2\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F} P\ Seq\ c_1\ c_2\ Q,A$
⟨*proof*⟩

lemma *SeqSwap*:
$[\![\Gamma,\Theta\vdash_{t/F} R\ c2\ Q,A; \Gamma,\Theta\vdash_{t/F} P\ c1\ R,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F} P\ Seq\ c1\ c2\ Q,A$
⟨*proof*⟩

lemma *BSeq*:
$[\![\Gamma,\Theta\vdash_{t/F} P\ c_1\ R,A; \Gamma,\Theta\vdash_{t/F} R\ c_2\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F} P\ (bseq\ c_1\ c_2)\ Q,A$
⟨*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 $\Gamma,\Theta\vdash_{t/F} P\ (Cond\ b\ c_1\ c_2)\ Q,A$
⟨*proof*⟩

lemma *CondSwap*:
$[\![\Gamma,\Theta\vdash_{t/F} P1\ c1\ Q,A; \Gamma,\Theta\vdash_{t/F} P2\ c2\ Q,A;$
$P \subseteq \{s. (s\in b \longrightarrow s\in P1) \wedge (s\notin b \longrightarrow s\in P2)\}]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F} P\ (Cond\ b\ c1\ c2)\ Q,A$
⟨*proof*⟩

lemma *Cond'*:
$[\![P \subseteq \{s. (b \subseteq P1) \wedge (-\ b \subseteq P2)\};\Gamma,\Theta\vdash_{t/F} P1\ c1\ Q,A; \Gamma,\Theta\vdash_{t/F} P2\ c2\ Q,A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F} P\ (Cond\ b\ c1\ c2)\ Q,A$
⟨*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 $(Cond\ b\ c_1\ c_2)$ 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*: $\Gamma,\Theta\vdash_{t/F}$ 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 $(Cond\ b\ c_1\ c_2)$ Q,A
⟨*proof*⟩

lemma *switchNil*:
$P \subseteq Q \Longrightarrow \Gamma,\Theta\vdash_{t/F}$ P $(switch\ v\ [])$ Q,A
⟨*proof*⟩

lemma *switchCons*:
$[\![P \subseteq \{s.\ (v\ s \in V \longrightarrow s \in P_1) \wedge (v\ s \notin V \longrightarrow s \in P_2)\};$
$\Gamma,\Theta\vdash_{t/F}$ P_1 c $Q,A;$
$\Gamma,\Theta\vdash_{t/F}$ P_2 $(switch\ v\ vs)$ $Q,A]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{t/F}$ P $(switch\ v\ ((V,c)\#vs))$ Q,A
⟨*proof*⟩

lemma *Guard*:
$[\![P \subseteq g \cap R;\ \Gamma,\Theta\vdash_{t/F}$ R c $Q,A]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{t/F}$ P *Guard f g c* Q,A
⟨*proof*⟩

lemma *GuardSwap*:
$[\![\ \Gamma,\Theta\vdash_{t/F}$ R c $Q,A;\ P \subseteq g \cap R]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{t/F}$ P *Guard f g c* Q,A
⟨*proof*⟩

lemma *Guarantee*:
$[\![P \subseteq \{s.\ s \in g \longrightarrow s \in R\};\ \Gamma,\Theta\vdash_{t/F}$ R c $Q,A;\ f \in F]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{t/F}$ P $(Guard\ f\ g\ c)$ Q,A
⟨*proof*⟩

lemma *GuaranteeSwap*:
$[\![\ \Gamma,\Theta\vdash_{t/F}$ R c $Q,A;\ P \subseteq \{s.\ s \in g \longrightarrow s \in R\};\ f \in F]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{t/F}$ P $(Guard\ f\ g\ c)$ Q,A
⟨*proof*⟩

lemma *GuardStrip*:

$[\![P \subseteq R;\ \Gamma,\Theta \vdash_{t/F}\ R\ c\ Q,A;\ f \in F]\!]$
$\Longrightarrow \Gamma,\Theta \vdash_{t/F}\ P\ (\mathit{Guard}\ f\ g\ c)\ Q,A$
$\langle \mathit{proof} \rangle$

lemma *GuardStripSwap*:
$[\![\Gamma,\Theta \vdash_{t/F}\ R\ c\ Q,A;\ P \subseteq R;\ f \in F]\!]$
$\Longrightarrow \Gamma,\Theta \vdash_{t/F}\ P\ (\mathit{Guard}\ f\ g\ c)\ Q,A$
$\langle \mathit{proof} \rangle$

lemma *GuaranteeStrip*:
$[\![P \subseteq R;\ \Gamma,\Theta \vdash_{t/F}\ R\ c\ Q,A;\ f \in F]\!]$
$\Longrightarrow \Gamma,\Theta \vdash_{t/F}\ P\ (\mathit{guaranteeStrip}\ f\ g\ c)\ Q,A$
$\langle \mathit{proof} \rangle$

lemma *GuaranteeStripSwap*:
$[\![\Gamma,\Theta \vdash_{t/F}\ R\ c\ Q,A;\ P \subseteq R;\ f \in F]\!]$
$\Longrightarrow \Gamma,\Theta \vdash_{t/F}\ P\ (\mathit{guaranteeStrip}\ f\ g\ c)\ Q,A$
$\langle \mathit{proof} \rangle$

lemma *GuaranteeAsGuard*:
$[\![P \subseteq g \cap R;\ \Gamma,\Theta \vdash_{t/F}\ R\ c\ Q,A]\!]$
$\Longrightarrow \Gamma,\Theta \vdash_{t/F}\ P\ \mathit{guaranteeStrip}\ f\ g\ c\ Q,A$
$\langle \mathit{proof} \rangle$

lemma *GuaranteeAsGuardSwap*:
$[\![\ \Gamma,\Theta \vdash_{t/F}\ R\ c\ Q,A;\ P \subseteq g \cap R]\!]$
$\Longrightarrow \Gamma,\Theta \vdash_{t/F}\ P\ \mathit{guaranteeStrip}\ f\ g\ c\ Q,A$
$\langle \mathit{proof} \rangle$

lemma *GuardsNil*:
$\Gamma,\Theta \vdash_{t/F}\ P\ c\ Q,A \Longrightarrow$
$\Gamma,\Theta \vdash_{t/F}\ P\ (\mathit{guards}\ []\ c)\ Q,A$
$\langle \mathit{proof} \rangle$

lemma *GuardsCons*:
$\Gamma,\Theta \vdash_{t/F}\ P\ \mathit{Guard}\ f\ g\ (\mathit{guards}\ \mathit{gs}\ c)\ Q,A \Longrightarrow$
$\Gamma,\Theta \vdash_{t/F}\ P\ (\mathit{guards}\ ((f,g)\#\mathit{gs})\ c)\ Q,A$
$\langle \mathit{proof} \rangle$

lemma *GuardsConsGuaranteeStrip*:
$\Gamma,\Theta \vdash_{t/F}\ P\ \mathit{guaranteeStrip}\ f\ g\ (\mathit{guards}\ \mathit{gs}\ c)\ Q,A \Longrightarrow$
$\Gamma,\Theta \vdash_{t/F}\ P\ (\mathit{guards}\ (\mathit{guaranteeStripPair}\ f\ g\#\mathit{gs})\ c)\ Q,A$
$\langle \mathit{proof} \rangle$

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 (*whileAnno* b I V c) Q,A
⟨*proof*⟩

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*:
$\Gamma,\Theta\vdash_{/F}$ ($I \cap b$) c I,A
assumes *I-Q*: $I \cap -b \subseteq Q$
assumes *wf*: *wf* V
shows $\Gamma,\Theta\vdash_{t/F}$ P (*whileAnno* b I V c) Q,A
⟨*proof*⟩

lemma $\Gamma,\Theta\vdash_{/F}$ ($P \cap b$) c Q,A $\Longrightarrow$ $\Gamma,\Theta\vdash_{/F}$ ($P \cap b$) (*Seq* c (*Guard* f Q *Skip*)) Q,A
⟨*proof*⟩

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 (*guards gs*
(*whileAnno* b J V (*Seq* c (*guards gs Skip*)))) Q,A
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F}$ P (*whileAnnoG gs* b I V c) Q,A
⟨*proof*⟩

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 (*whileAnno* b I V (*Seq* c *Skip*)) Q,A
⟨*proof*⟩

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*: $\forall Z.\ c\ Z = c$ *undefined*
assumes *wf*: $\forall Z.$ *wf* (V Z)

shows $\Gamma,\Theta\vdash_{t/F}$ P (*whileAnnoFix b I V c*) Q,A
⟨*proof*⟩

lemma *WhileAnnoFix′*:
assumes *consequence*: $P \subseteq \{s.\ (\exists\ Z.\ s{\in}I\ Z \wedge$
$(\forall t.\ t \in I\ Z \cap -b \longrightarrow t \in Q))\ \}$
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*: $\forall Z.\ c\ Z = c\ undefined$
assumes *wf*: $\forall Z.\ wf\ (V\ Z)$
shows $\Gamma,\Theta\vdash_{t/F}$ P (*whileAnnoFix b I V c*) Q,A
⟨*proof*⟩

lemma *WhileAnnoGFix*:
assumes *whileAnnoFix*:
$\Gamma,\Theta\vdash_{t/F}$ P (*guards gs*
(*whileAnnoFix b J V* (λZ. (*Seq* (*c Z*) (*guards gs Skip*))))) Q,A
shows $\Gamma,\Theta\vdash_{t/F}$ P (*whileAnnoGFix gs b I V c*) Q,A
⟨*proof*⟩

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
⟨*proof*⟩

lemma *Block-exn*:
assumes *adapt*: $P \subseteq \{s.\ init\ s \in P'\ s\}$
assumes *bdy*: $\forall s.\ \Gamma,\Theta\vdash_{t/F} (P'\ s)\ bdy\ \{t.\ return\ s\ t \in R\ s\ t\},\{t.\ result\text{-}exn\ (return\ s\ t)\ t \in A\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
shows $\Gamma,\Theta\vdash_{t/F}$ P (*block-exn init bdy return result-exn c*) Q,A
⟨*proof*⟩

lemma *Block*:
assumes *adapt*: $P \subseteq \{s.\ init\ s \in P'\ s\}$
assumes *bdy*: $\forall s.\ \Gamma,\Theta\vdash_{t/F} (P'\ s)\ bdy\ \{t.\ return\ s\ t \in R\ s\ t\},\{t.\ return\ s\ t \in A\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
shows $\Gamma,\Theta\vdash_{t/F}$ P (*block init bdy return c*) Q,A
⟨*proof*⟩

lemma *BlockSwap*:
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *bdy*: $\forall s.\ \Gamma,\Theta\vdash_{t/F} (P'\ s)\ bdy\ \{t.\ return\ s\ t \in R\ s\ t\},\{t.\ return\ s\ t \in A\}$
assumes *adapt*: $P \subseteq \{s.\ init\ s \in P'\ s\}$
shows $\Gamma,\Theta\vdash_{t/F}$ P (*block init bdy return c*) Q,A
⟨*proof*⟩

lemma *Block-exnSwap*:
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F}\ (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *bdy*: $\forall s.\ \Gamma,\Theta\vdash_{t/F}\ (P'\ s)\ bdy\ \{t.\ return\ s\ t \in R\ s\ t\},\{t.\ result\text{-}exn\ (return\ s\ t)\ t \in A\}$
assumes *adapt*: $P \subseteq \{s.\ init\ s \in P'\ s\}$
shows $\Gamma,\Theta\vdash_{t/F}\ P\ (block\text{-}exn\ init\ bdy\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *Block-exnSpec*:
assumes *adapt*: $P \subseteq \{s.\ \exists Z.\ init\ s \in P'\ Z \land (\forall t.\ t \in Q'\ Z \longrightarrow return\ s\ t \in R\ s\ t) \land (\forall t.\ t \in A'\ Z \longrightarrow result\text{-}exn\ (return\ s\ t)\ t \in A)\}$
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 $\Gamma,\Theta\vdash_{t/F}\ P\ (block\text{-}exn\ init\ bdy\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *BlockSpec*:
assumes *adapt*: $P \subseteq \{s.\ \exists Z.\ init\ s \in P'\ Z \land (\forall t.\ t \in Q'\ Z \longrightarrow return\ s\ t \in R\ s\ t) \land (\forall t.\ t \in A'\ Z \longrightarrow return\ s\ t \in A)\}$
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 $\Gamma,\Theta\vdash_{t/F}\ P\ (block\ init\ bdy\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *Throw*: $P \subseteq A \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ Throw\ Q,A$
⟨*proof*⟩

lemmas *Catch* = *hoaret.Catch*
lemma *CatchSwap*: $[\![\Gamma,\Theta\vdash_{t/F}\ R\ c_2\ Q,A;\ \Gamma,\Theta\vdash_{t/F}\ P\ c_1\ Q,R]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ Catch\ c_1\ c_2\ Q,A$
⟨*proof*⟩

lemma *raise*: $P \subseteq \{s.\ f\ s \in A\} \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ raise\ f\ Q,A$
⟨*proof*⟩

lemma *condCatch*: $[\![\Gamma,\Theta\vdash_{t/F}\ P\ c_1\ Q,((b \cap R) \cup (-b \cap A));\Gamma,\Theta\vdash_{t/F}\ R\ c_2\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ condCatch\ c_1\ b\ c_2\ Q,A$
⟨*proof*⟩

lemma *condCatchSwap*: $[\![\Gamma,\Theta\vdash_{t/F}\ R\ c_2\ Q,A;\ \Gamma,\Theta\vdash_{t/F}\ P\ c_1\ Q,((b \cap R) \cup (-b \cap A))]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ condCatch\ c_1\ b\ c_2\ Q,A$
⟨*proof*⟩

lemma *Proc-exnSpec*:
assumes *adapt*: $P \subseteq \{s. \exists Z. \ init\ s \in P'\ Z \land$
$(\forall t.\ t \in Q'\ Z \longrightarrow return\ s\ t \in R\ s\ t) \land$
$(\forall t.\ t \in A'\ Z \longrightarrow result\text{-}exn\ (return\ s\ t)\ t \in A)\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *p*: $\forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ Z)\ Call\ p\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{t/F} P\ (call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *ProcSpec*:
assumes *adapt*: $P \subseteq \{s. \exists Z. \ init\ s \in P'\ Z \land$
$(\forall t.\ t \in Q'\ Z \longrightarrow return\ s\ t \in R\ s\ t) \land$
$(\forall t.\ t \in A'\ Z \longrightarrow return\ s\ t \in A)\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *p*: $\forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ Z)\ Call\ p\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{t/F} P\ (call\ init\ p\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *Proc-exnSpec'*:
assumes *adapt*: $P \subseteq \{s. \exists Z. \ init\ s \in P'\ Z \land$
$(\forall t \in Q'\ Z.\ return\ s\ t \in R\ s\ t) \land$
$(\forall t \in A'\ Z.\ result\text{-}exn\ (return\ s\ t)\ t \in A)\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *p*: $\forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ Z)\ Call\ p\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{t/F} P\ (call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *ProcSpec'*:
assumes *adapt*: $P \subseteq \{s. \exists Z. \ init\ s \in P'\ Z \land$
$(\forall t \in Q'\ Z.\ return\ s\ t \in R\ s\ t) \land$
$(\forall t \in A'\ Z.\ return\ s\ t \in A)\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *p*: $\forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ Z)\ Call\ p\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{t/F} P\ (call\ init\ p\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *Proc-exnSpecNoAbrupt*:
assumes *adapt*: $P \subseteq \{s. \exists Z. \ init\ s \in P'\ Z \land$
$(\forall t.\ t \in Q'\ Z \longrightarrow return\ s\ t \in R\ s\ t)\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *p*: $\forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ Z)\ Call\ p\ (Q'\ Z),\{\}$
shows $\Gamma,\Theta\vdash_{t/F} P\ (call\text{-}exn\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *ProcSpecNoAbrupt*:
assumes *adapt*: $P \subseteq \{s. \exists Z. \ init\ s \in P'\ Z \land$

$(\forall t.\ t \in Q'\ Z \longrightarrow \mathit{return}\ s\ t \in R\ s\ t)\}$
assumes c: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F}\ (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes p: $\forall Z.\ \Gamma,\Theta\vdash_{t/F}\ (P'\ Z)\ \mathit{Call}\ p\ (Q'\ Z),\{\}$
shows $\Gamma,\Theta\vdash_{t/F}\ P\ (\mathit{call\ init}\ p\ \mathit{return}\ c)\ Q,A$
$\langle \mathit{proof} \rangle$

lemma *FCall*:
$\Gamma,\Theta\vdash_{t/F}\ P\ (\mathit{call\ init}\ p\ \mathit{return}\ (\lambda s\ t.\ c\ (\mathit{result}\ t)))\ Q,A$
$\Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ (\mathit{fcall\ init}\ p\ \mathit{return\ result}\ c)\ Q,A$
$\langle \mathit{proof} \rangle$

lemma *ProcRec*:
assumes *deriv-bodies*:
$\forall p\in \mathit{Procs}.$
$\forall \sigma\ Z.\ \Gamma,\Theta\cup(\bigcup q\in \mathit{Procs}.\ \bigcup Z.$
$\{(P\ q\ Z \cap \{s.\ ((s,q),\ \sigma,p) \in r\},q,Q\ q\ Z,A\ q\ Z)\})$
$\vdash_{t/F}\ (\{\sigma\} \cap P\ p\ Z)\ (\mathit{the}\ (\Gamma\ p))\ (Q\ p\ Z),(A\ p\ Z)$
assumes *wf*: *wf r*
assumes *Procs-defined*: $\mathit{Procs} \subseteq \mathit{dom}\ \Gamma$
shows $\forall p\in \mathit{Procs}.\ \forall Z.$
$\Gamma,\Theta\vdash_{t/F}(P\ p\ Z)\ \mathit{Call}\ p\ (Q\ p\ Z),(A\ p\ Z)$
$\langle \mathit{proof} \rangle$

lemma *ProcRec'*:
assumes *ctxt*:
$\Theta'=(\lambda\sigma\ p.\ \Theta\cup(\bigcup q\in \mathit{Procs}.$
$\bigcup Z.\ \{(P\ q\ Z \cap \{s.\ ((s,q),\ \sigma,p) \in r\},q,Q\ q\ Z,A\ q\ Z)\}))$
assumes *deriv-bodies*:
$\forall p\in \mathit{Procs}.$
$\forall \sigma\ Z.\ \Gamma,\Theta'\ \sigma\ p\vdash_{t/F}\ (\{\sigma\} \cap P\ p\ Z)\ (\mathit{the}\ (\Gamma\ p))\ (Q\ p\ Z),(A\ p\ Z)$
assumes *wf*: *wf r*
assumes *Procs-defined*: $\mathit{Procs} \subseteq \mathit{dom}\ \Gamma$
shows $\forall p\in \mathit{Procs}.\ \forall Z.\ \ \Gamma,\Theta\vdash_{t/F}(P\ p\ Z)\ \mathit{Call}\ p\ (Q\ p\ Z),(A\ p\ Z)$
$\langle \mathit{proof} \rangle$

lemma *ProcRecList*:
assumes *deriv-bodies*:
$\forall p\in \mathit{set\ Procs}.$
$\forall \sigma\ Z.\ \Gamma,\Theta\cup(\bigcup q\in \mathit{set\ Procs}.\ \bigcup Z.$
$\{(P\ q\ Z \cap \{s.\ ((s,q),\ \sigma,p) \in r\},q,Q\ q\ Z,A\ q\ Z)\})$
$\vdash_{t/F}\ (\{\sigma\} \cap P\ p\ Z)\ (\mathit{the}\ (\Gamma\ p))\ (Q\ p\ Z),(A\ p\ Z)$
assumes *wf*: *wf r*
assumes *dist*: *distinct Procs*
assumes *Procs-defined*: $\mathit{set\ Procs} \subseteq \mathit{dom}\ \Gamma$
shows $\forall p\in \mathit{set\ Procs}.\ \forall Z.$
$\Gamma,\Theta\vdash_{t/F}(P\ p\ Z)\ \mathit{Call}\ p\ (Q\ p\ Z),(A\ p\ Z)$
$\langle \mathit{proof} \rangle$

lemma *ProcRecSpecs*:
$[\![\forall \sigma.\ \forall (P,p,Q,A) \in Specs.$
$\Gamma,\Theta \cup ((\lambda(P,q,Q,A).\ (P \cap \{s.\ ((s,q),(\sigma,p)) \in r\},q,Q,A))\ `\ Specs)$
$\vdash_{t/F} (\{\sigma\} \cap P)\ (the\ (\Gamma\ p))\ Q,A;$
$wf\ r;$
$\forall (P,p,Q,A) \in Specs.\ p \in dom\ \Gamma]\!]$
$\Longrightarrow \forall (P,p,Q,A) \in Specs.\ \Gamma,\Theta\vdash_{t/F} P\ (Call\ p)\ Q,A$
⟨*proof*⟩

lemma *ProcRec1*:
assumes *deriv-body*:
$\forall \sigma\ Z.\ \Gamma,\Theta\cup(\bigcup Z.\ \{(P\ Z \cap \{s.\ ((s,p),\ \sigma,p) \in r\},p,Q\ Z,A\ Z)\})$
$\vdash_{t/F} (\{\sigma\} \cap P\ Z)\ (the\ (\Gamma\ p))\ (Q\ Z),(A\ Z)$
assumes *wf*: $wf\ r$
assumes *p-defined*: $p \in dom\ \Gamma$
shows $\forall Z.\ \Gamma,\Theta\vdash_{t/F} (P\ Z)\ Call\ p\ (Q\ Z),(A\ Z)$
⟨*proof*⟩

lemma *ProcNoRec1*:
assumes *deriv-body*:
$\forall Z.\ \Gamma,\Theta\vdash_{t/F} (P\ Z)\ (the\ (\Gamma\ p))\ (Q\ Z),(A\ Z)$
assumes *p-defined*: $p \in dom\ \Gamma$
shows $\forall Z.\ \Gamma,\Theta\vdash_{t/F} (P\ Z)\ Call\ p\ (Q\ Z),(A\ Z)$
⟨*proof*⟩

lemma *ProcBody*:
assumes *WP*: $P \subseteq P'$
assumes *deriv-body*: $\Gamma,\Theta\vdash_{t/F} P'\ body\ Q,A$
assumes *body*: $\Gamma\ p = Some\ body$
shows $\Gamma,\Theta\vdash_{t/F} P\ Call\ p\ Q,A$
⟨*proof*⟩

lemma *CallBody*:
assumes *adapt*: $P \subseteq \{s.\ init\ s \in P'\ s\}$
assumes *bdy*: $\forall s.\ \Gamma,\Theta\vdash_{t/F} (P'\ s)\ body\ \{t.\ return\ s\ t \in R\ s\ t\},\{t.\ return\ s\ t \in A\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *body*: $\Gamma\ p = Some\ body$
shows $\Gamma,\Theta\vdash_{t/F} P\ (call\ init\ p\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *Call-exnBody*:
assumes *adapt*: $P \subseteq \{s.\ init\ s \in P'\ s\}$
assumes *bdy*: $\forall s.\ \Gamma,\Theta\vdash_{t/F} (P'\ s)\ body\ \{t.\ return\ s\ t \in R\ s\ t\},\{t.\ result\text{-}exn\ (return\ s\ t)\ t \in A\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *body*: $\Gamma\ p = Some\ body$

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*:
  $\forall s\ t.\ t \in$ *Modif* (*init s*) $\longrightarrow$ (*return′ s t*) = (*return s t*)
 assumes *modifies-spec*:
 $\forall\sigma.\ \Gamma,\Theta\vdash_{/UNIV}$ $\{\sigma\}$ *Call p* (*Modif* σ),{}
 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*:
  $\forall s\ t.\ t \in$ *Modif* (*init s*) $\longrightarrow$ (*return′ s t*) = (*return s t*)
 assumes *modifies-spec*:
 $\forall\sigma.\ \Gamma,\Theta\vdash_{/UNIV}$ $\{\sigma\}$ *Call p* (*Modif* σ),{}
 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*:
  $\forall s\ t.\ t \in$ *Modif* (*init s*) $\longrightarrow$ (*return′ s t*) = (*return s t*)
 assumes *modifies-spec*:
 $\forall\sigma.\ \Gamma,\Theta\vdash_{/F}$ $\{\sigma\}$ *Call p* (*Modif* σ),{}
 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*:
  $\forall s\ t.\ t \in$ *Modif* (*init s*) $\longrightarrow$ (*return′ s t*) = (*return s t*)
 assumes *modifies-spec*:
 $\forall\sigma.\ \Gamma,\Theta\vdash_{/F}$ $\{\sigma\}$ *Call p* (*Modif* σ),{}
 shows $\Gamma,\Theta\vdash_{t/F}$ *P* (*call-exn init p return result-exn c*) *Q*,*A*
 ⟨*proof*⟩

lemma *DynProc-exn*:

assumes *adapt*: $P \subseteq \{s. \exists Z. \ init\ s \in P'\ s\ Z \wedge$
$(\forall t.\ t \in Q'\ s\ Z \longrightarrow\ return\ s\ t \in R\ s\ t) \wedge$
$(\forall t.\ t \in A'\ s\ Z \longrightarrow result\text{-}exn\ (return\ s\ t)\ t \in A)\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *p*: $\forall s\in P.\ \forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ s\ Z)\ Call\ (p\ s)\ (Q'\ s\ Z),(A'\ s\ Z)$
shows $\Gamma,\Theta\vdash_{t/F} P\ dynCall\text{-}exn\ f\ UNIV\ init\ p\ return\ result\text{-}exn\ c\ Q,A$
⟨*proof*⟩

lemma *DynProc-exn-guards-cons*:
assumes *p*: $\Gamma,\Theta\vdash_{t/F} P\ dynCall\text{-}exn\ f\ UNIV\ init\ p\ return\ result\text{-}exn\ c\ Q,A$
shows $\Gamma,\Theta\vdash_{t/F} (g \cap P)\ dynCall\text{-}exn\ f\ g\ init\ p\ return\ result\text{-}exn\ c\ Q,A$
⟨*proof*⟩

lemma *DynProc*:
assumes *adapt*: $P \subseteq \{s. \exists Z. \ init\ s \in P'\ s\ Z \wedge$
$(\forall t.\ t \in Q'\ s\ Z \longrightarrow\ return\ s\ t \in R\ s\ t) \wedge$
$(\forall t.\ t \in A'\ s\ Z \longrightarrow return\ s\ t \in A)\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *p*: $\forall s\in P.\ \forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ s\ Z)\ Call\ (p\ s)\ (Q'\ s\ Z),(A'\ s\ Z)$
shows $\Gamma,\Theta\vdash_{t/F} P\ dynCall\ init\ p\ return\ c\ Q,A$
⟨*proof*⟩

lemma *DynProc-exn′*:
assumes *adapt*: $P \subseteq \{s. \exists Z. \ init\ s \in P'\ s\ Z \wedge$
$(\forall t \in Q'\ s\ Z.\ return\ s\ t \in R\ s\ t) \wedge$
$(\forall t \in A'\ s\ Z.\ result\text{-}exn\ (return\ s\ t)\ t \in A)\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *p*: $\forall s\in P.\ \forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ s\ Z)\ Call\ (p\ s)\ (Q'\ s\ Z),(A'\ s\ Z)$
shows $\Gamma,\Theta\vdash_{t/F} P\ dynCall\text{-}exn\ f\ UNIV\ init\ p\ return\ result\text{-}exn\ c\ Q,A$
⟨*proof*⟩

lemma *DynProc′*:
assumes *adapt*: $P \subseteq \{s. \exists Z. \ init\ s \in P'\ s\ Z \wedge$
$(\forall t \in Q'\ s\ Z.\ return\ s\ t \in R\ s\ t) \wedge$
$(\forall t \in A'\ s\ Z.\ return\ s\ t \in A)\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *p*: $\forall s\in P.\ \forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ s\ Z)\ Call\ (p\ s)\ (Q'\ s\ Z),(A'\ s\ Z)$
shows $\Gamma,\Theta\vdash_{t/F} P\ dynCall\ init\ p\ return\ c\ Q,A$
⟨*proof*⟩

lemma *DynProc-exnStaticSpec*:
assumes *adapt*: $P \subseteq \{s.\ s \in S \wedge (\exists Z.\ init\ s \in P'\ Z\ \wedge$
$(\forall \tau.\ \tau \in Q'\ Z \longrightarrow return\ s\ \tau \in R\ s\ \tau) \wedge$
$(\forall \tau.\ \tau \in A'\ Z \longrightarrow result\text{-}exn\ (return\ s\ \tau)\ \tau \in A))\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *spec*: $\forall s\in S.\ \forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ Z)\ Call\ (p\ s)\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{t/F} P\ (dynCall\text{-}exn\ f\ UNIV\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$

⟨*proof*⟩

lemma *DynProcStaticSpec*:
assumes *adapt*: $P \subseteq \{s.\ s \in S \wedge (\exists Z.\ init\ s \in P'\ Z\ \wedge$
$(\forall \tau.\ \tau \in Q'\ Z \longrightarrow return\ s\ \tau \in R\ s\ \tau) \wedge$
$(\forall \tau.\ \tau \in A'\ Z \longrightarrow return\ s\ \tau \in A))\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *spec*: $\forall s{\in}S.\ \forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ Z)\ Call\ (p\ s)\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{t/F} P\ (dynCall\ init\ p\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *DynProc-exnProcPar*:
assumes *adapt*: $P \subseteq \{s.\ p\ s = q \wedge (\exists Z.\ init\ s \in P'\ Z\ \wedge$
$(\forall \tau.\ \tau \in Q'\ Z \longrightarrow return\ s\ \tau \in R\ s\ \tau) \wedge$
$(\forall \tau.\ \tau \in A'\ Z \longrightarrow result\text{-}exn\ (return\ s\ \tau)\ \tau \in A))\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *spec*: $\forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ Z)\ Call\ q\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{t/F} P\ (dynCall\text{-}exn\ f\ UNIV\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *DynProcProcPar*:
assumes *adapt*: $P \subseteq \{s.\ p\ s = q \wedge (\exists Z.\ init\ s \in P'\ Z\ \wedge$
$(\forall \tau.\ \tau \in Q'\ Z \longrightarrow return\ s\ \tau \in R\ s\ \tau) \wedge$
$(\forall \tau.\ \tau \in A'\ Z \longrightarrow return\ s\ \tau \in A))\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *spec*: $\forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ Z)\ Call\ q\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{t/F} P\ (dynCall\ init\ p\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *DynProc-exnProcParNoAbrupt*:
assumes *adapt*: $P \subseteq \{s.\ p\ s = q \wedge (\exists Z.\ init\ s \in P'\ Z\ \wedge$
$(\forall \tau.\ \tau \in Q'\ Z \longrightarrow return\ s\ \tau \in R\ s\ \tau))\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *spec*: $\forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ Z)\ Call\ q\ (Q'\ Z),\{\}$
shows $\Gamma,\Theta\vdash_{t/F} P\ (dynCall\text{-}exn\ f\ UNIV\ init\ p\ return\ result\text{-}exn\ c)\ Q,A$
⟨*proof*⟩

lemma *DynProcProcParNoAbrupt*:
assumes *adapt*: $P \subseteq \{s.\ p\ s = q \wedge (\exists Z.\ init\ s \in P'\ Z\ \wedge$
$(\forall \tau.\ \tau \in Q'\ Z \longrightarrow return\ s\ \tau \in R\ s\ \tau))\}$
assumes *c*: $\forall s\ t.\ \Gamma,\Theta\vdash_{t/F} (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *spec*: $\forall Z.\ \Gamma,\Theta\vdash_{t/F} (P'\ Z)\ Call\ q\ (Q'\ Z),\{\}$
shows $\Gamma,\Theta\vdash_{t/F} P\ (dynCall\ init\ p\ return\ c)\ Q,A$
⟨*proof*⟩

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*: $\forall s\ t.\ t \in$ (*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* σ),{}
shows $\Gamma,\Theta\vdash_{t/F}$ P (*dynCall-exn f g init p return result-exn c*) Q,A
⟨*proof*⟩

lemma *DynProcModifyReturnNoAbr*:
assumes *to-prove*: $\Gamma,\Theta\vdash_{t/F}$ P (*dynCall init p return' c*) Q,A
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in$ (*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* σ),{}
shows $\Gamma,\Theta\vdash_{t/F}$ P (*dynCall init p return c*) Q,A
⟨*proof*⟩

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*: $\forall s\ t.\ t \in$ (*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* σ),{}
shows $\Gamma,\Theta\vdash_{t/F}$ P (*dynCall-exn f g init p return result-exn c*) Q,A
⟨*proof*⟩

lemma *ProcDynModifyReturnNoAbrSameFaults*:
assumes *to-prove*: $\Gamma,\Theta\vdash_{t/F}$ P (*dynCall init p return' c*) Q,A
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in$ (*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* σ),{}
shows $\Gamma,\Theta\vdash_{t/F}$ P (*dynCall init p return c*) Q,A
⟨*proof*⟩

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*: $\Gamma,\Theta\vdash_{t/F}$ 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 *ret-abr-modif*: $\forall s\ t.\ t \in$ (*ModifAbr* (*init s*))
$\longrightarrow$ *result-exn* (*return' s t*) *t* = *result-exn* (*return s t*) *t*
assumes *modif-clause*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/UNIV}$ $\{\sigma\}$ (*Call q*) (*Modif* σ),(*ModifAbr* σ)

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*: $\forall s\ t.\ t \in (Modif\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *ret-abr-modif*: $\forall s\ t.\ t \in (ModifAbr\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
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*: $\forall s\ t.\ t \in (Modif\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *ret-abr-modif*: $\forall s\ t.\ t \in (ModifAbr\ (init\ s))$
$\longrightarrow result\text{-}exn\ (return'\ s\ t)\ t = result\text{-}exn\ (return\ s\ t)\ t$
assumes *modif-clause*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ Call\ q\ (Modif\ \sigma),(ModifAbr\ \sigma)$
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*: $\forall s\ t.\ t \in (Modif\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *ret-abr-modif*: $\forall s\ t.\ t \in (ModifAbr\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *modif-clause*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ Call\ q\ (Modif\ \sigma),(ModifAbr\ \sigma)$
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*: $\Gamma,\Theta\vdash_{t/F} P'$ $(dynCall\text{-}exn\ f\ g\ init\ p\ return'\ result\text{-}exn\ c)$ Q,A
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (Modif\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *modif-clause*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/UNIV} \{\sigma\}\ (Call\ q)\ (Modif\ \sigma),\{\}$
shows $\Gamma,\Theta\vdash_{t/F} P$ $(dynCall\text{-}exn\ f\ g\ init\ p\ return\ result\text{-}exn\ c)$ 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*: $\Gamma,\Theta\vdash_{t/F} P'$ $(dynCall\ init\ p\ return'\ c)$ Q,A
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (Modif\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *modif-clause*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/UNIV} \{\sigma\}\ (Call\ q)\ (Modif\ \sigma),\{\}$
shows $\Gamma,\Theta\vdash_{t/F} P$ $(dynCall\ init\ p\ return\ c)$ 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*: $\Gamma,\Theta\vdash_{t/F} P'$ $(dynCall\text{-}exn\ f\ g\ init\ p\ return'\ result\text{-}exn\ c)$ Q,A
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (Modif\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *modif-clause*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ (Call\ q)\ (Modif\ \sigma),\{\}$
shows $\Gamma,\Theta\vdash_{t/F} P$ $(dynCall\text{-}exn\ f\ g\ init\ p\ return\ result\text{-}exn\ c)$ 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*: $\Gamma,\Theta\vdash_{t/F} P'$ $(dynCall\ init\ p\ return'\ c)$ Q,A
assumes *ret-nrm-modif*: $\forall s\ t.\ t \in (Modif\ (init\ s))$
$\longrightarrow return'\ s\ t = return\ s\ t$
assumes *modif-clause*:
$\forall \sigma.\ \Gamma,\Theta\vdash_{/F} \{\sigma\}\ (Call\ q)\ (Modif\ \sigma),\{\}$
shows $\Gamma,\Theta\vdash_{t/F} P$ $(dynCall\ init\ p\ return\ c)$ Q,A
⟨*proof*⟩

lemma *MergeGuards-iff*: $\Gamma,\Theta\vdash_{t/F}$ P *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*: $\Gamma,\{\}\vdash_{/\{\}}$ P c'' *UNIV*,*UNIV*
assumes c'': $c''=$ *mark-guards False* (*strip-guards* $(-F)$ c')
assumes c: *merge-guards* c = *merge-guards* (*mark-guards False* c')
shows $\Gamma,\Theta\vdash_{t/\{\}}$ P c Q,A
⟨*proof*⟩

lemma *CombineStrip''*:
assumes *deriv*: $\Gamma,\Theta\vdash_{t/\{True\}}$ P c' Q,A
assumes *deriv-strip-triv*: $\Gamma,\{\}\vdash_{/\{\}}$ P c'' *UNIV*,*UNIV*
assumes c'': $c''=$ *mark-guards False* (*strip-guards* ({*False*}) c')
assumes c: *merge-guards* c = *merge-guards* (*mark-guards False* c')
shows $\Gamma,\Theta\vdash_{t/\{\}}$ P c Q,A
⟨*proof*⟩

lemma *AsmUN*:
$(\bigcup Z.\ \{(P\ Z,\ p,\ Q\ Z, A\ Z)\}) \subseteq \Theta$
$\Longrightarrow$
$\forall Z.\ \Gamma,\Theta\vdash_{t/F}$ $(P\ Z)$ (*Call* p) $(Q\ Z),(A\ Z)$
⟨*proof*⟩

lemma *hoaret-to-hoarep'*:
$\forall Z.\ \Gamma,\{\}\vdash_{t/F}$ $(P\ Z)$ p $(Q\ Z),(A\ Z) \Longrightarrow \forall Z.\ \Gamma,\{\}\vdash_{/F}$ $(P\ Z)$ p $(Q\ Z),(A\ Z)$
⟨*proof*⟩

lemma *augment-context'*:
$[\![\Theta \subseteq \Theta';\ \forall Z.\ \Gamma,\Theta\vdash_{t/F}$ $(P\ Z)$ p $(Q\ Z),(A\ Z)]\!]$
$\Longrightarrow \forall Z.\ \Gamma,\Theta'\vdash_{t/F}$ $(P\ Z)$ p $(Q\ Z),(A\ Z)$
⟨*proof*⟩

lemma *augment-emptyFaults*:
$[\![\forall Z.\ \Gamma,\{\}\vdash_{t/\{\}}$ $(P\ Z)$ p $(Q\ Z),(A\ Z)]\!] \Longrightarrow$
$\forall Z.\ \Gamma,\{\}\vdash_{t/F}$ $(P\ Z)$ p $(Q\ Z),(A\ Z)$
⟨*proof*⟩

lemma *augment-FaultsUNIV*:
$[\![\forall Z.\ \Gamma,\{\}\vdash_{t/F}$ $(P\ Z)$ p $(Q\ Z),(A\ Z)]\!] \Longrightarrow$
$\forall Z.\ \Gamma,\{\}\vdash_{t/UNIV}$ $(P\ Z)$ p $(Q\ Z),(A\ Z)$
⟨*proof*⟩

lemma *PostConjI* [*trans*]:
$[\![\Gamma,\Theta\vdash_{t/F}\ P\ c\ Q,A;\ \Gamma,\Theta\vdash_{t/F}\ P\ c\ R,B]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ c\ (Q \cap R),(A \cap B)$
⟨*proof*⟩

lemma *PostConjI′* :
$[\![\Gamma,\Theta\vdash_{t/F}\ P\ c\ Q,A;\ \Gamma,\Theta\vdash_{t/F}\ P\ c\ Q,A \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ c\ R,B]\!]$
$\Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ c\ (Q \cap R),(A \cap B)$
⟨*proof*⟩

lemma *PostConjE* [*consumes 1*]:
assumes *conj*: $\Gamma,\Theta\vdash_{t/F}\ P\ c\ (Q \cap R),(A \cap B)$
assumes *E*: $[\![\Gamma,\Theta\vdash_{t/F}\ P\ c\ Q,A;\ \Gamma,\Theta\vdash_{t/F}\ P\ c\ R,B]\!] \Longrightarrow S$
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*]:
$[\![\Gamma,\Theta\vdash_{t/F}\ P\ anno\ Q,A;\ c = anno]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *annotate-normI*:
assumes *deriv-anno*: $\Gamma,\Theta\vdash_{t/F}P\ anno\ Q,A$
assumes *norm-eq*: *normalize c* = *normalize anno*
shows $\Gamma,\Theta\vdash_{t/F}P\ c\ Q,A$
⟨*proof*⟩

lemma *annotateWhile*:
$[\![\Gamma,\Theta\vdash_{t/F}\ P\ (whileAnnoG\ gs\ b\ I\ V\ c)\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ (while\ gs\ b\ c)\ Q,A$
⟨*proof*⟩

lemma *reannotateWhile*:
$[\![\Gamma,\Theta\vdash_{t/F}\ P\ (whileAnnoG\ gs\ b\ I\ V\ c)\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ (whileAnnoG\ gs\ b\ J\ V\ c)\ Q,A$
⟨*proof*⟩

lemma *reannotateWhileNoGuard*:
$[\![\Gamma,\Theta\vdash_{t/F}\ P\ (whileAnno\ b\ I\ V\ c)\ Q,A]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ (whileAnno\ b\ J\ V\ c)\ Q,A$
⟨*proof*⟩

lemma [*trans*] : $P' \subseteq P \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ c\ Q,A \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P'\ c\ Q,A$
⟨*proof*⟩

lemma [*trans*]: $Q \subseteq Q' \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ c\ Q,A \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ c\ Q',A$
⟨*proof*⟩

lemma [*trans*]:
$\Gamma,\Theta\vdash_{t/F}\ \{s.\ P\ s\}\ c\ Q,A \Longrightarrow (\bigwedge s.\ P'\ s \longrightarrow P\ s) \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ \{s.\ P'\ s\}\ c\ Q,A$
⟨*proof*⟩

lemma [*trans*]:
$(\bigwedge s.\ P'\ s \longrightarrow P\ s) \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ \{s.\ P\ s\}\ c\ Q,A \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ \{s.\ P'\ s\}\ c\ Q,A$
⟨*proof*⟩

lemma [*trans*]:
$\Gamma,\Theta\vdash_{t/F}\ P\ c\ \{s.\ Q\ s\},A \Longrightarrow (\bigwedge s.\ Q\ s \longrightarrow Q'\ s) \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ c\ \{s.\ Q'\ s\},A$
⟨*proof*⟩

lemma [*trans*]:
$(\bigwedge s.\ Q\ s \longrightarrow Q'\ s) \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ c\ \{s.\ Q\ s\},A \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ c\ \{s.\ Q'\ s\},A$
⟨*proof*⟩

lemma [*intro?*]: $\Gamma,\Theta\vdash_{t/F}\ P\ Skip\ P,A$
⟨*proof*⟩

lemma *CondInt* [*trans,intro?*]:
$[\![\Gamma,\Theta\vdash_{t/F}\ (P \cap b)\ c1\ Q,A;\ \Gamma,\Theta\vdash_{t/F}\ (P \cap -\ b)\ c2\ Q,A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F}\ P\ (Cond\ b\ c1\ c2)\ Q,A$
⟨*proof*⟩

lemma *CondConj* [*trans, intro?*]:
$[\![\Gamma,\Theta\vdash_{t/F}\ \{s.\ P\ s \wedge b\ s\}\ c1\ Q,A;\ \Gamma,\Theta\vdash_{t/F}\ \{s.\ P\ s \wedge \neg\ b\ s\}\ c2\ Q,A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{t/F}\ \{s.\ P\ s\}\ (Cond\ \{s.\ b\ s\}\ c1\ c2)\ Q,A$
⟨*proof*⟩
end

12 Auxiliary Definitions/Lemmas to Facilitate Hoare Logic

theory *Hoare* **imports** *HoarePartial HoareTotal* **begin**

syntax

-hoarep-emptyFaults::
$[('s,'p,'f)\ body,('s,'p)\ quadruple\ set,$
$'f\ set,'s\ assn,('s,'p,'f)\ com,\ 's\ assn,'s\ assn] => bool$
$(\langle(3\text{-},\text{-}/\vdash\ (\text{-}/\ (\text{-})/\ \text{-},/\text{-}))\rangle\ [61,60,1000,20,1000,1000]60)$

-hoarep-emptyCtx::
$[('s,'p,'f)\ body,'f\ set,'s\ assn,('s,'p,'f)\ com,\ 's\ assn,'s\ assn] => bool$
$(\langle(3\text{-}/\vdash_{'/\text{-}}\ (\text{-}/\ (\text{-})/\ \text{-},/\text{-}))\rangle\ [61,60,1000,20,1000,1000]60)$

-hoarep-emptyCtx-emptyFaults::
$[('s,'p,'f)\ body,'s\ assn,('s,'p,'f)\ com,\ 's\ assn,'s\ assn] => bool$
$(\langle(3\text{-}/\vdash\ (\text{-}/\ (\text{-})/\ \text{-},/\text{-}))\rangle\ [61,1000,20,1000,1000]60)$

-hoarep-noAbr::
$[('s,'p,'f)\ body,('s,'p)\ quadruple\ set,'f\ set,$
$'s\ assn,('s,'p,'f)\ com,\ 's\ assn] => bool$
$(\langle(3\text{-},\text{-}/\vdash_{'/\text{-}}\ (\text{-}/\ (\text{-})/\ \text{-}))\rangle\ [61,60,60,1000,20,1000]60)$

-hoarep-noAbr-emptyFaults::
$[('s,'p,'f)\ body,('s,'p)\ quadruple\ set,'s\ assn,('s,'p,'f)\ com,\ 's\ assn] => bool$
$(\langle(3\text{-},\text{-}/\vdash\ (\text{-}/\ (\text{-})/\ \text{-}))\rangle\ [61,60,1000,20,1000]60)$

-hoarep-emptyCtx-noAbr::
$[('s,'p,'f)\ body,'f\ set,'s\ assn,('s,'p,'f)\ com,\ 's\ assn] => bool$
$(\langle(3\text{-}/\vdash_{'/\text{-}}\ (\text{-}/\ (\text{-})/\ \text{-}))\rangle\ [61,60,1000,20,1000]60)$

-hoarep-emptyCtx-noAbr-emptyFaults::
$[('s,'p,'f)\ body,'s\ assn,('s,'p,'f)\ com,\ 's\ assn] => bool$
$(\langle(3\text{-}/\vdash\ (\text{-}/\ (\text{-})/\ \text{-}))\rangle\ [61,1000,20,1000]60)$

-hoaret-emptyFaults::
$[('s,'p,'f)\ body,('s,'p)\ quadruple\ set,$
$'s\ assn,('s,'p,'f)\ com,\ 's\ assn,'s\ assn] => bool$
$(\langle(3\text{-},\text{-}/\vdash_t\ (\text{-}/\ (\text{-})/\ \text{-},/\text{-}))\rangle\ [61,60,1000,20,1000,1000]60)$

-hoaret-emptyCtx::
$[('s,'p,'f)\ body,'f\ set,'s\ assn,('s,'p,'f)\ com,\ 's\ assn,'s\ assn] => bool$
$(\langle(3\text{-}/\vdash_{t'/\text{-}}\ (\text{-}/\ (\text{-})/\ \text{-},/\text{-}))\rangle\ [61,60,1000,20,1000,1000]60)$

-hoaret-emptyCtx-emptyFaults::
$[('s,'p,'f)\ body,'s\ assn,('s,'p,'f)\ com,\ 's\ assn,'s\ assn] => bool$
$(\langle(3\text{-}/\vdash_t\ (\text{-}/\ (\text{-})/\ \text{-},/\text{-}))\rangle\ [61,1000,20,1000,1000]60)$

-hoaret-noAbr::
$[('s,'p,'f)\ body,'f\ set,\ ('s,'p)\ quadruple\ set,$
$'s\ assn,('s,'p,'f)\ com,\ 's\ assn] => bool$
$(\langle(3\text{-},\text{-}/\vdash_{t'/\text{-}}\ (\text{-}/\ (\text{-})/\ \text{-}))\rangle\ [61,60,60,1000,20,1000]60)$

-hoaret-noAbr-emptyFaults::
[('s,'p,'f) body,('s,'p) quadruple set,'s assn,('s,'p,'f) com, 's assn] => bool
(‹(3-,-/⊢$_t$ (-/ (-)/ -))› [61,60,1000,20,1000]60)

-hoaret-emptyCtx-noAbr::
[('s,'p,'f) body,'f set,'s assn,('s,'p,'f) com, 's assn] => bool
(‹(3-/⊢$_{t'/-}$ (-/ (-)/ -))› [61,60,1000,20,1000]60)

-hoaret-emptyCtx-noAbr-emptyFaults::
[('s,'p,'f) body,'s assn,('s,'p,'f) com, 's assn] => bool
(‹(3-/⊢$_t$ (-/ (-)/ -))› [61,1000,20,1000]60)

syntax (*ASCII*)

-hoarep-emptyFaults::
[('s,'p,'f) body,('s,'p) quadruple set,
's assn,('s,'p,'f) com, 's assn,'s assn] ⇒ bool
(‹(3-,-/|− (-/ (-)/ -,/-))› [61,60,1000,20,1000,1000]60)

-hoarep-emptyCtx::
[('s,'p,'f) body,'f set,'s assn,('s,'p,'f) com, 's assn,'s assn] => bool
(‹(3-/|−'/- (-/ (-)/ -,/-))› [61,60,1000,20,1000,1000]60)

-hoarep-emptyCtx-emptyFaults::
[('s,'p,'f) body,'s assn,('s,'p,'f) com, 's assn,'s assn] => bool
(‹(3-/|−(-/ (-)/ -,/-))› [61,1000,20,1000,1000]60)

-hoarep-noAbr::
[('s,'p,'f) body,('s,'p) quadruple set,'f set,
's assn,('s,'p,'f) com, 's assn] => bool
(‹(3-,-/|−'/- (-/ (-)/ -))› [61,60,60,1000,20,1000]60)

-hoarep-noAbr-emptyFaults::
[('s,'p,'f) body,('s,'p) quadruple set,'s assn,('s,'p,'f) com, 's assn] => bool
(‹(3-,-/|−(-/ (-)/ -))› [61,60,1000,20,1000]60)

-hoarep-emptyCtx-noAbr::
[('s,'p,'f) body,'f set,'s assn,('s,'p,'f) com, 's assn] => bool
(‹(3-/|−'/- (-/ (-)/ -))› [61,60,1000,20,1000]60)

-hoarep-emptyCtx-noAbr-emptyFaults::
[('s,'p,'f) body,'s assn,('s,'p,'f) com, 's assn] => bool
(‹(3-/|−(-/ (-)/ -))› [61,1000,20,1000]60)

-hoaret-emptyFault::
[('s,'p,'f) body,('s,'p) quadruple set,
's assn,('s,'p,'f) com, 's assn,'s assn] => bool

(‹(3-,-/|−t (-/ (-)/ -,/-))› [61,60,1000,20,1000,1000]60)

-hoaret-emptyCtx::
[('s,'p,'f) body,'f set,'s assn,('s,'p,'f) com, 's assn,'s assn] => bool
(‹(3-/|−t'/- (-/ (-)/ -,/-))› [61,60,1000,20,1000,1000]60)

-hoaret-emptyCtx-emptyFaults::
[('s,'p,'f) body,'s assn,('s,'p,'f) com, 's assn,'s assn] => bool
(‹(3-/|−t(-/ (-)/ -,/-))› [61,1000,20,1000,1000]60)

-hoaret-noAbr::
[('s,'p,'f) body,('s,'p) quadruple set,'f set,
's assn,('s,'p,'f) com, 's assn] => bool
(‹(3-,-/|−t'/- (-/ (-)/ -))› [61,60,60,1000,20,1000]60)

-hoaret-noAbr-emptyFaults::
[('s,'p,'f) body,('s,'p) quadruple set,'s assn,('s,'p,'f) com, 's assn] => bool
(‹(3-,-/|−t(-/ (-)/ -))› [61,60,1000,20,1000]60)

-hoaret-emptyCtx-noAbr::
[('s,'p,'f) body,'f set,'s assn,('s,'p,'f) com, 's assn] => bool
(‹(3-/|−t'/- (-/ (-)/ -))› [61,60,1000,20,1000]60)

-hoaret-emptyCtx-noAbr-emptyFaults::
[('s,'p,'f) body,'s assn,('s,'p,'f) com, 's assn] => bool
(‹(3-/|−t(-/ (-)/ -))› [61,1000,20,1000]60)

translations

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

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

$\Gamma\vdash\ P\ c\ Q\ \ \ \ == \ \Gamma\vdash_{/\{\}}\ P\ c\ Q$
$\Gamma\vdash_{/F}\ P\ c\ Q\ \ == \Gamma,\{\}\vdash_{/F}\ P\ c\ Q$
$\Gamma\vdash_{/F}\ P\ c\ Q\ \ <= \ \Gamma\vdash_{/F}\ P\ c\ Q,\{\}$
$\Gamma\vdash\ P\ c\ Q\ \ \ \ <= \ \Gamma\vdash\ P\ c\ Q,\{\}$

$\Gamma\vdash_t\ P\ c\ Q,A\ \ \ == \Gamma\vdash_{t/\{\}}\ P\ c\ Q,A$
$\Gamma\vdash_{t/F}\ P\ c\ Q,A\ \ == \Gamma,\{\}\vdash_{t/F}\ P\ c\ Q,A$

$\Gamma,\Theta\vdash_t\ P\ c\ Q\quad == \Gamma,\Theta\vdash_{t/\{\}}\ P\ c\ Q$
$\Gamma,\Theta\vdash_{t/F}\ P\ c\ Q == \Gamma,\Theta\vdash_{t/F}\ P\ c\ Q,\{\}$
$\Gamma,\Theta\vdash_t\ P\ c\ Q,A\quad == \Gamma,\Theta\vdash_{t/\{\}}\ P\ c\ Q,A$

$\Gamma\vdash_t\ P\ c\ Q\quad == \Gamma\vdash_{t/\{\}}\ P\ c\ Q$
$\Gamma\vdash_{t/F}\ P\ c\ Q\ == \Gamma,\{\}\vdash_{t/F}\ P\ c\ Q$
$\Gamma\vdash_{t/F}\ P\ c\ Q\ <=\ \Gamma\vdash_{t/F}\ P\ c\ Q,\{\}$
$\Gamma\vdash_t\ P\ c\ Q\quad <=\ \Gamma\vdash_t\ P\ c\ Q,\{\}$

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 Γ::('s,'p,'f) *body*

primrec *assoc*:: ('a ×'b) *list* ⇒ 'a ⇒ 'b
where
assoc [] *x* = *undefined* |
assoc (*p*#*ps*) *x* = (*if fst p* = *x then* (*snd p*) *else assoc ps x*)

lemma *conjE-simp*: ($P \wedge Q \Longrightarrow PROP\ R$) ≡ ($P \Longrightarrow Q \Longrightarrow PROP\ R$)
 ⟨*proof*⟩

lemma *CollectInt-iff*: $\{s.\ P\ s\} \cap \{s.\ Q\ s\} = \{s.\ P\ s \wedge Q\ s\}$
⟨*proof*⟩

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

lemma *Collect-False*: $\{s.\ False\} = \{\}$
⟨*proof*⟩

lemma *Collect-True*: $\{s.\ True\} = UNIV$
⟨*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*: $f \in insert\ f\ X$
⟨*proof*⟩

lemma *lookup-Some-in-dom*: $\Gamma\ p = Some\ bdy \Longrightarrow p \in dom\ \Gamma$
⟨*proof*⟩

lemma *unit-object*: $(\forall\, u::unit.\ P\ u) = P\ ()$
⟨*proof*⟩

lemma *unit-ex*: $(\exists\, u::unit.\ P\ u) = P\ ()$

⟨*proof*⟩

lemma *unit-meta*: $(\bigwedge(u::unit).\ PROP\ P\ u) \equiv PROP\ P\ ()$
⟨*proof*⟩

lemma *unit-UN*: $(\bigcup z::unit.\ P\ z) = P\ ()$
⟨*proof*⟩

lemma *subset-singleton-insert1*: $y = x \Longrightarrow \{y\} \subseteq insert\ x\ A$
⟨*proof*⟩

lemma *subset-singleton-insert2*: $\{y\} \subseteq A \Longrightarrow \{y\} \subseteq insert\ x\ A$
⟨*proof*⟩

lemma *in-Specs-simp*: $(\forall x\in\bigcup Z.\ \{(P\ Z,\ p,\ Q\ Z,\ A\ Z)\}.\ Prop\ x) =$
$(\forall Z.\ Prop\ (P\ Z,p,Q\ Z,A\ Z))$
⟨*proof*⟩

lemma *in-set-Un-simp*: $(\forall x\in A \cup B.\ P\ x) = ((\forall x \in A.\ P\ x) \wedge (\forall x \in B.\ P\ x))$
⟨*proof*⟩

lemma *split-all-conj*: $(\forall x.\ P\ x \wedge Q\ x) = ((\forall x.\ P\ x) \wedge (\forall x.\ Q\ x))$
⟨*proof*⟩

lemma *image-Un-single-simp*: $f\ `\ (\bigcup Z.\ \{P\ Z\}) = (\bigcup Z.\ \{f\ (P\ Z)\})$
⟨*proof*⟩

lemma *measure-lex-prod-def'*:
$f <*mlex*> r \equiv (\{(x,y).\ (x,y) \in measure\ f \vee f\ x=f\ y \wedge (x,y) \in\ r\})$
⟨*proof*⟩

lemma *in-measure-iff*: $(x,y) \in measure\ f = (f\ x < f\ y)$
⟨*proof*⟩

lemma *in-lex-iff*:
$((a,b),(x,y)) \in r <*lex*> s = ((a,x) \in r \vee (a=x \wedge (b,y)\in s))$
⟨*proof*⟩

lemma *in-mlex-iff*:
$(x,y) \in f <*mlex*> r = (f\ x < f\ y \vee (f\ x=f\ y \wedge (x,y) \in r))$
⟨*proof*⟩

lemma *in-inv-image-iff*: $(x,y) \in inv\text{-}image\ r\ f = ((f\ x,\ f\ y) \in r)$
⟨*proof*⟩

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* ⇒ *′g*) ⇒ (*′g*,*′z*) *state-scheme* ⇒ (*′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* = (λ*s*. *s*(|*globals* := *f* (*globals s*)|))
  ⟨*proof*⟩

record (*′g*, *′l*) *state-locals* = *′g state* +
  locals :: *′l*

type-synonym (*′g*, *′n*, *′val*) *stateSP* = (*′g*, *′n* ⇒ *′val*) *state-locals*
type-synonym (*′g*, *′n*, *′val*, *′x*) *stateSP-scheme* = (*′g*, *′n* ⇒ *′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. Thats 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 × ('s,'p,'f) com list*

type-synonym *('s,'p,'f) config =*
('s,'p,'f)com list × ('s,'p,'f)continuation list × ('s,'f) xstate

inductive *step*::[*('s,'p,'f) body*,*('s,'p,'f) config*,*('s,'p,'f) config*] ⇒ *bool*
(‹-⊢ (- →/ -)› [*81,81,81*] *100*)
for Γ::(*'s,'p,'f*) *body*
where
Skip: Γ⊢(*Skip#cs*,*css*,*Normal s*) → (*cs*,*css*,*Normal s*)
| *Guard*: *s*∈*g* ⟹ Γ⊢(*Guard f g c#cs*,*css*,*Normal s*) → (*c#cs*,*css*,*Normal s*)

| *GuardFault*: *s*∉*g* ⟹ Γ⊢(*Guard f g c#cs*,*css*,*Normal s*) → (*cs*,*css*,*Fault f*)

| *FaultProp*: Γ⊢(*c#cs*,*css*,*Fault f*) → (*cs*,*css*,*Fault f*)
| *FaultPropBlock*: Γ⊢([],(*nrms*,*abrs*)#*css*,*Fault f*) → (*nrms*,*css*,*Fault f*)

| *AbruptProp*: Γ⊢(*c#cs*,*css*,*Abrupt s*) → (*cs*,*css*,*Abrupt s*)

| *ExitBlockNormal*:
Γ⊢([],(*nrms*,*abrs*)#*css*,*Normal s*) → (*nrms*,*css*,*Normal s*)
| *ExitBlockAbrupt*:
Γ⊢([],(*nrms*,*abrs*)#*css*,*Abrupt s*) → (*abrs*,*css*,*Normal s*)

| *Basic*: Γ⊢(*Basic f#cs*,*css*,*Normal s*) → (*cs*,*css*,*Normal* (*f s*))

| *Spec*: (*s*,*t*) ∈ *r* ⟹ Γ⊢(*Spec r#cs*,*css*,*Normal s*) → (*cs*,*css*,*Normal t*)
| *SpecStuck*: ∀ *t*. (*s*,*t*) ∉ *r* ⟹ Γ⊢(*Spec r#cs*,*css*,*Normal s*) → (*cs*,*css*,*Stuck*)

| *Seq*: Γ⊢(*Seq* c_1 c_2*#cs*,*css*,*Normal s*) → (c_1#c_2*#cs*,*css*,*Normal s*)

| *CondTrue*: *s*∈*b* ⟹ Γ⊢(*Cond b* c_1 c_2*#cs*,*css*,*Normal s*) → (c_1*#cs*,*css*,*Normal s*)
| *CondFalse*: *s*∉*b* ⟹ Γ⊢(*Cond b* c_1 c_2*#cs*,*css*,*Normal s*) → (c_2*#cs*,*css*,*Normal s*)

| *WhileTrue*: $[\![s \in b]\!]$
$\Longrightarrow$
$\Gamma \vdash (\textit{While } b\ c\#cs,css,\textit{Normal } s) \rightarrow (c\#\textit{While } b\ c\#cs,css,\textit{Normal } s)$
| *WhileFalse*: $[\![s \notin b]\!]$
$\Longrightarrow$
$\Gamma \vdash (\textit{While } b\ c\#cs,css,\textit{Normal } s) \rightarrow (cs,css,\textit{Normal } s)$

| *Call*: $\Gamma\ p = \textit{Some bdy} \Longrightarrow$
$\Gamma \vdash (\textit{Call } p\#cs,css,\textit{Normal } s) \rightarrow ([bdy],(cs,\textit{Throw}\#cs)\#css,\textit{Normal } s)$

| *CallUndefined*: $\Gamma\ p = \textit{None} \Longrightarrow$
$\Gamma \vdash (\textit{Call } p\#cs,css,\textit{Normal } s) \rightarrow (cs,css,\textit{Stuck})$

| *StuckProp*: $\Gamma \vdash (c\#cs,css,\textit{Stuck}) \rightarrow (cs,css,\textit{Stuck})$
| *StuckPropBlock*: $\Gamma \vdash ([],(nrms,abrs)\#css,\textit{Stuck}) \rightarrow (nrms,css,\textit{Stuck})$

| *DynCom*: $\Gamma \vdash (\textit{DynCom } c\#cs,css,\textit{Normal } s) \rightarrow (c\ s\#cs,css,\textit{Normal } s)$

| *Throw*: $\Gamma \vdash (\textit{Throw}\#cs,css,\textit{Normal } s) \rightarrow (cs,css,\textit{Abrupt } s)$
| *Catch*: $\Gamma \vdash (\textit{Catch } c_1\ c_2\#cs,css,\textit{Normal } s) \rightarrow ([c_1],(cs,c_2\#cs)\#css,\textit{Normal } s)$

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*]:
$\Gamma \vdash (c\#cs,css,\textit{Fault } f) \rightarrow u$
$\Gamma \vdash ([],css,\textit{Fault } f) \rightarrow u$
$\Gamma \vdash (c\#cs,css,\textit{Stuck}) \rightarrow u$
$\Gamma \vdash ([],css,\textit{Stuck}) \rightarrow u$
$\Gamma \vdash (c\#cs,css,\textit{Abrupt } s) \rightarrow u$
$\Gamma \vdash ([],css,\textit{Abrupt } s) \rightarrow u$
$\Gamma \vdash ([],css,\textit{Normal } s) \rightarrow u$
$\Gamma \vdash (\textit{Skip}\#cs,css,s) \rightarrow u$
$\Gamma \vdash (\textit{Guard } f\ g\ c\#cs,css,s) \rightarrow u$
$\Gamma \vdash (\textit{Basic } f\#cs,css,s) \rightarrow u$
$\Gamma \vdash (\textit{Spec } r\#cs,css,s) \rightarrow u$
$\Gamma \vdash (\textit{Seq } c1\ c2\#cs,css,s) \rightarrow u$
$\Gamma \vdash (\textit{Cond } b\ c1\ c2\#cs,css,s) \rightarrow u$
$\Gamma \vdash (\textit{While } b\ c\#cs,css,s) \rightarrow u$
$\Gamma \vdash (\textit{Call } p\#cs,css,s) \rightarrow u$
$\Gamma \vdash (\textit{DynCom } c\#cs,css,s) \rightarrow u$
$\Gamma \vdash (\textit{Throw}\#cs,css,s) \rightarrow u$
$\Gamma \vdash (\textit{Catch } c1\ c2\#cs,css,s) \rightarrow u$

inductive-cases *step-Normal-elim-cases* [*cases set*]:
Γ⊢(*c#cs*,*css*,*Fault f*) → *u*
Γ⊢([],*css*,*Fault f*) → *u*
Γ⊢(*c#cs*,*css*,*Stuck*) → *u*
Γ⊢([],*css*,*Stuck*) → *u*
Γ⊢([],(*nrms*,*abrs*)#*css*,*Normal s*) → *u*
Γ⊢([],(*nrms*,*abrs*)#*css*,*Abrupt s*) → *u*
Γ⊢(*Skip#cs*,*css*,*Normal s*) → *u*
Γ⊢(*Guard f g c#cs*,*css*,*Normal s*) → *u*
Γ⊢(*Basic f#cs*,*css*,*Normal s*) → *u*
Γ⊢(*Spec r#cs*,*css*,*Normal s*) → *u*
Γ⊢(*Seq c1 c2#cs*,*css*,*Normal s*) → *u*
Γ⊢(*Cond b c1 c2#cs*,*css*,*Normal s*) → *u*
Γ⊢(*While b c#cs*,*css*,*Normal s*) → *u*
Γ⊢(*Call p#cs*,*css*,*Normal s*) → *u*
Γ⊢(*DynCom c#cs*,*css*,*Normal s*) → *u*
Γ⊢(*Throw#cs*,*css*,*Normal s*) → *u*
Γ⊢(*Catch c1 c2#cs*,*css*,*Normal s*) → *u*

abbreviation
step-rtrancl :: [('s,'p,'f) *body*,('s,'p,'f) *config*,('s,'p,'f) *config*] ⇒ *bool*
(‹-⊢ (- →*/ -)› [*81*,*81*,*81*] *100*)
where
Γ⊢*cs0* →* *cs1* == (*step* Γ)** *cs0 cs1*

abbreviation

step-trancl :: [('s,'p,'f) *body*,('s,'p,'f) *config*,('s,'p,'f) *config*] ⇒ *bool*
(‹-⊢ (- →$^+$/ -)› [*81*,*81*,*81*] *100*)
where
Γ⊢*cs0* →$^+$ *cs1* == (*step* Γ)$^{++}$ *cs0 cs1*

14.1.1 Structural Properties of Small Step Computations

lemma *Fault-app-steps*: Γ⊢(*cs@xs*,*css*,*Fault f*) →* (*xs*,*css*,*Fault f*)
⟨*proof*⟩

lemma *Stuck-app-steps*: Γ⊢(*cs@xs*,*css*,*Stuck*) →* (*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*: Γ⊢(*cs*,*css*,*s*) → (*cs'*,*css'*,*t*)
shows *css*=*css'* ⟹ Γ⊢(*cs@xs*,*css*,*s*) → (*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*: Γ⊢(*cs*,*css*,*s*) → (*cs'*,*css'*,*t*)
  shows Γ⊢(*cs*,*css*@*xs*,*s*) → (*cs'*,*css'*@*xs*,*t*)
⟨*proof*⟩

⟨*ML*⟩

lemma *app-css-steps*:
  assumes *step*: Γ⊢(*cs*,*css*,*s*) →⁺ (*cs'*,*css'*,*t*)
  shows Γ⊢(*cs*,*css*@*xs*,*s*) →⁺ (*cs'*,*css'*@*xs*,*t*)
⟨*proof*⟩

lemma *step-Cons'*:
  assumes *step*: Γ⊢(*ccs*,*css*,*s*) → (*cs'*,*css'*,*t*)
  shows
  ⋀*c cs*. *ccs*=*c*#*cs* ⟹ ∃ *css''*. *css'*=*css''*@*css* ∧
    (*if css''*=[] *then* ∃ *p*. *cs'*=*p*@*cs*
     else (∃ *pnorm pabr*. *css''*=[(*pnorm*@*cs*,*pabr*@*cs*)]))
⟨*proof*⟩

lemma *step-Cons*:
  assumes *step*: Γ⊢(*c*#*cs*,*css*,*s*) → (*cs'*,*css'*,*t*)
  shows ∃ *pcss*. *css'*=*pcss*@*css* ∧
    (*if pcss*=[] *then* ∃ *ps*. *cs'*=*ps*@*cs*
     else (∃ *pcs-normal pcs-abrupt*. *pcss*=[(*pcs-normal*@*cs*,*pcs-abrupt*@*cs*)]))
⟨*proof*⟩

lemma *step-Nil'*:
  assumes *step*: Γ⊢(*cs*,*asscss*,*s*) → (*cs'*,*css'*,*t*)
  shows
  ⋀*ass*. ⟦*cs*=[]; *asscss*=*ass*@*css*; *ass*≠*Nil*⟧ ⟹
    css'=*tl ass*@*css* ∧
    (*case s of*
      Abrupt s' ⇒ *cs'*=*snd* (*hd ass*) ∧ *t*=*Normal s'*
     | - ⇒ *cs'*=*fst* (*hd ass*) ∧ *t*=*s*)
⟨*proof*⟩

lemma *step-Nil*:
  assumes *step*: Γ⊢([],*ass*@*css*,*s*) → (*cs'*,*css'*,*t*)
  assumes *ass-not-Nil*: *ass*≠[]
  shows *css'*=*tl ass*@*css* ∧
    (*case s of*
      Abrupt s' ⇒ *cs'*=*snd* (*hd ass*) ∧ *t*=*Normal s'*
     | - ⇒ *cs'*=*fst* (*hd ass*) ∧ *t*=*s*)
  ⟨*proof*⟩

lemma *step-Nil''*:
  assumes *step*: Γ⊢([],(*pcs-normal*,*pcs-abrupt*)#*pcss*@*css*,*s*) → (*cs'*,*pcss*@*css*,*t*)

shows (*case s of*
Abrupt s' ⇒ *cs'=pcs-abrupt* ∧ *t=Normal s'*
| - ⇒ *cs'=pcs-normal* ∧ *t=s*)
⟨*proof*⟩

lemma *drop-suffix-css-step'*:
assumes *step*: $\Gamma\vdash(cs,cssxs,s) \rightarrow (cs',css'xs,t)$
shows $\bigwedge css\ css'\ xs.\ [\![cssxs = css@xs;\ css'xs=css'@xs]\!]$
$\Longrightarrow \Gamma\vdash(cs,css,s) \rightarrow (cs',css',t)$
⟨*proof*⟩

lemma *drop-suffix-css-step*:
assumes *step*: $\Gamma\vdash(cs,pcss@css,s) \rightarrow (cs',pcss'@css,t)$
shows $\Gamma\vdash(cs,pcss,s) \rightarrow (cs',pcss',t)$
⟨*proof*⟩

lemma *drop-suffix-hd-css-step'*:
assumes *step*: $\Gamma\vdash (pcs,css,s) \rightarrow (cs',css'css,t)$
shows $\bigwedge p\ ps\ cs\ pnorm\ pabr.\ [\![pcs=p\#ps@cs;\ css'css=(pnorm@cs,pabr@cs)\#css]\!]$
$\Longrightarrow \Gamma\vdash (p\#ps,css,s) \rightarrow (cs',(pnorm,pabr)\#css,t)$
⟨*proof*⟩

lemma *drop-suffix-hd-css-step''*:
assumes *step*: $\Gamma\vdash (p\#ps@cs,css,s) \rightarrow (cs',(pnorm@cs,pabr@cs)\#css,t)$
shows $\Gamma\vdash (p\#ps,css,s) \rightarrow (cs',(pnorm,pabr)\#css,t)$
⟨*proof*⟩

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 $\Gamma\vdash (p\#ps,css,s) \rightarrow (cs',[(pnorm@ps,pabr@ps)]@css,t)$
⟨*proof*⟩

lemma *drop-suffix'*:
assumes *step*: $\Gamma\vdash(csxs,css,s) \rightarrow (cs'xs,css',t)$
shows $\bigwedge xs\ cs\ cs'.\ [\![css=css';\ csxs=cs@xs;\ cs'xs = cs'@xs;\ cs\neq[]\ ]\!]$
$\Longrightarrow \Gamma\vdash(cs,css,s) \rightarrow (cs',css,t)$
⟨*proof*⟩

lemma *drop-suffix*:
assumes *step*: $\Gamma\vdash(c\#cs@xs,css,s) \rightarrow (cs'@xs,css,t)$
shows $\Gamma\vdash(c\#cs,css,s) \rightarrow (cs',css,t)$
⟨*proof*⟩

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 $\Gamma\vdash(cs,xss,s) \rightarrow (cs',xss,t)$
⟨*proof*⟩

lemma *Cons-change-css-step*:
  assumes *step*: $\Gamma\vdash$ $(cs,css,s) \rightarrow (cs',css'@css,t)$
  shows $\Gamma\vdash$ $(cs,xss,s) \rightarrow (cs',css'@xss,t)$
⟨*proof*⟩

lemma *Nil-change-css-step*:
  assumes *step*: $\Gamma\vdash([],ass@css,s) \rightarrow (cs',ass'@css,t)$
  assumes *ass-not-Nil*: $ass\neq[]$
  shows $\Gamma\vdash([],ass@xss,s) \rightarrow (cs',ass'@xss,t)$
⟨*proof*⟩

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 $\bigwedge cs\ css.\ \Gamma\vdash(c\#cs,css,s) \rightarrow^* (cs,css,t)$
⟨*proof*⟩

inductive *execs*::[('*s*,'*p*,'*f*) *body*,('*s*,'*p*,'*f*) *com list*,
  ('*s*,'*p*,'*f*) *continuation list*,
  ('*s*,'*f*) *xstate*,('*s*,'*f*) *xstate*] ⇒ *bool*
  (‹-⊢ ⟨-,-,-⟩ ⇒ -› [*50*,*50*,*50*,*50*,*50*] *50*)
  for Γ:: ('*s*,'*p*,'*f*) *body*
where
  Nil: $\Gamma\vdash\langle[],[],s\rangle \Rightarrow s$

| *ExitBlockNormal*: $\Gamma\vdash\langle nrms,css,Normal\ s\rangle \Rightarrow t$
  $\Longrightarrow$
  $\Gamma\vdash\langle[],(nrms,abrs)\#css,Normal\ s\rangle \Rightarrow t$

| *ExitBlockAbrupt*: $\Gamma\vdash\langle abrs,css,Normal\ s\rangle \Rightarrow t$
  $\Longrightarrow$
  $\Gamma\vdash\langle[],(nrms,abrs)\#css,Abrupt\ s\rangle \Rightarrow t$

| *ExitBlockFault*: $\Gamma\vdash\langle nrms,css,Fault\ f\rangle \Rightarrow t$
  $\Longrightarrow$
  $\Gamma\vdash\langle[],(nrms,abrs)\#css,Fault\ f\rangle \Rightarrow t$

| *ExitBlockStuck*: $\Gamma\vdash\langle nrms,css,Stuck\rangle \Rightarrow t$
  $\Longrightarrow$
  $\Gamma\vdash\langle[],(nrms,abrs)\#css,Stuck\rangle \Rightarrow t$

| *Cons*: $[\![\Gamma\vdash\langle c,s\rangle \Rightarrow t;\ \Gamma\vdash\langle cs,css,t\rangle \Rightarrow u]\!]$
  $\Longrightarrow$
  $\Gamma\vdash\langle c\#cs,css,s\rangle \Rightarrow u$

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

$\Gamma\vdash\langle[],css,s\rangle \Rightarrow t$
$\Gamma\vdash\langle c\#cs,css,s\rangle \Rightarrow t$

⟨*ML*⟩

lemma *execs-Fault-end*:
assumes *execs*: $\Gamma\vdash\langle cs,css,s\rangle \Rightarrow t$ **shows** $s{=}Fault\ f \Longrightarrow t{=}Fault\ f$
⟨*proof*⟩

lemma *execs-Stuck-end*:
assumes *execs*: $\Gamma\vdash\langle cs,css,s\rangle \Rightarrow t$ **shows** $s{=}Stuck \Longrightarrow t{=}Stuck$
⟨*proof*⟩

theorem *steps-impl-execs*:
assumes *steps*: $\Gamma\vdash(cs,css,s) \rightarrow^* ([],[],t)$
shows $\Gamma\vdash\langle cs,css,s\rangle \Rightarrow t$
⟨*proof*⟩

theorem *steps-impl-exec*:
assumes *steps*: $\Gamma\vdash([c],[],s) \rightarrow^* ([],[],t)$
shows $\Gamma\vdash\langle c,s\rangle \Rightarrow t$
⟨*proof*⟩

corollary *steps-eq-exec*: $\Gamma\vdash([c],[],s) \rightarrow^* ([],[],t) = \Gamma\vdash\langle c,s\rangle \Rightarrow t$
⟨*proof*⟩

14.2 Infinite Computations: *inf* Γ *cs css s*

definition *inf* ::
$[('s,'p,'f)\ body,('s,'p,'f)\ com\ list,('s,'p,'f)\ continuation\ list,('s,'f)\ xstate]$
$\Rightarrow bool$
where $inf\ \Gamma\ cs\ css\ s = (\exists f.\ f\ 0 = (cs,css,s) \wedge (\forall i.\ \Gamma\vdash f\ i \rightarrow f(Suc\ i)))$

lemma *not-infI*: $[\![\bigwedge f.\ [\![f\ 0 = (cs,css,s);\ \bigwedge i.\ \Gamma\vdash f\ i \rightarrow f\ (Suc\ i)]\!] \Longrightarrow False]\!]$
$\Longrightarrow \neg inf\ \Gamma\ cs\ css\ s$
⟨*proof*⟩

14.3 Equivalence of Termination and Absence of Infinite Computations

inductive *terminatess*:: $[('s,'p,'f)\ body,('s,'p,'f)\ com\ list,$
$('s,'p,'f)\ continuation\ list,('s,'f)\ xstate] \Rightarrow bool$
(‹-⊢-,- ⇓ -› [60,20,60] 89)
for Γ::$('s,'p,'f)\ body$
where
Nil: $\Gamma\vdash[],[]\Downarrow s$

| *ExitBlockNormal*: $\Gamma\vdash nrms,css\Downarrow Normal\ s$

$\Longrightarrow$
$\Gamma\vdash[],(nrms,abrs)\#css\Downarrow Normal\ s$

| *ExitBlockAbrupt*: $\Gamma\vdash abrs,css\Downarrow Normal\ s$
$\Longrightarrow$
$\Gamma\vdash[],(nrms,abrs)\#css\Downarrow Abrupt\ s$

| *ExitBlockFault*: $\Gamma\vdash nrms,css\Downarrow Fault\ f$
$\Longrightarrow$
$\Gamma\vdash[],(nrms,abrs)\#css\Downarrow Fault\ f$

| *ExitBlockStuck*: $\Gamma\vdash nrms,css\Downarrow Stuck$
$\Longrightarrow$
$\Gamma\vdash[],(nrms,abrs)\#css\Downarrow Stuck$

| *Cons*: $[\![\Gamma\vdash c\downarrow s;\ (\forall\, t.\ \Gamma\vdash\langle c,s\rangle \Rightarrow\ t \longrightarrow \Gamma\vdash cs,css\Downarrow t)]\!]$
$\Longrightarrow$
$\Gamma\vdash c\#cs,css\Downarrow s$

inductive-cases *terminatess-elim-cases* [*cases set*]:
$\Gamma\vdash[],css\Downarrow t$
$\Gamma\vdash c\#cs,css\Downarrow t$

lemma *terminatess-Fault*: $\bigwedge cs.\ \Gamma\vdash cs,css\Downarrow Fault\ f$
⟨*proof*⟩

lemma *terminatess-Stuck*: $\bigwedge cs.\ \Gamma\vdash cs,css\Downarrow Stuck$
⟨*proof*⟩

lemma *Basic-terminates*: $\Gamma\vdash Basic\ f \downarrow t$
⟨*proof*⟩

lemma *step-preserves-terminations*:
assumes *step*: $\Gamma\vdash(cs,css,s) \rightarrow (cs',css',t)$
shows $\Gamma\vdash cs,css\Downarrow s \Longrightarrow \Gamma\vdash cs',css'\Downarrow t$
⟨*proof*⟩

⟨*ML*⟩

lemma *steps-preserves-terminations*:
assumes *steps*: $\Gamma\vdash(cs,css,s) \rightarrow^{*} (cs',css',t)$
shows $\Gamma\vdash cs,css\Downarrow s \Longrightarrow \Gamma\vdash cs',css'\Downarrow t$
⟨*proof*⟩

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 $b = f\ 0 \Longrightarrow (\exists f.\ f\ 0 = a \wedge (\forall i.\ (f\ i,\ f(Suc\ i)) \in r))$
⟨*proof*⟩

lemma *renumber*:
$\forall i.\ (a,f\ i) \in r^* \wedge (f\ i,f(Suc\ i)) \in r$
$\Longrightarrow \exists f.\ f\ 0 = a \wedge (\forall i.\ (f\ i,\ f(Suc\ i)) \in r)$
⟨*proof*⟩

lemma *not-inf-Fault'*:
assumes *enum-step*: $\forall i.\ \Gamma\vdash f\ i \rightarrow f\ (Suc\ i)$
shows $\bigwedge k\ cs.\ f\ k = (cs,css,Fault\ m) \Longrightarrow False$
⟨*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 $\bigwedge k\ cs.\ f\ k = (cs,css,Stuck) \Longrightarrow False$
⟨*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*
⟨*proof*⟩

locale *inf* =
fixes *CS*:: ('*s*,'*p*,'*f*) *config* ⇒ ('*s*, '*p*,'*f*) *com list*
and *CSS*:: ('*s*,'*p*,'*f*) *config* ⇒ ('*s*, '*p*,'*f*) *continuation list*
and *S*:: ('*s*,'*p*,'*f*) *config* ⇒ ('*s*,'*f*) *xstate*
defines *CS-def* : *CS* ≡ *fst*
defines *CSS-def* : *CSS* ≡ λ*c*. *fst* (*snd c*)
defines *S-def*: *S* ≡ λ*c*. *snd* (*snd c*)

lemma (**in** *inf*) *steps-hd-drop-suffix*:
assumes *f-0*: *f 0* = (*c*#*cs*,*css*,*s*)
assumes *f-step*: ∀ *i*. Γ⊢ *f*(*i*) → *f*(*Suc i*)
assumes *not-finished*: ∀ *i* < *k*. ¬ (*CS* (*f i*) = *cs* ∧ *CSS* (*f i*) = *css*)
assumes *simul*: ∀ *i*≤*k*.
(*if pcss i* = [] *then CSS* (*f i*)=*css* ∧ *CS* (*f i*)=*pcs i*@*cs*
else CS (*f i*)=*pcs i* ∧
CSS (*f i*)= *butlast* (*pcss i*)@
[(*fst* (*last* (*pcss i*))@*cs*,(*snd* (*last* (*pcss i*)))@*cs*)]@
css)
defines *p*≡λ*i*. (*pcs i*, *pcss i*, *S* (*f i*))
shows ∀ *i*<*k*. Γ⊢ *p i* → *p* (*Suc i*)
⟨*proof*⟩

lemma *k-steps-to-rtrancl*:
assumes *steps*: ∀ *i*<*k*. Γ⊢ *p i* → *p* (*Suc i*)
shows Γ⊢*p 0*→* *p k*
⟨*proof*⟩

lemma (**in** *inf*) *steps-hd-drop-suffix-finite*:
assumes *f-0*: *f 0* = (*c*#*cs*,*css*,*s*)
assumes *f-step*: ∀ *i*. Γ⊢ *f*(*i*) → *f*(*Suc i*)
assumes *not-finished*: ∀ *i* < *k*. ¬ (*CS* (*f i*) = *cs* ∧ *CSS* (*f i*) = *css*)
assumes *simul*: ∀ *i*≤*k*.
(*if pcss i* = [] *then CSS* (*f i*)=*css* ∧ *CS* (*f i*)=*pcs i*@*cs*
else CS (*f i*)=*pcs i* ∧
CSS (*f i*)= *butlast* (*pcss i*)@
[(*fst* (*last* (*pcss i*))@*cs*,(*snd* (*last* (*pcss i*)))@*cs*)]@
css)
shows Γ⊢([*c*],[],*s*) →* (*pcs k*, *pcss k*, *S* (*f k*))
⟨*proof*⟩

lemma (**in** *inf*) *steps-hd-drop-suffix-infinite*:
assumes *f-0*: *f 0* = (*c*#*cs*,*css*,*s*)
assumes *f-step*: ∀ *i*. Γ⊢ *f*(*i*) → *f*(*Suc i*)
assumes *not-finished*: ∀ *i*. ¬ (*CS* (*f i*) = *cs* ∧ *CSS* (*f i*) = *css*)

assumes *simul*: ∀ *i*.
    (*if pcss i* = [] *then CSS* (*f i*)=*css* ∧ *CS* (*f i*)=*pcs i*@*cs*
        else CS (*f i*)=*pcs i* ∧
            CSS (*f i*)= *butlast* (*pcss i*)@
                [(*fst* (*last* (*pcss i*))@*cs*,(*snd* (*last* (*pcss i*)))@*cs*)]@
                css)
defines *p*≡λ*i*. (*pcs i*, *pcss i*, *S* (*f i*))
shows Γ⊢ *p i* → *p* (*Suc i*)
⟨*proof*⟩

lemma (**in** *inf*) *steps-hd-progress*:
assumes *f-0*: *f 0* = (*c*#*cs*,*css*,*s*)
assumes *f-step*: ∀ *i*. Γ⊢ *f*(*i*) → *f*(*Suc i*)
assumes *c-unfinished*: ∀ *i* < *k*. ¬ (*CS* (*f i*) = *cs* ∧ *CSS* (*f i*) = *css*)
shows ∀ *i* ≤ *k*. (∃ *pcs pcss*.
    (*if pcss* = [] *then CSS* (*f i*)=*css* ∧ *CS* (*f i*)=*pcs*@*cs*
     else CS (*f i*)=*pcs* ∧
        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*: ∀ *i*. Γ⊢ *f*(*i*) → *f*(*Suc i*)
assumes *unfinished*: ∀ *i*. ¬ ((*CS* (*f i*) = *cs*) ∧ (*CSS* (*f i*) = *css*))
shows ∃ *pcs pcss*.
    (*if pcss* = [] *then CSS* (*f i*)=*css* ∧ *CS* (*f i*)=*pcs*@*cs*
     else CS (*f i*)=*pcs* ∧
        CSS (*f i*)= *butlast pcss*@
            [(*fst* (*last pcss*)@*cs*,(*snd* (*last pcss*))@*cs*)]@
            css)
⟨*proof*⟩

lemma *skolemize1*: ∀ *x*. *P x* ⟶ (∃ *y*. *Q x y*) ⟹ ∃ *f*.∀ *x*. *P x* ⟶ *Q x* (*f x*)
  ⟨*proof*⟩

lemma *skolemize2*: ∀ *x*. *P x* ⟶ (∃ *y z*. *Q x y z*) ⟹ ∃ *f g*.∀ *x*. *P x* ⟶ *Q x* (*f x*)
(*g x*)
⟨*proof*⟩

lemma *skolemize2′*: ∀ *x*.∃ *y z*. *P x y z* ⟹ ∃ *f g*.∀ *x*. *P x* (*f x*) (*g x*)
⟨*proof*⟩

theorem (**in** *inf*) *inf-cases*:
  fixes *c*::(′*s*,′*p*,′*f*) *com*
  assumes *inf*: *inf* Γ (*c*#*cs*) *css s*
  shows *inf* Γ [*c*] [] *s* ∨ (∃ *t*. Γ⊢⟨*c*,*s*⟩ ⇒ *t* ∧ *inf* Γ *cs css t*)

$\langle proof \rangle$

lemma *infE* [*consumes 1*]:
assumes *inf*: *inf* Γ $(c\#cs)$ *css s*
assumes *cases*: *inf* Γ $[c]$ $[]$ $s \Longrightarrow P$
$\bigwedge t.\ [\![\Gamma\vdash\langle c,s\rangle \Rightarrow t;\ inf\ \Gamma\ cs\ css\ t]\!] \Longrightarrow P$
shows *P*
$\langle proof \rangle$

lemma *inf-Seq*:
$inf\ \Gamma\ (Seq\ c1\ c2\#cs)\ css\ (Normal\ s) = inf\ \Gamma\ (c1\#c2\#cs)\ css\ (Normal\ s)$
$\langle proof \rangle$

lemma *inf-WhileTrue*:
assumes *b*: $s \in b$
shows $inf\ \Gamma\ (While\ b\ c\#cs)\ css\ (Normal\ s) =$
$inf\ \Gamma\ (c\#While\ b\ c\#cs)\ css\ (Normal\ s)$
$\langle proof \rangle$

lemma *inf-Catch*:
$inf\ \Gamma\ (Catch\ c1\ c2\#cs)\ css\ (Normal\ s) = inf\ \Gamma\ [c1]\ ((cs,c2\#cs)\#css)\ (Normal\ s)$
$\langle proof \rangle$

theorem *terminates-impl-not-inf*:
assumes *termi*: $\Gamma\vdash c \downarrow s$
shows $\neg inf\ \Gamma\ [c]\ []\ s$
$\langle proof \rangle$

lemma *terminatess-impl-not-inf*:
assumes *termi*: $\Gamma\vdash cs,css\Downarrow s$
shows $\neg inf\ \Gamma\ cs\ css\ s$
$\langle proof \rangle$

lemma *lem*:
$\forall y.\ r^{++}\ a\ y \longrightarrow P\ a \longrightarrow P\ y$
$\Longrightarrow ((b,a) \in \{(y,x).\ P\ x \wedge r\ x\ y\}^{+}) = ((b,a) \in \{(y,x).\ P\ x \wedge r^{++}\ x\ y\})$
$\langle proof \rangle$

corollary *terminatess-impl-no-inf-chain*:
assumes *terminatess*: $\Gamma\vdash cs,css\Downarrow s$
shows $\neg(\exists f.\ f\ 0 = (cs,css,s) \wedge (\forall i{::}nat.\ \Gamma\vdash f\ i \rightarrow^{+} f(Suc\ i)))$
$\langle proof \rangle$

corollary *terminates-impl-no-inf-chain*:
$\Gamma\vdash c\downarrow s \Longrightarrow \neg(\exists f.\ f\ 0 = ([c],[],s) \wedge (\forall i{::}nat.\ \Gamma\vdash f\ i \rightarrow^{+} f(Suc\ i)))$
$\langle proof \rangle$

definition

termi-call-steps :: $('s,'p,'f)$ *body* $\Rightarrow$ $(('s \times 'p) \times ('s \times 'p))set$
where
termi-call-steps Γ =
$\{((t,q),(s,p)).\ \Gamma\vdash the\ (\Gamma\ p)\downarrow Normal\ s\ \wedge$
$(\exists\ css.\ \Gamma\vdash([the\ (\Gamma\ p)],[],Normal\ s) \rightarrow^{+} ([the\ (\Gamma\ q)],css,Normal\ t))\}$

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* Γ)
⟨*proof*⟩

An alternative proof using Hilbert-choice instead of axiom of choice.

theorem *wf* (*termi-call-steps* Γ)
⟨*proof*⟩

lemma *not-inf-implies-wf*: **assumes** *not-inf*: $\neg$ *inf* Γ *cs css s*
shows *wf* $\{(c2,c1).\ \Gamma \vdash (cs,css,s) \rightarrow^{*} c1 \wedge \Gamma \vdash c1 \rightarrow c2\}$
⟨*proof*⟩

lemma *wf-implies-termi-reach*:
assumes *wf*: *wf* $\{(c2,c1).\ \Gamma \vdash (cs,css,s) \rightarrow^{*} c1 \wedge \Gamma \vdash c1 \rightarrow c2\}$
shows $\bigwedge cs1\ css1\ s1.\ [\![\Gamma \vdash (cs,css,s) \rightarrow^{*} c1;\ c1=(cs1,css1,s1)]\!] \Longrightarrow \Gamma\vdash cs1,css1\Downarrow s1$
⟨*proof*⟩

lemma *not-inf-impl-terminatess*:
assumes *not-inf*: $\neg$ *inf* Γ *cs css s*
shows $\Gamma\vdash cs,css\Downarrow s$
⟨*proof*⟩

lemma *not-inf-impl-terminates*:
assumes *not-inf*: $\neg$ *inf* Γ [*c*] [] *s*
shows $\Gamma\vdash c\downarrow s$
⟨*proof*⟩

theorem *terminatess-iff-not-inf*:
$\Gamma\vdash cs,css\Downarrow s = (\neg\ inf\ \Gamma\ cs\ css\ s)$
⟨*proof*⟩

corollary *terminates-iff-not-inf*:
$\Gamma\vdash c\downarrow s = (\neg\ inf\ \Gamma\ [c]\ []\ s)$
⟨*proof*⟩

14.4 Completeness of Total Correctness Hoare Logic

lemma *ConseqMGT*:
 assumes *modif*: $\forall Z::'a.\ \Gamma,\Theta \vdash_{t/F} (P'\ Z::'a\ assn)\ c\ (Q'\ Z),(A'\ Z)$
 assumes *impl*: $\bigwedge s.\ s \in P \Longrightarrow s \in P'\ s \wedge (\forall t.\ t \in Q'\ s \longrightarrow t \in Q) \wedge$
$(\forall t.\ t \in A'\ s \longrightarrow t \in A)$
 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*: $R \Longrightarrow \Gamma,\Theta\vdash_{t/F} P\ c\ Q,A$
 shows $\Gamma,\Theta\vdash_{t/F} P\ c\ Q,A$
 ⟨*proof*⟩

lemma *Call-lemma'*:
 assumes *Call-hyp*:
 $\forall q{\in}dom\ \Gamma.\ \forall Z.\ \Gamma,\Theta\vdash_{t/F}\{s.\ s{=}Z \wedge \Gamma\vdash\langle Call\ q,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F)) \wedge$
$\Gamma\vdash Call\ q{\downarrow}Normal\ s \wedge ((s,q),(\sigma,p)) \in termi\text{-}call\text{-}steps\ \Gamma\}$
$(Call\ q)$
$\{t.\ \Gamma\vdash\langle Call\ q,Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma\vdash\langle Call\ q,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
 shows $\bigwedge Z.\ \Gamma,\Theta \vdash_{t/F}$
$\{s.\ s{=}Z \wedge \Gamma\vdash\langle c,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F)) \wedge \Gamma\vdash the\ (\Gamma\ p){\downarrow}Normal\ \sigma \wedge$
$(\exists\ cs\ css.\ \Gamma\vdash([the\ (\Gamma\ p)],[],Normal\ \sigma) \rightarrow^*\ (c\#cs,css,Normal\ s))\}$
c
$\{t.\ \Gamma\vdash\langle c,Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma\vdash\langle c,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
⟨*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: $\forall q \in dom\ \Gamma.\ \forall Z.\ \Gamma,\Theta \vdash_{t/F}$
$\{s.\ s{=}Z \wedge \Gamma\vdash\langle Call\ q,Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F)) \wedge$
$\Gamma\vdash Call\ q{\downarrow}Normal\ s \wedge ((s,q),(\sigma,p)) \in termi\text{-}call\text{-}steps\ \Gamma\}$
$(Call\ q)$
$\{t.\ \Gamma\vdash\langle Call\ q,Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma\vdash\langle Call\ q,Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
 shows $\bigwedge Z.\ \Gamma,\Theta \vdash_{t/F}$
$(\{\sigma\} \cap \{s.\ s{=}Z \wedge \Gamma\vdash\langle the\ (\Gamma\ p),Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F)) \wedge$
$\Gamma\vdash the\ (\Gamma\ p){\downarrow}Normal\ s\})$
$the\ (\Gamma\ p)$

$\{t.\ \Gamma\vdash\langle the\ (\Gamma\ p), Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma\vdash\langle the\ (\Gamma\ p), Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
⟨*proof*⟩

lemma *Call-lemma-switch-Call-body*:
assumes
call: $\forall q \in dom\ \Gamma.\ \forall Z.\ \Gamma,\Theta \vdash_{t/F}$
$\{s.\ s=Z \wedge \Gamma\vdash\langle Call\ q, Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F)) \wedge$
$\Gamma\vdash Call\ q\downarrow Normal\ s \wedge ((s,q),(\sigma,p)) \in termi\text{-}call\text{-}steps\ \Gamma\}$
$(Call\ q)$
$\{t.\ \Gamma\vdash\langle Call\ q, Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma\vdash\langle Call\ q, Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
assumes *p-defined*: $p \in dom\ \Gamma$
shows $\bigwedge Z.\ \Gamma,\Theta \vdash_{t/F}$
$(\{\sigma\} \cap \{s.\ s=Z \wedge \Gamma\vdash\langle Call\ p, Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F)) \wedge$
$\Gamma\vdash Call\ p\downarrow Normal\ s\})$
$the\ (\Gamma\ p)$
$\{t.\ \Gamma\vdash\langle Call\ p, Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma\vdash\langle Call\ p, Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
⟨*proof*⟩

lemma *MGT-Call*:
$\forall p \in dom\ \Gamma.\ \forall Z.$
$\Gamma,\Theta \vdash_{t/F} \{s.\ s=Z \wedge \Gamma\vdash\langle Call\ p, Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F)) \wedge$
$\Gamma\vdash(Call\ p)\downarrow Normal\ s\}$
$(Call\ p)$
$\{t.\ \Gamma\vdash\langle Call\ p, Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma\vdash\langle Call\ p, Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
⟨*proof*⟩

lemma *CollInt-iff*: $\{s.\ P\ s\} \cap \{s.\ Q\ s\} = \{s.\ P\ s \wedge Q\ s\}$
⟨*proof*⟩

lemma *image-Un-conv*: $f\ `\ (\bigcup p{\in}dom\ \Gamma.\ \bigcup Z.\ \{x\ p\ Z\}) = (\bigcup p{\in}dom\ \Gamma.\ \bigcup Z.\ \{f\ (x\ p\ Z)\})$
⟨*proof*⟩

Another proof of *MGT-Call*, maybe a little more readable

lemma
$\forall p \in dom\ \Gamma.\ \forall Z.$
$\Gamma,\{\} \vdash_{t/F} \{s.\ s=Z \wedge \Gamma\vdash\langle Call\ p, Normal\ s\rangle \Rightarrow\notin(\{Stuck\} \cup Fault\ `\ (-F)) \wedge$
$\Gamma\vdash(Call\ p)\downarrow Normal\ s\}$
$(Call\ p)$
$\{t.\ \Gamma\vdash\langle Call\ p, Normal\ Z\rangle \Rightarrow Normal\ t\}$,
$\{t.\ \Gamma\vdash\langle Call\ p, Normal\ Z\rangle \Rightarrow Abrupt\ t\}$
⟨*proof*⟩

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 ∃ *m*. ∀ *n*∈*N*. *n* < *m*
⟨*proof*⟩

lemma *infinite-nat*: ¬*finite* (*UNIV*::*nat set*)
⟨*proof*⟩

lemma *infinite-ref* [*simp*,*intro*]: ¬*finite* (*UNIV*::*ref set*)
⟨*proof*⟩

consts *Null* :: *ref*

definition *new* :: *ref set* ⇒ *ref* **where**
  new A = (*SOME a*. *a* ∉ {*Null*} ∪ *A*)

Constant *Null* can be defined later on. Conceptually *Null* and *new* are *fixes* of a locale with *finite A* ⟹ *new A* ∉ *A* ∪ {*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 ⟹ *new* (*A*) ∉ *A*
  ⟨*proof*⟩

lemma *new-not-Null* [*simp*,*intro*]:
  finite A ⟹ *new* (*A*) ≠ *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
$Path\ x\ h\ y\ [] = (x = y)\ |$
$Path\ x\ h\ y\ (p\#ps) = (x = p \wedge x \neq Null \wedge Path\ (h\ x)\ h\ y\ ps)$

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

lemma *Path-not-Null-iff* [*simp*]: $p{\neq}Null \Longrightarrow$
$Path\ p\ h\ q\ as = (as = [] \wedge q = p\ \vee\ (\exists\, ps.\ as = p\#ps \wedge Path\ (h\ p)\ h\ q\ ps\ ))$
⟨*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*]:
$\bigwedge p.\ u \notin set\ ps \Longrightarrow Path\ p\ (f(u := v))\ q\ ps\ = Path\ p\ f\ q\ ps$
⟨*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 $p \neq Null \Longrightarrow Path\ p\ (f(p := p))\ q\ qs = X$ from looping, because of *Path-not-Null-iff* and *fun-upd-apply*.

lemma *notin-Path-updateI* [*intro*]:
$[\![Path\ p\ h\ q\ ps\ ;\ r \notin set\ ps]\!] \Longrightarrow Path\ p\ (h(r := y))\ q\ ps$
⟨*proof*⟩

lemma *Path-update-new* [*simp*]: $[\![set\ ps \subseteq set\ alloc]\!]$
$\Longrightarrow Path\ p\ (f(new\ (set\ alloc) := x))\ q\ ps\ = Path\ p\ f\ q\ ps$
⟨*proof*⟩

lemma *Null-notin-Path* [*simp,intro*]:
$\bigwedge p.\ Path\ p\ f\ q\ ps \Longrightarrow Null \notin set\ ps$
⟨*proof*⟩

lemma *Path-snoc*:
$[\![Path\ p\ (f(a := q))\ a\ as\ ;\ a{\neq}Null]\!] \Longrightarrow Path\ p\ (f(a := q))\ q\ (as\ @\ [a])$

$\langle proof \rangle$

15.2 Lists on The Heap

15.2.1 Relational Abstraction

definition
List :: *ref* ⇒ (*ref* ⇒ *ref*) ⇒ *ref list* ⇒ *bool* **where**
List p h ps = *Path p h Null ps*

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

lemma *List-cons* [*simp*]: *List p h* (*a*#*ps*) = (*p* = *a* ∧ *p*≠*Null* ∧ *List* (*h p*) *h ps*)
$\langle proof \rangle$

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

lemma *List-not-Null* [*simp*]: *p*≠*Null* ⟹
List p h as = (∃ *ps*. *as* = *p*#*ps* ∧ *List* (*h p*) *h ps*)
$\langle proof \rangle$

lemma *Null-notin-List* [*simp*,*intro*]: ⋀*p*. *List p h ps* ⟹ *Null* ∉ *set ps*
$\langle proof \rangle$

theorem *notin-List-update*[*simp*]:
⋀*p*. *q* ∉ *set ps* ⟹ *List p* (*h*(*q* := *y*)) *ps* = *List p h ps*
$\langle proof \rangle$

lemma *List-upd-same-lemma*: ⋀*p*. *p* ≠ *Null* ⟹ ¬ *List p* (*h*(*p* := *p*)) *ps*
$\langle proof \rangle$

lemma *List-upd-same* [*simp*]: *List p* (*h*(*p*:=*p*)) *ps* = (*p* = *Null* ∧ *ps* = [])
$\langle proof \rangle$

List-upd-same prevents *p* ≠ *Null* ⟹ *List p* (*h*(*p* := *p*)) *as* = *X* from looping, because of *List-not-Null* and *fun-upd-apply*.

lemma *List-update-new* [*simp*]: ⟦*set ps* ⊆ *set alloc*⟧
⟹ *List p* (*h*(*new* (*set alloc*) := *x*)) *ps* = *List p h ps*
$\langle proof \rangle$

lemma *List-updateI* [*intro*]:
⟦*List p h ps*; *q* ∉ *set ps*⟧ ⟹ *List p* (*h*(*q* := *y*)) *ps*
$\langle proof \rangle$

lemma *List-unique*: ⋀*p bs*. *List p h as* ⟹ *List p h bs* ⟹ *as* = *bs*
$\langle proof \rangle$

lemma *List-unique1*: $List\ p\ h\ as \Longrightarrow \exists!as.\ List\ p\ h\ as$
⟨*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*]: $List\ (h\ p)\ h\ ps \Longrightarrow p \notin set\ ps$
⟨*proof*⟩

lemma *List-distinct*[*simp*]: $\bigwedge p.\ List\ p\ h\ ps \Longrightarrow distinct\ ps$
⟨*proof*⟩

lemma *heap-eq-List-eq*:
$\bigwedge p.\ \forall\, x \in set\ ps.\ h\ x = g\ x \Longrightarrow List\ p\ h\ ps = List\ p\ g\ ps$
⟨*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-ListI1*:
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 usefull 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*]:
$List\ p\ h\ ps \Longrightarrow (\forall\, ps.\ List\ p\ h\ ps \longrightarrow P\ ps) = P\ ps$
⟨*proof*⟩

lemma *List-unique-ex-conj-simp* [*simp*]:
$List\ p\ h\ ps \Longrightarrow (\exists\, ps.\ List\ p\ h\ ps \wedge P\ ps) = P\ ps$
⟨*proof*⟩

15.3 Functional abstraction

definition

islist :: *ref* ⇒ (*ref* ⇒ *ref*) ⇒ *bool* **where**
islist p h = (∃ *ps*. *List p h ps*)

definition
list :: *ref* ⇒ (*ref* ⇒ *ref*) ⇒ *ref list* **where**
list p h = (*THE ps*. *List p h ps*)

lemma *List-conv-islist-list*: *List p h ps* = (*islist p h* ∧ *ps* = *list p h*)
⟨*proof*⟩

lemma *List-islist* [*intro*]:
List p h ps ⟹ *islist p h*
⟨*proof*⟩

lemma *List-list*:
List p h ps ⟹ *list p h* = *ps*
⟨*proof*⟩

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

lemma [*simp*]: *p*≠*Null* ⟹ *islist* (*h p*) *h* = *islist p h*
⟨*proof*⟩

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

lemma *list-Ref-conv*[*simp*]:
⟦*islist* (*h p*) *h*; *p*≠*Null* ⟧ ⟹ *list p h* = *p* # *list* (*h p*) *h*
⟨*proof*⟩

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

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

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

lemma *list-distinct*[*simp*]: *islist p h* ⟹ *distinct* (*list p h*)
⟨*proof*⟩

lemma *Null-notin-list* [*simp*,*intro*]: *islist p h* ⟹ *Null* ∉ *set* (*list p h*)
⟨*proof*⟩

end

theory *Generalise* **imports** *HOL−Statespace.DistinctTreeProver*
begin

lemma *protectRefl*: *PROP Pure.prop* (*PROP C*) $\Longrightarrow$ *PROP Pure.prop* (*PROP C*)
⟨*proof*⟩

lemma *protectImp*:
assumes *i*: *PROP Pure.prop* (*PROP P* $\Longrightarrow$ *PROP Q*)
shows *PROP Pure.prop* (*PROP Pure.prop P* $\Longrightarrow$ *PROP Pure.prop Q*)
⟨*proof*⟩

lemma *generaliseConj*:
assumes *i1*: *PROP Pure.prop* (*PROP Pure.prop* (*Trueprop P*) $\Longrightarrow$ *PROP Pure.prop* (*Trueprop Q*))
assumes *i2*: *PROP Pure.prop* (*PROP Pure.prop* (*Trueprop P′*) $\Longrightarrow$ *PROP Pure.prop* (*Trueprop Q′*))
shows *PROP Pure.prop* (*PROP Pure.prop* (*Trueprop* (*P* ∧ *P′*)) $\Longrightarrow$ (*PROP Pure.prop* (*Trueprop* (*Q* ∧ *Q′*))))
⟨*proof*⟩

lemma *generaliseAll*:
assumes *i*: *PROP Pure.prop* (⋀*s*. *PROP Pure.prop* (*Trueprop* (*P s*)) $\Longrightarrow$ *PROP Pure.prop* (*Trueprop* (*Q s*)))
shows *PROP Pure.prop* (*PROP Pure.prop* (*Trueprop* (∀ *s*. *P s*)) $\Longrightarrow$ *PROP Pure.prop* (*Trueprop* (∀ *s*. *Q s*)))
⟨*proof*⟩

lemma *generalise-all*:
assumes *i*: *PROP Pure.prop* (⋀*s*. *PROP Pure.prop* (*PROP P s*) $\Longrightarrow$ *PROP Pure.prop* (*PROP Q s*))
shows *PROP Pure.prop* ((*PROP Pure.prop* (⋀*s*. *PROP P s*)) $\Longrightarrow$ (*PROP Pure.prop* (⋀*s*. *PROP Q s*)))
⟨*proof*⟩

lemma *generaliseTrans*:
assumes *i1*: *PROP Pure.prop* (*PROP P* $\Longrightarrow$ *PROP Q*)
assumes *i2*: *PROP Pure.prop* (*PROP Q* $\Longrightarrow$ *PROP R*)
shows *PROP Pure.prop* (*PROP P* $\Longrightarrow$ *PROP R*)
⟨*proof*⟩

lemma *meta-spec*:
assumes ⋀*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*)
⟨*proof*⟩

lemma *generaliseImp*:
assumes *i*: *PROP Pure.prop* (*PROP Pure.prop* (*Trueprop P*) $\Longrightarrow$ *PROP Pure.prop* (*Trueprop Q*))
shows *PROP Pure.prop* (*PROP Pure.prop* (*Trueprop* ($X \longrightarrow P$)) $\Longrightarrow$ *PROP Pure.prop* (*Trueprop* ($X \longrightarrow Q$)))
⟨*proof*⟩

lemma *generaliseEx*:
assumes *i*: *PROP Pure.prop* ($\bigwedge s$. *PROP Pure.prop* (*Trueprop* (*P s*)) $\Longrightarrow$ *PROP Pure.prop* (*Trueprop* (*Q s*)))
shows *PROP Pure.prop* (*PROP Pure.prop* (*Trueprop* ($\exists s.\ P\ s$)) $\Longrightarrow$ *PROP Pure.prop* (*Trueprop* ($\exists s.\ Q\ s$)))
⟨*proof*⟩

lemma *generaliseRefl*: *PROP Pure.prop* (*PROP Pure.prop* (*Trueprop P*) $\Longrightarrow$ *PROP Pure.prop* (*Trueprop P*))
⟨*proof*⟩

lemma *generaliseRefl'*: *PROP Pure.prop* (*PROP P* $\Longrightarrow$ *PROP P*)
⟨*proof*⟩

lemma *generaliseAllShift*:
assumes *i*: *PROP Pure.prop* ($\bigwedge s.\ P \Longrightarrow Q\ s$)
shows *PROP Pure.prop* (*PROP Pure.prop* (*Trueprop P*) $\Longrightarrow$ *PROP Pure.prop* (*Trueprop* ($\forall s.\ Q\ s$)))
⟨*proof*⟩

lemma *generalise-allShift*:
assumes *i*: *PROP Pure.prop* ($\bigwedge s$. *PROP P* $\Longrightarrow$ *PROP Q s*)
shows *PROP Pure.prop* (*PROP Pure.prop* (*PROP P*) $\Longrightarrow$ *PROP Pure.prop* ($\bigwedge s$. *PROP Q s*))
⟨*proof*⟩

lemma *generaliseImpl*:
assumes *i*: *PROP Pure.prop* (*PROP Pure.prop P* $\Longrightarrow$ *PROP Pure.prop Q*)
shows *PROP Pure.prop* ((*PROP Pure.prop* (*PROP X* $\Longrightarrow$ *PROP P*)) $\Longrightarrow$ (*PROP Pure.prop* (*PROP X* $\Longrightarrow$ *PROP Q*)))
⟨*proof*⟩

⟨*ML*⟩

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*

⟨*ML*⟩

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* ⇒ *nat list* ⇒ *′a list* (**infixl** ‹!!› *100*)
  where *xs* !! *ns* = *map* (*nth xs*) *ns*

definition *list-multupd*:: *′a list* ⇒ *nat list* ⇒ *′a list* ⇒ *′a list*
  where *list-multupd xs ns ys* = *foldl* (λ*xs* (*n*,*v*). *xs*[*n*:=*v*]) *xs* (*zip ns ys*)

nonterminal *lmupdbinds* **and** *lmupdbind*

syntax
  — multiple list update
  -lmupdbind:: [*′a*, *′a*] => *lmupdbind*   (‹(*2*- [:=]/ -)›)
  :: *lmupdbind* => *lmupdbinds*   (‹-›)
  -lmupdbinds :: [*lmupdbind*, *lmupdbinds*] => *lmupdbinds*   (‹-,/ -›)
  -LMUpdate :: [*′a*, *lmupdbinds*] => *′a*   (‹-/[(-)]› [*900*,*0*] *900*)

syntax-consts
  -lmupdbind -lmupdbinds -LMUpdate == *list-multupd*

translations
  -LMUpdate xs (*-lmupdbinds 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* ⇒ (*′a* ⇒ *′b*) ⇒ *′b* (**infixr** ‹|>› *60*)
  where *rapp x f* = *f x*

nonterminal
  newinit **and**

newinits **and**
locinit **and**
locinits **and**
switchcase **and**
switchcases **and**
grds **and**
grd **and**
bdy **and**
basics **and**
basic **and**
basicblock

notation
Skip (‹*SKIP*›) **and**
Throw (‹*THROW*›)

syntax
-raise:: *'c* ⇒ *'c* ⇒ (*'a*,*'b*,*'f*) *com* (‹(*RAISE* - :==/ -)› [*30*, *30*] *23*)
-seq::(*'s*,*'p*,*'f*) *com* ⇒ (*'s*,*'p*,*'f*) *com* ⇒ (*'s*,*'p*,*'f*) *com* (‹-;;/ -› [*20*, *21*] *20*)
-guarantee :: *'s set* ⇒ *grd* (‹-√› [*1000*] *1000*)
-guaranteeStrip:: *'s set* ⇒ *grd* (‹-#› [*1000*] *1000*)
-grd :: *'s set* ⇒ *grd* (‹-› [*1000*] *1000*)
-last-grd :: *grd* ⇒ *grds* (‹-› *1000*)
-grds :: [*grd*, *grds*] ⇒ *grds* (‹-,/ -› [*999*,*1000*] *1000*)
-guards :: *grds* ⇒ (*'s*,*'p*,*'f*) *com* ⇒ (*'s*,*'p*,*'f*) *com*
(‹(-/⟼ -)› [*60*, *21*] *23*)
-quote :: *'b* => (*'a* => *'b*)
-antiquoteCur0 :: (*'a* => *'b*) => *'b* (‹´-› [*1000*] *1000*)
-antiquoteCur :: (*'a* => *'b*) => *'b*
-antiquoteOld0 :: (*'a* => *'b*) => *'a* => *'b* (‹ˉ-› [*1000*,*1000*] *1000*)
-antiquoteOld :: (*'a* => *'b*) => *'a* => *'b*
-Assert :: *'a* => *'a set* (‹(⦃-⦄)› [*0*] *1000*)
-AssertState :: *idt* ⇒ *'a* => *'a set* (‹(⦃-. -⦄)› [*1000*,*0*] *1000*)
-Assign :: *'b* => *'b* => (*'a*,*'p*,*'f*) *com* (‹(- :==/ -)› [*30*, *30*] *23*)
-Init :: *ident* ⇒ *'c* ⇒ *'b* ⇒ (*'a*,*'p*,*'f*) *com*
(‹(´- :==.-/ -)› [*30*,*1000*, *30*] *23*)
-GuardedAssign:: *'b* => *'b* => (*'a*,*'p*,*'f*) *com* (‹(- :==$_g$/ -)› [*30*, *30*] *23*)
-newinit :: [*ident*,*'a*] ⇒ *newinit* (‹(*2* ´- :==/ -)›)
:: *newinit* ⇒ *newinits* (‹-›)
-newinits :: [*newinit*, *newinits*] ⇒ *newinits* (‹-,/ -›)
-New :: [*'a*, *'b*, *newinits*] ⇒ (*'a*,*'b*,*'f*) *com*
(‹(- :==/(*2 NEW* -/ [-]))› [*30*, *65*, *0*] *23*)
-GuardedNew :: [*'a*, *'b*, *newinits*] ⇒ (*'a*,*'b*,*'f*) *com*
(‹(- :==$_g$/(*2 NEW* -/ [-]))› [*30*, *65*, *0*] *23*)
-NNew :: [*'a*, *'b*, *newinits*] ⇒ (*'a*,*'b*,*'f*) *com*
(‹(- :==/(*2 NNEW* -/ [-]))› [*30*, *65*, *0*] *23*)
-GuardedNNew :: [*'a*, *'b*, *newinits*] ⇒ (*'a*,*'b*,*'f*) *com*
(‹(- :==$_g$/(*2 NNEW* -/ [-]))› [*30*, *65*, *0*] *23*)

```
-Cond          :: 'a bexp => ('a,'p,'f) com => ('a,'p,'f) com => ('a,'p,'f) com
      (‹(0IF (-)/ (2THEN/ -)/ (2ELSE -)/ FI)› [0, 0, 0] 71)
-Cond-no-else:: 'a bexp => ('a,'p,'f) com => ('a,'p,'f) com
      (‹(0IF (-)/ (2THEN/ -)/ FI)› [0, 0] 71)
-GuardedCond :: 'a bexp => ('a,'p,'f) com => ('a,'p,'f) com => ('a,'p,'f) com
      (‹(0IF_g (-)/ (2THEN -)/ (2ELSE -)/ FI)› [0, 0, 0] 71)
-GuardedCond-no-else:: 'a bexp => ('a,'p,'f) com => ('a,'p,'f) com
      (‹(0IF_g (-)/ (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 => pttrn ⇒ ('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 => pttrn ⇒ ('z ⇒ 'a assn)  ⇒ bdy
                    ⇒ ('a,'p,'f) com
      (‹(0WHILE (-)/ FIX -./ INV (-) /-)›  [25, 0, 0, 81] 71)
-GuardedWhileFix-inv-var   :: 'a bexp => pttrn ⇒ ('z ⇒ 'a assn)  ⇒
                      ('z ⇒ ('a × 'a) set) ⇒ bdy
                    ⇒ ('a,'p,'f) com
      (‹(0WHILE_g (-)/ 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 => pttrn ⇒ ('z ⇒ 'a assn)  ⇒ bdy
                    ⇒ ('a,'p,'f) com
      (‹(0WHILE_g (-)/ FIX -./ INV (-)/-)›  [25, 0, 0, 81] 71)

-GuardedWhile-inv-var::
     'a bexp => 'a assn  ⇒ ('a × 'a) set ⇒ bdy ⇒ ('a,'p,'f) com
      (‹(0WHILE_g (-)/ 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_g (-)/ 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_g (-) /-)›  [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⇒pttrn⇒('z⇒'a assn)⇒('z⇒('a×'a) set)
                       ⇒bdy ⇒ ('a,'p,'f) com
```

(‹(0WHILE (-/⟼ (1-)) FIX -./ INV (-)/ VAR (-) /-)› [1000,25,0,0,0,81] 71)
-WhileFix-guard-inv:: grds ⇒'a bexp⇒pttrn⇒('z⇒'a assn)
⇒bdy ⇒ ('a,'p,'f) com
(‹(0WHILE (-/⟼ (1-)) FIX -./ INV (-)/-)› [1000,25,0,0,81] 71)

-Try-Catch:: ('a,'p,'f) com ⇒('a,'p,'f) com ⇒ ('a,'p,'f) com
(‹(0TRY (-)/ (2CATCH -)/ END)› [0,0] 71)

-DoPre :: ('a,'p,'f) com ⇒ ('a,'p,'f) com
-Do :: ('a,'p,'f) com ⇒ bdy (‹(2DO/ (-)) /OD› [0] 1000)
-Lab:: 'a bexp ⇒ ('a,'p,'f) com ⇒ bdy
(‹-•/-› [1000,71] 81)
:: bdy ⇒ ('a,'p,'f) com (‹-›)
-Spec:: pttrn ⇒ 's set ⇒ ('s,'p,'f) com ⇒ 's set ⇒ 's set ⇒ ('s,'p,'f) com
(‹(ANNO -. -/ (-)/ -,/-)› [0,1000,20,1000,1000] 60)
-SpecNoAbrupt:: pttrn ⇒ 's set ⇒ ('s,'p,'f) com ⇒ 's set ⇒ ('s,'p,'f) com
(‹(ANNO -. -/ (-)/ -)› [0,1000,20,1000] 60)
-LemAnno:: 'n ⇒ ('s,'p,'f) com ⇒ ('s,'p,'f) com
(‹(0 LEMMA (-)/ - END)› [1000,0] 71)
-locnoinit :: ident ⇒ locinit (‹´-›)
-locinit :: [ident,'a] ⇒ locinit (‹(2´- :==/ -)›)
:: locinit ⇒ locinits (‹-›)
-locinits :: [locinit, locinits] ⇒ locinits (‹-,/ -›)
-Loc:: [locinits,('s,'p,'f) com] ⇒ ('s,'p,'f) com
(‹(2 LOC -;;/ (-) COL)› [0,0] 71)
-Switch:: ('s ⇒ 'v) ⇒ switchcases ⇒ ('s,'p,'f) com
(‹(0 SWITCH (-)/ - END)› [22,0] 71)
-switchcase:: 'v set ⇒ ('s,'p,'f) com ⇒ switchcase (‹-⇒/ -›)
-switchcasesSingle :: switchcase ⇒ switchcases (‹-›)
-switchcasesCons:: switchcase ⇒ switchcases ⇒ switchcases
(‹-/ | -›)
-Basic:: basicblock ⇒ ('s,'p,'f) com (‹(0BASIC/ (-)/ END)› [22] 71)
-BasicBlock:: basics ⇒ basicblock (‹-›)
-BAssign :: 'b => 'b => basic (‹(- :==/ -)› [30, 30] 23)
:: basic ⇒ basics (‹-›)
-basics :: [basic, basics] ⇒ basics (‹-,/ -›)

syntax (ASCII)
-Assert :: 'a => 'a set (‹({|-|})› [0] 1000)
-AssertState :: idt ⇒ 'a ⇒ 'a set (‹({|-. -|})› [1000,0] 1000)
-While-guard :: grds => 'a bexp => bdy ⇒ ('a,'p,'f) com
(‹(0WHILE (-|-> /-) /-)› [0,0,1000] 71)
-While-guard-inv:: grds⇒'a bexp⇒'a assn⇒bdy ⇒ ('a,'p,'f) com
(‹(0WHILE (-|-> /-) INV (-) /-)› [0,0,0,1000] 71)
-guards :: grds ⇒ ('s,'p,'f) com ⇒ ('s,'p,'f) com (‹(-|->-)› [60, 21] 23)

syntax (**output**)
-hidden-grds :: grds (‹. . .›)

translations

```
-Do c => c
b· c => CONST condCatch c b SKIP
b· (-DoPre c) <= CONST condCatch c b SKIP
l· (CONST whileAnnoG gs b I V c) <= l· (-DoPre (CONST whileAnnoG gs b I
V c))
l· (CONST whileAnno b I V c) <= l· (-DoPre (CONST whileAnno b I V c))
CONST condCatch c b SKIP <= (-DoPre (CONST condCatch c b SKIP))
-Do c <= -DoPre c
c;; d == CONST Seq c d
-guarantee g => (CONST True, g)
-guaranteeStrip g == CONST guaranteeStripPair (CONST True) g
-grd g => (CONST False, g)
-grds g gs => g#gs
-last-grd g => [g]
-guards gs c == CONST guards gs c

{|s. P|}                  == {|-antiquoteCur((=) s) ∧ P |}
{|b|}                    => CONST Collect (-quote b)
IF b THEN c1 ELSE c2 FI => CONST Cond {|b|} c1 c2
IF b THEN c1 FI          == IF b THEN c1 ELSE SKIP FI
IF_g b THEN c1 FI        == IF_g b THEN c1 ELSE SKIP FI

-While-inv-var b I V c          => CONST whileAnno {|b|} I V c
-While-inv-var b I V (-DoPre c) <= CONST whileAnno {|b|} I V c
-While-inv b I c                == -While-inv-var b I (CONST undefined) c
-While b c                      == -While-inv b {|CONST undefined|} c

-While-guard-inv-var gs b I V c         => CONST whileAnnoG gs {|b|} I V c

-While-guard-inv gs b I c      == -While-guard-inv-var gs b I (CONST undefined)
c
-While-guard gs b c            == -While-guard-inv gs b {|CONST undefined|} c

-GuardedWhile-inv b I c  == -GuardedWhile-inv-var b I (CONST undefined) c
-GuardedWhile b c        == -GuardedWhile-inv b {|CONST undefined|} c

TRY c1 CATCH c2 END     == CONST Catch c1 c2
ANNO s. P c Q,A => CONST specAnno (λs. P) (λs. c) (λs. Q) (λs. A)
ANNO s. P c Q == ANNO s. P c Q,{}

-WhileFix-inv-var b z I V c => CONST whileAnnoFix {|b|} (λz. I) (λz. V) (λz.
c)
-WhileFix-inv-var b z I V (-DoPre c) <= -WhileFix-inv-var {|b|} z I V c
-WhileFix-inv b z I c == -WhileFix-inv-var b z I (CONST undefined) c

-GuardedWhileFix-inv b z I c == -GuardedWhileFix-inv-var b z I (CONST un-
defined) c
```

-GuardedWhileFix-inv-var b z I V c =>
-GuardedWhileFix-inv-var-hook {|*b*|} (λ*z*. *I*) (λ*z*. *V*) (λ*z*. *c*)

-WhileFix-guard-inv-var gs b z I V c =>
CONST whileAnnoGFix gs {|*b*|} (λ*z*. *I*) (λ*z*. *V*) (λ*z*. *c*)
-WhileFix-guard-inv-var gs b z I V (*-DoPre c*) *<=*
-WhileFix-guard-inv-var gs {|*b*|} *z I V c*
-WhileFix-guard-inv gs b z I c == -WhileFix-guard-inv-var gs b z I (*CONST undefined*) *c*
LEMMA x c END == CONST lem x c

translations
(*-switchcase V c*) => (*V*,*c*)
(*-switchcasesSingle b*) => [*b*]
(*-switchcasesCons b bs*) => *CONST Cons b bs*
(*-Switch v vs*) => *CONST switch* (*-quote v*) *vs*

⟨*ML*⟩

syntax
-faccess :: ′*ref* ⇒ (′*ref* ⇒ ′*v*) ⇒ ′*v*
(‹-→-› [*65*,*1000*] *100*)

syntax (*ASCII*)
-faccess :: ′*ref* ⇒ (′*ref* ⇒ ′*v*) ⇒ ′*v*
(‹-−>-› [*65*,*1000*] *100*)

translations

p→*f* => *f p*
g→(*-antiquoteCur f*) <= *-antiquoteCur f g*

nonterminal *par* **and** *pars* **and** *actuals*

syntax
-par :: ′*a* ⇒ *par* (‹-›)
:: *par* ⇒ *pars* (‹-›)
-pars :: [*par*,*pars*] ⇒ *pars* (‹-,/-›)
-actuals :: *pars* ⇒ *actuals* (‹′(-′)›)
-actuals-empty :: *actuals* (‹′(′)›)

syntax *-Call* :: ′*p* ⇒ *actuals* ⇒ ((′*a*,*string*,′*f*) *com*) (‹*CALL* --› [*1000*,*1000*] *21*)
-GuardedCall :: ′*p* ⇒ *actuals* ⇒ ((′*a*,*string*,′*f*) *com*) (‹$CALL_g$ --› [*1000*,*1000*] *21*)
-CallAss:: ′*a* ⇒ ′*p* ⇒ *actuals* ⇒ ((′*a*,*string*,′*f*) *com*)

(‹- :== CALL --› [30,1000,1000] 21)
-Call-exn :: 'p ⇒ actuals ⇒ (('a,string,'f) com) (‹$CALL_e$ --› [1000,1000] 21)
-CallAss-exn:: 'a ⇒ 'p ⇒ actuals ⇒ (('a,string,'f) com)
(‹- :== $CALL_e$ --› [30,1000,1000] 21)
-Proc :: 'p ⇒ actuals ⇒ (('a,string,'f) com) (‹PROC --› 21)
-ProcAss:: 'a ⇒ 'p ⇒ actuals ⇒ (('a,string,'f) com)
(‹- :== PROC --› [30,1000,1000] 21)
-GuardedCallAss:: 'a ⇒ 'p ⇒ actuals ⇒ (('a,string,'f) com)
(‹- :== $CALL_g$ --› [30,1000,1000] 21)
-DynCall :: 'p ⇒ actuals ⇒ (('a,string,'f) com) (‹DYNCALL --› [1000,1000] 21)
-GuardedDynCall :: 'p ⇒ actuals ⇒ (('a,string,'f) com) (‹$DYNCALL_g$ --› [1000,1000] 21)
-DynCallAss:: 'a ⇒ 'p ⇒ actuals ⇒ (('a,string,'f) com)
(‹- :== DYNCALL --› [30,1000,1000] 21)
-DynCall-exn :: 'p ⇒ actuals ⇒ (('a,string,'f) com) (‹$DYNCALL_e$ --› [1000,1000] 21)
-DynCallAss-exn:: 'a ⇒ 'p ⇒ actuals ⇒ (('a,string,'f) com)
(‹- :== $DYNCALL_e$ --› [30,1000,1000] 21)
-GuardedDynCallAss:: 'a ⇒ 'p ⇒ actuals ⇒ (('a,string,'f) com)
(‹- :== $DYNCALL_g$ --› [30,1000,1000] 21)

-Bind:: ['s ⇒ 'v, idt, 'v ⇒ ('s,'p,'f) com] ⇒ ('s,'p,'f) com
(‹- ≫ -./ -› [22,1000,21] 21)
-bseq::('s,'p,'f) com ⇒ ('s,'p,'f) com ⇒ ('s,'p,'f) com
(‹-≫/ -› [22, 21] 21)
-FCall :: ['p,actuals,idt,(('a,string,'f) com)]⇒ (('a,string,'f) com)
(‹CALL -- ≫ -./ -› [1000,1000,1000,21] 21)

translations
-Bind e i c == CONST bind (-quote e) (λi. c)
-FCall p acts i c == -FCall p acts (λi. c)
-bseq c d == CONST bseq c d

nonterminal *modifyargs*

syntax
-may-modify :: ['a,'a,modifyargs] ⇒ bool
(‹- may'-only'-modify'-globals - in [-]› [100,100,0] 100)
-may-not-modify :: ['a,'a] ⇒ bool
(‹- may'-not'-modify'-globals -› [100,100] 100)
-may-modify-empty :: ['a,'a] ⇒ bool
(‹- may'-only'-modify'-globals - in []› [100,100] 100)
-modifyargs :: [id,modifyargs] ⇒ modifyargs (‹-,/ -›)
:: id => modifyargs (‹-›)

translations
s may-only-modify-globals Z in [] => *s may-not-modify-globals Z*

definition *Let'*:: ['*a*, '*a* => '*b*] => '*b*
  where *Let'* = *Let*

⟨*ML*⟩

syntax
-*Measure*:: ('*a* ⇒ *nat*) ⇒ ('*a* × '*a*) *set*
      (‹*MEASURE* -› [*22*] *1*)
-*Mlex*:: ('*a* ⇒ *nat*) ⇒ ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set*
      (**infixr** ‹<∗*MLEX*∗>› *30*)

syntax-consts
-*Measure* == *measure* **and**
-*Mlex* == *mlex-prod*

translations
MEASURE f => (*CONST measure*) (-*quote f*)
f <∗*MLEX*∗> *r* => (-*quote f*) <∗*mlex*∗> *r*

⟨*ML*⟩

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 Γ⊢ {|´*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 *vcg* method performs the basic state update for us; we may apply the Simplifier afterwards to achieve "obvious" consequences as well.

lemma Γ⊢ ⦃*True*⦄ ´*N* :== *10* ⦃´*N* = *10*⦄
⟨*proof*⟩

lemma Γ⊢ ⦃*2* ∗ ´*N* = *10*⦄ ´*N* :== *2* ∗ ´*N* ⦃´*N* = *10*⦄
⟨*proof*⟩

lemma Γ⊢ ⦃´*N* = *5*⦄ ´*N* :== *2* ∗ ´*N* ⦃´*N* = *10*⦄
⟨*proof*⟩

lemma $\Gamma\vdash$ $\{\!|\,{}^\acute{}N + 1 = a + 1|\!\}$ ${}^\acute{}N :== {}^\acute{}N + 1$ $\{\!|\,{}^\acute{}N = a + 1|\!\}$
⟨*proof*⟩

lemma $\Gamma\vdash$ $\{\!|\,{}^\acute{}N = a|\!\}$ ${}^\acute{}N :== {}^\acute{}N + 1$ $\{\!|\,{}^\acute{}N = a + 1|\!\}$
⟨*proof*⟩

lemma $\Gamma\vdash$ $\{\!|a = a \wedge b = b|\!\}$ ${}^\acute{}M :== a;;\ {}^\acute{}N :== b$ $\{\!|\,{}^\acute{}M = a \wedge {}^\acute{}N = b|\!\}$
⟨*proof*⟩

lemma $\Gamma\vdash$ $\{\!|\,True|\!\}$ ${}^\acute{}M :== a;;\ {}^\acute{}N :== b$ $\{\!|\,{}^\acute{}M = a \wedge {}^\acute{}N = b|\!\}$
⟨*proof*⟩

lemma $\Gamma\vdash$ $\{\!|\,{}^\acute{}M = a \wedge {}^\acute{}N = b|\!\}$
${}^\acute{}I :== {}^\acute{}M;;\ {}^\acute{}M :== {}^\acute{}N;;\ {}^\acute{}N :== {}^\acute{}I$
$\{\!|\,{}^\acute{}M = b \wedge {}^\acute{}N = a|\!\}$
⟨*proof*⟩

We can also perform verification conditions generation step by step by using the *vcg-step* method.

lemma $\Gamma\vdash$ $\{\!|\,{}^\acute{}M = a \wedge {}^\acute{}N = b|\!\}$
${}^\acute{}I :== {}^\acute{}M;;\ {}^\acute{}M :== {}^\acute{}N;;\ {}^\acute{}N :== {}^\acute{}I$
$\{\!|\,{}^\acute{}M = b \wedge {}^\acute{}N = a|\!\}$
⟨*proof*⟩

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 $\Gamma\vdash$ $\{\!|\,{}^\acute{}N = a|\!\}$ ${}^\acute{}N :== {}^\acute{}N$ $\{\!|\,{}^\acute{}N = a|\!\}$
⟨*proof*⟩

lemma $\Gamma\vdash\{s.\ x\text{-}'\ s = a\}$ $(Basic\ (\lambda s.\ x\text{-}'\text{-}update\ (x\text{-}'\ s)\ s))$ $\{s.\ x\text{-}'\ s = a\}$
⟨*proof*⟩

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 $\Gamma\vdash$ $\{\!|\,{}^\acute{}M = {}^\acute{}N|\!\}$ ${}^\acute{}M :== {}^\acute{}M + 1$ $\{\!|\,{}^\acute{}M \neq {}^\acute{}N|\!\}$
⟨*proof*⟩

lemma $\Gamma\vdash$ $\{\!|\,{}^\acute{}M = {}^\acute{}N|\!\}$ ${}^\acute{}M :== {}^\acute{}M + 1$ $\{\!|\,{}^\acute{}M \neq {}^\acute{}N|\!\}$
⟨*proof*⟩

lemma $\Gamma\vdash$ $\{\!|\,{}^\acute{}M = {}^\acute{}N|\!\}$ ${}^\acute{}M :== {}^\acute{}M + 1$ $\{\!|\,{}^\acute{}M \neq {}^\acute{}N|\!\}$

⟨*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 Γ⊢ {|´*M* = *0* ∧ ´*S* = *0*|}
WHILE ´*M* ≠ *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 Γ⊢ {|´*M* = *0* ∧ ´*S* = *0*|}
WHILE ´*M* ≠ *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 Γ⊢ {|´*M* = *0* ∧ ´*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*|}
⟨*proof*⟩

lemma Γ⊢ {|´*M* = *0* ∧ ´*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

$\{\!|\acute{}S = a * b|\!\}$
⟨proof⟩

Some more syntactic sugar, the label statement ... • ... as shorthand for the *TRY−CATCH* above, and the *RAISE* for an state-update followed by a *THROW*.

lemma $\Gamma\vdash$ $\{\!|\acute{}M = 0 \wedge \acute{}S = 0|\!\}$
$\{\!|\acute{}Abr = ''Break''|\!\}\cdot$ *WHILE True INV* $\{\!|\acute{}S = \acute{}M * b|\!\}$
DO IF ´M = a THEN RAISE ´Abr :== ''Break''
ELSE ´S :== ´S + b;; ´M :== ´M + 1
FI
OD
$\{\!|\acute{}S = a * b|\!\}$
⟨proof⟩

lemma $\Gamma\vdash$ $\{\!|\acute{}M = 0 \wedge \acute{}S = 0|\!\}$
TRY
WHILE True
INV $\{\!|\acute{}S = \acute{}M * b|\!\}$
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
$\{\!|\acute{}S = a * b|\!\}$
⟨proof⟩

lemma $\Gamma\vdash$ $\{\!|\acute{}M = 0 \wedge \acute{}S = 0|\!\}$
$\{\!|\acute{}Abr = ''Break''|\!\}\cdot$ *WHILE True*
INV $\{\!|\acute{}S = \acute{}M * b|\!\}$
DO IF ´M = a THEN RAISE ´Abr :== ''Break''
ELSE ´S :== ´S + b;; ´M :== ´M + 1
FI
OD
$\{\!|\acute{}S = a * b|\!\}$
⟨proof⟩

Blocks

lemma $\Gamma\vdash\{\!|\acute{}I = i|\!\}$ *LOC ´I;; ´I :== 2 COL* $\{\!|\acute{}I \leq i|\!\}$
⟨proof⟩
lemma $\Gamma\vdash$ $\{\!|\acute{}N = n|\!\}$ *LOC ´N :== 10;; ´N :== ´N + 2 COL* $\{\!|\acute{}N = n|\!\}$
⟨proof⟩

lemma $\Gamma\vdash$ $\{\!|\acute{}N = n|\!\}$ *LOC ´N :== 10, ´M;; ´N :== ´N + 2 COL* $\{\!|\acute{}N = n|\!\}$
⟨proof⟩

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* => *nat*) => *nat* => *nat*
where
  sum f 0 = *0*
| *sum f* (*Suc n*) = *f n* + *sum f n*

syntax
  -sum :: *idt* => *nat* => *nat* => *nat*
    (‹*SUMM* -<-. -› [*0*, *0*, *10*] *10*)
syntax-consts
  -sum == *sum*
translations
  SUMM j<*k*. *b* == *CONST sum* (λ*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 Γ⊢ {|*True*|}
    ´S :== *0*;; *´I* :== *1*;;
    WHILE ´I ≠ *n*
    DO
      ´S :== *´S* + *´I*;;
      ´I :== *´I* + *1*
    OD
    {|*´S* = (*SUMM j*<*n*. *j*)|}
  (**is** Γ⊢ - (-;; *?while*) -)
⟨*proof*⟩

The next version uses the *vcg* method, while still explaining the resulting proof obligations in an abstract, structured manner.

theorem Γ⊢ {|*True*|}
    ´S :== *0*;; *´I* :== *1*;;
    WHILE ´I ≠ *n*
    INV {|*´S* = (*SUMM j*<*´I*. *j*)|}
    DO
      ´S :== *´S* + *´I*;;
      ´I :== *´I* + *1*
    OD
    {|*´S* = (*SUMM j*<*n*. *j*)|}
⟨*proof*⟩

Certainly, this proof may be done fully automatically as well, provided that the invariant is given beforehand.

theorem Γ⊢ ⦃*True*⦄
´S :== 0;; ´I :== 1;;
WHILE ´I ≠ n
INV ⦃´S = (SUMM j<´I. j)⦄
DO
´S :== ´S + ´I;;
´I :== ´I + 1
OD
⦃´S = (SUMM j<n. j)⦄
⟨*proof*⟩

17.5 SWITCH

lemma Γ⊢ ⦃*´N = 5*⦄ *SWITCH ´B*
{True} ⇒ ´N :== 6
| {False} ⇒ ´N :== 7
END
⦃´N > 5⦄
⟨*proof*⟩

lemma Γ⊢ ⦃*´N = 5*⦄ *SWITCH ´N*
{v. v < 5} ⇒ ´N :== 6
| {v. v ≥ 5} ⇒ ´N :== 7
END
⦃´N > 5⦄
⟨*proof*⟩

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 ⇒ nat*
where
fac 0 = 1 |
fac (Suc n) = (Suc n) ∗ fac n

lemma *fac-simp* [*simp*]: *0 < i ⟹ fac i = i ∗ fac (i − 1)*
⟨*proof*⟩

Now we define the procedure

procedures
Fac (N|R) = IF ´N = 0 THEN ´R :== 1
ELSE ´R :== CALL Fac(´N − 1);;
´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 Γ ''*Fac*'' = *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 Γ 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 (λ*s. s*(|*N-'* := *N-' s*|)) *Fac-'proc* (λ*s t. s*(|*globals* := *globals t*|)) (λ*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 parameterless 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*. Γ,$\Theta$$\vdash${|*´N=n*|} *PROC Fac(´N,´R)* {|*´R = fac 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):

$[\![\forall Z.\ \Gamma,\Theta \cup (\bigcup_Z \{(P\ Z,\ p,\ Q\ Z,\ A\ Z)\})\vdash_{/F} (P\ Z)\ \mathit{the}\ (\Gamma\ p)\ (Q\ Z),(A\ Z);$
$p \in \mathit{dom}\ \Gamma]\!]$
$\Longrightarrow \forall Z.\ \Gamma,\Theta\vdash_{/F} (P\ Z)\ \mathit{Call}\ p\ (Q\ Z),(A\ Z)$

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 $\forall n.\ \Gamma\vdash\{\!|\ \acute{}N{=}n|\!\}\ \acute{}R := = \mathit{PROC}\ \mathit{Fac}(\acute{}N)\ \{\!|\acute{}R = \mathit{fac}\ n|\!\}$
  ⟨*proof*⟩

Here some Isar style version of the proof

lemma (**in** *Fac-impl*)
  shows $\forall n.\ \Gamma\vdash\{\!|\ \acute{}N{=}n|\!\}\ \acute{}R := = \mathit{PROC}\ \mathit{Fac}(\acute{}N)\ \{\!|\acute{}R = \mathit{fac}\ n|\!\}$
⟨*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 $\forall n.\ \Gamma\vdash\{\!|\ \acute{}N{=}n|\!\}\ \acute{}R := = \mathit{PROC}\ \mathit{Fac}(\acute{}N)\ \{\!|\acute{}R = \mathit{fac}\ n|\!\}$
  (**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 $\forall n.\ \Gamma\vdash\{\!|\ \acute{}N{=}n|\!\}\ \acute{}R := = \mathit{PROC}\ \mathit{Fac}(\acute{}N)\ \{\!|\acute{}R = \mathit{fac}\ n|\!\}$
  (**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. Thats 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:

⟨*ML*⟩

lemma (**in** *odd-even-clique*)
shows *odd-spec*: ∀ *n*. Γ⊢⦃´*N*=*n*⦄ ´*A* :== *PROC odd*(´*N*)
⦃(∃ *b*. *n* = *2* ∗ *b* + ´*A*) ∧ ´*A* < *2* ⦄ (**is** *?P1*)
and *even-spec*: ∀ *n*. Γ⊢⦃´*N*=*n*⦄ ´*A* :== *PROC even*(´*N*)
⦃(∃ *b*. *n* + *1* = *2* ∗ *b* + ´*A*) ∧ ´*A* < *2* ⦄ (**is** *?P2*)
⟨*proof*⟩

17.7 Expressions With Side Effects

```
R := N++ + M++
```

lemma Γ⊢ ⦃*True*⦄
´*N* ≫ *n*. ´*N* :== ´*N* + *1* ≫

´*M* ≫ *m*. ´*M* :== ´*M* + *1* ≫
´*R* :== *n* + *m*
⦃´*R* = ´*N* + ´*M* − *2*⦄
⟨*proof*⟩

`R := Fac (N) + Fac (M)`

lemma (**in** *Fac-impl*) **shows**
Γ⊢ ⦃*True*⦄
CALL Fac(´*N*) ≫ *n*. *CALL Fac*(´*M*) ≫ *m*.
´*R* :== *n* + *m*
⦃´*R* = *fac* ´*N* + *fac* ´*M*⦄
⟨*proof*⟩

`R := (Fac(Fac (N)))`

lemma (**in** *Fac-impl*) **shows**
Γ⊢ ⦃*True*⦄
CALL Fac(´*N*) ≫ *n*. *CALL Fac*(*n*) ≫ *m*.
´*R* :== *m*
⦃´*R* = *fac* (*fac* ´*N*)⦄
⟨*proof*⟩

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* ⇒ *ref*
cont-′ :: *ref* ⇒ *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 [[*hoare-use-call-tr′* = *false*]]
term *CALL append(´p,´q,´p→´next)*
declare [[*hoare-use-call-tr′* = *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 σ, 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 $\{\!|\sigma. \ldots|\!\}$ is a shorthand to fix the current state: $\{s.\ \sigma = s \ldots\}$.

lemma (**in** *append-impl*) *append-spec*:
shows $\forall \sigma\ Ps\ Qs.\ \Gamma\vdash$
$\{\!|\sigma.\ List\ ´p\ ´next\ Ps \wedge\ List\ ´q\ ´next\ Qs \wedge set\ Ps \cap set\ Qs = \{\}|\!\}$
$´p :== PROC\ append(´p,´q)$
$\{\!|List\ ´p\ ´next\ (Ps@Qs) \wedge (\forall x.\ x \notin set\ Ps \longrightarrow\ ´next\ x = {}^{\sigma}next\ x)|\!\}$
⟨*proof*⟩

The modifies clause is equal to a proper record update specification of the following form.

lemma $\{t.\ t\ may\text{-}only\text{-}modify\text{-}globals\ Z\ in\ [next]\}$
=
$\{t.\ \exists next.\ globals\ t{=}next\text{-}'\text{-}update\ (\lambda\text{-}.\ next)\ (globals\ Z)\}$
⟨*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 $\Gamma\vdash$ $\{\!|\,\acute{}p{=}Null \wedge \acute{}cont{=}c|\!\}$ $\acute{}p :== CALL\ append(\acute{}p{,}Null)$ $\{\!|\,\acute{}cont{=}c|\!\}$
  ⟨*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
  $\forall \sigma.\ \Gamma\vdash \{\sigma\}\ \acute{}p :== PROC\ append(\acute{}p,\acute{}q)\{t.\ t\ may\text{-}only\text{-}modify\text{-}globals\ \sigma\ in\ [next]\}$
  ⟨*proof*⟩

lemma (**in** *append-impl*)
  shows $\Gamma\vdash$ $\{\!|\,\acute{}p{=}Null \wedge \acute{}cont{=}c|\!\}$ $\acute{}p{\rightarrow}\acute{}next :== CALL\ append(\acute{}p{,}Null)$ $\{\!|\,\acute{}cont{=}c|\!\}$
  ⟨*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 me 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 $(x\#xs) = ((\forall y{\in}set\ xs.\ le\ x\ y) \wedge sorted\ le\ xs)$

procedures

insert(*r*,*p* | *p*) =
  IF ´r=Null THEN SKIP
   ELSE IF ´p=Null THEN ´p :== ´r;; ´p→´next :== Null
      ELSE IF ´r→´cont ≤ ´p→´cont
         THEN ´r→´next :== ´p;; ´p:==´r
         ELSE ´p→´next :== CALL insert(*´r*,*´p→´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 *´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. Thats 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 :== PROC insert*(*´r*,*´p*)$\{t.$ *t may-only-modify-globals* σ *in* [*next*]$\}$
⟨*proof*⟩

lemma (**in** *insert-impl*) *insert-spec*:
  $\forall\sigma$ *Ps* . $\Gamma\vdash$ ⦃σ. *List ´p ´next Ps* ∧ *sorted* (≤) (*map ´cont Ps*) ∧
      ´r ≠ *Null* ∧ *´r* ∉ *set Ps*⦄
    ´p :== PROC insert(*´r*,*´p*)
  ⦃∃ *Qs*. *List ´p ´next Qs* ∧ *sorted* (≤) (*map* $^{\sigma}$*cont Qs*) ∧
    set Qs = *insert* $^{\sigma}$*r* (*set Ps*) ∧
    (∀ *x*. *x* ∉ *set Qs* ⟶ *´next x* = $^{\sigma}$*next x*)⦄

⟨*proof*⟩

procedures

insertSort(*p* | *p*) =
  ´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
$\forall\sigma.\ \Gamma\vdash \{\sigma\}$ *´p :== PROC insertSort(´p)*
$\{t.$ *t may-only-modify-globals* σ *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*. Γ,Θ⊢ ⦃σ. *List ´p ´next Ps* ⦄
´p :== PROC insertSort(´p)
⦃∃ *Qs*. *List ´p ´next Qs* ∧ *sorted* (≤) (*map* $^{\sigma}$*cont Qs*) ∧
set Qs = *set Ps*⦄
⟨*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 Γ **for** Γ :: *'c* ⇀ ((*'a globals-list-alloc-scheme*, *'b*) *list-vars'-scheme*, *'c*, *'d*) *com*

lemma (**in** *hoare-ex*)
$\Gamma\vdash$ $\{\!|$ ´$i = 0 \wedge$ ´$first = Null \wedge n{*}sz \leq$ ´$free|\!\}$
WHILE ´$i < n$
INV $\{\!|\exists Ps.$ *List* ´*first* ´*next Ps* $\wedge$ *length Ps* $=$ ´$i \wedge$ ´$i \leq n \wedge$
set Ps $\subseteq$ *set* ´*alloc* $\wedge$ $(n -$ ´$i){*}sz \leq$ ´$free|\!\}$
DO
´*p* :== *NEW sz* [´*cont*:==*0*,´*next*:== *Null*];;
´*p*→´*next* :== ´*first*;;
´*first* :== ´*p*;;
´*i* :== ´*i*+ *1*
OD
$\{\!|\exists Ps.$ *List* ´*first* ´*next* *Ps* $\wedge$ *length Ps* $= n \wedge$ *set Ps* $\subseteq$ *set* ´*alloc*$|\!\}$

⟨*proof*⟩

lemma (**in** *hoare-ex*)
$\Gamma\vdash$ $\{\!|$ ´$i = 0 \wedge$ ´$first = Null \wedge n{*}sz \leq$ ´$free|\!\}$
WHILE ´$i < n$
INV $\{\!|\exists Ps.$ *List* ´*first* ´*next Ps* $\wedge$ *length Ps* $=$ ´$i \wedge$ ´$i \leq n \wedge$
set Ps $\subseteq$ *set* ´*alloc* $\wedge$ $(n -$ ´$i){*}sz \leq$ ´$free|\!\}$
DO
´*p* :== *NNEW sz* [´*cont*:==*0*,´*next*:== *Null*];;
´*p*→´*next* :== ´*first*;;
´*first* :== ´*p*;;
´*i* :== ´*i*+ *1*
OD
$\{\!|\exists Ps.$ *List* ´*first* ´*next* *Ps* $\wedge$ *length Ps* $= n \wedge$ *set Ps* $\subseteq$ *set* ´*alloc*$|\!\}$

⟨*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*$|\!\}$ $\{\!|$´$p \neq Null|\!\}\longmapsto$ ´*p*→´*next* :== ´*p* $\{\!|$*True*$|\!\}$
⟨*proof*⟩

lemma $\Gamma\vdash$ $\{\}$ $\{\!|$´$p \neq Null|\!\}\longmapsto$ ´*p*→´*next* :== ´*p* $\{\!|$*True*$|\!\}$
⟨*proof*⟩

Let us consider this small program that reverts a list. At first without guards.

lemma (**in** *hoare-ex*) *rev-strip*:
$\Gamma\vdash$ $\{\!|$*List* ´*p* ´*next Ps* $\wedge$ *List* ´*q* ´*next Qs* $\wedge$ *set Ps* $\cap$ *set Qs* $= \{\} \wedge$
set Ps $\subseteq$ *set* ´*alloc* $\wedge$ *set Qs* $\subseteq$ *set* ´*alloc*$|\!\}$

  WHILE ´p ≠ Null
  INV ⦃∃ps qs. List ´p ´next ps ∧ List ´q ´next qs ∧ set ps ∩ set qs = {} ∧
        rev ps @ qs = rev Ps @ Qs ∧
        set ps ⊆ set ´alloc ∧ set qs ⊆ set ´alloc⦄
  DO ´r :== ´p;;
    ´p :== ´p→ ´next;;
    ´r→´next :== ´q;;
    ´q :== ´r OD
  ⦃List ´q ´next (rev Ps @ Qs) ∧ set Ps⊆ set ´alloc ∧ set Qs ⊆ set ´alloc⦄
⟨proof⟩

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 Γ **for** Γ :: 'c ⇀ (('a globals-list-alloc-scheme, 'b) list-vars'-scheme, 'c, bool) com

lemma
  (**in** *hoare-ex-guard*)
  Γ⊢ ⦃List ´p ´next Ps ∧ List ´q ´next Qs ∧ set Ps ∩ set Qs = {} ∧
     set Ps ⊆ set ´alloc ∧ set Qs ⊆ set ´alloc⦄
  WHILE ´p ≠ Null
  INV ⦃∃ps qs. List ´p ´next ps ∧ List ´q ´next qs ∧ set ps ∩ set qs = {} ∧
        rev ps @ qs = rev Ps @ Qs ∧
        set ps ⊆ set ´alloc ∧ set qs ⊆ set ´alloc⦄
  DO ´r :== ´p;;
    ⦃´p≠Null ∧ ´p∈set ´alloc⦄⟼ ´p :== ´p→ ´next;;
    ⦃´r≠Null ∧ ´r∈set ´alloc⦄⟼ ´r→´next :== ´q;;
    ´q :== ´r OD
  ⦃List ´q ´next (rev Ps @ Qs) ∧ set Ps ⊆ set ´alloc ∧ set Qs ⊆ set ´alloc⦄
⟨proof⟩

We can also just prove that no faults will occur, by giving the trivial post-condition.

lemma (**in** *hoare-ex-guard*) *rev-noFault*:
  Γ⊢ ⦃List ´p ´next Ps ∧ List ´q ´next Qs ∧ set Ps ∩ set Qs = {} ∧
     set Ps ⊆ set ´alloc ∧ set Qs ⊆ set ´alloc⦄
  WHILE ´p ≠ Null
  INV ⦃∃ps qs. List ´p ´next ps ∧ List ´q ´next qs ∧ set ps ∩ set qs = {} ∧
        rev ps @ qs = rev Ps @ Qs ∧
        set ps ⊆ set ´alloc ∧ set qs ⊆ set ´alloc⦄
  DO ´r :== ´p;;
    ⦃´p≠Null ∧ ´p∈set ´alloc⦄⟼ ´p :== ´p→ ´next;;
    ⦃´r≠Null ∧ ´r∈set ´alloc⦄⟼ ´r→´next :== ´q;;
    ´q :== ´r OD
  UNIV,UNIV
⟨proof⟩

lemma (**in** *hoare-ex-guard*) *rev-moduloGuards*:

$\Gamma\vdash_{/\{True\}}$ {|*List ´p ´next Ps* ∧ *List ´q ´next Qs* ∧ *set Ps* ∩ *set Qs* = {} ∧
    set Ps ⊆ *set ´alloc* ∧ *set Qs* ⊆ *set ´alloc*|}
  WHILE ´p ≠ *Null*
  INV {|∃ *ps qs. List ´p ´next ps* ∧ *List ´q ´next qs* ∧ *set ps* ∩ *set qs* = {} ∧
        rev ps @ *qs* = *rev Ps* @ *Qs* ∧
        set ps ⊆ *set ´alloc* ∧ *set qs* ⊆ *set ´alloc*|}
  DO ´r :== *´p*;;
    {|*´p*≠*Null* ∧ *´p*∈*set ´alloc*|}$\surd$ ⟼ *´p* :== *´p*→ *´next*;;
    {|*´r*≠*Null* ∧ *´r*∈*set ´alloc*|}$\surd$ ⟼ *´r*→*´next* :== *´q*;;
    ´q :== *´r OD*
 {|*List ´q ´next* (*rev Ps* @ *Qs*) ∧ *set Ps* ⊆ *set ´alloc* ∧ *set Qs* ⊆ *set ´alloc*|}
⟨*proof*⟩

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-guards False* (*strip-guards* (−*F*) *c′*)
  assumes *c*: *c* = *mark-guards False c′*
  shows $\Gamma,\Theta\vdash_{/\{\}}$ *P c Q*,*A*
⟨*proof*⟩

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 ⟦*?*Γ, *?*Θ$\vdash_{/?F}$ *?P ?c ?Q*, *?A*; *?*Γ, *?*Θ⊢ *?P strip-guards* (− *?F*) *?c UNIV*, *UNIV*⟧ ⟹ *?*Γ, *?*Θ⊢ *?P ?c ?Q*, *?A*

lemma
  (**in** *hoare-ex-guard*)
  Γ⊢ {|*List ´p ´next Ps* ∧ *List ´q ´next Qs* ∧ *set Ps* ∩ *set Qs* = {} ∧
    set Ps ⊆ *set ´alloc* ∧ *set Qs* ⊆ *set ´alloc*|}
  WHILE ´p ≠ *Null*
  INV {|∃ *ps qs. List ´p ´next ps* ∧ *List ´q ´next qs* ∧ *set ps* ∩ *set qs* = {} ∧
        rev ps @ *qs* = *rev Ps* @ *Qs* ∧
        set ps ⊆ *set ´alloc* ∧ *set qs* ⊆ *set ´alloc*|}
  DO ´r :== *´p*;;
    {|*´p*≠*Null* ∧ *´p*∈*set ´alloc*|}⟼ *´p* :== *´p*→ *´next*;;
    {|*´r*≠*Null* ∧ *´r*∈*set ´alloc*|}⟼ *´r*→*´next* :== *´q*;;
    ´q :== *´r OD*
 {|*List ´q ´next* (*rev Ps* @ *Qs*) ∧ *set Ps* ⊆ *set ´alloc* ∧ *set Qs* ⊆ *set ´alloc*|}

⟨*proof*⟩

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*, $\{\!|\,{}^{\prime}p \neq Null|\!\}$)$\longmapsto$ ´p→´next ≠ Null DO SKIP OD

17.10 Circular Lists

definition
  distPath :: *ref* ⇒ (*ref* ⇒ *ref*) ⇒ *ref* ⇒ *ref list* ⇒ *bool* **where**
  distPath x next y as = (*Path x next y as* ∧ *distinct as*)

lemma *neq-dP*: ⟦*p* ≠ *q*; *Path p h q Ps*; *distinct Ps*⟧ ⟹
∃ *Qs*. *p*≠*Null* ∧ *Ps* = *p*#*Qs* ∧ *p* ∉ *set Qs*
⟨*proof*⟩

lemma *circular-list-rev-I*:
  Γ⊢ {| ´*root* = *r* ∧ *distPath* ´*root* ´*next* ´*root* (*r*#*Ps*)|}
   ´*p* :== ´*root*;; ´*q* :== ´*root*→´*next*;;
  WHILE ´*q* ≠ ´*root*
  INV {|∃ *ps qs*. *distPath* ´*p* ´*next* ´*root ps* ∧ *distPath* ´*q* ´*next* ´*root qs* ∧
        ´*root* = *r* ∧ *r*≠*Null* ∧ *r* ∉ *set Ps* ∧ *set ps* ∩ *set qs* = {} ∧
        Ps = (*rev ps*) @ *qs* |}
  DO ´*tmp* :== ´*q*;; ´*q* :== ´*q*→´*next*;; ´*tmp*→´*next* :== ´*p*;; ´*p*:==´*tmp OD*;;
  ´*root*→´*next* :== ´*p*
  {|´*root* = *r* ∧ *distPath* ´*root* ´*next* ´*root* (*r*#*rev Ps*)|}
⟨*proof*⟩

lemma *path-is-list*:⋀*a next b*. ⟦*Path b next a Ps* ; *a* ∉ *set Ps*; *a*≠*Null*⟧
⟹ *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 *circular-list-rev-II*:
 Γ⊢
 {|´*p* = *r* ∧ *distPath* ´*p* ´*next* ´*p* (*r*#*Ps*)|}
´*q*:==*Null*;;
WHILE ´*p* ≠ *Null*
INV
 {| ((´*q* = *Null*) ⟶ (∃ *ps*. *distPath* ´*p* ´*next r ps* ∧ *ps* = *r*#*Ps*)) ∧
  ((´*q* ≠ *Null*) ⟶ (∃ *ps qs*. *distPath* ´*q* ´*next r qs* ∧ *List* ´*p* ´*next ps* ∧

$set\ ps \cap set\ qs = \{\} \wedge rev\ qs\ @\ ps = Ps@[r])) \wedge$
$\neg\ (´p = Null \wedge ´q = Null \wedge r = Null\ )$
$|\!\}$
DO
$´tmp :== ´p;;\ ´p :== ´p \rightarrow ´next;;\ ´tmp \rightarrow ´next :== ´q;;\ ´q :== ´tmp$
OD
$\{|´q = r \wedge distPath\ ´q\ ´next\ ´q\ (r\ \#\ rev\ Ps)|\}$

$\langle proof \rangle$

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 $\Gamma \vdash \{\sigma\}$
$´I :== ´M;;$
$ANNO\ \tau.\ \{|\tau.\ ´I = {}^{\sigma}M|\}$
$´M :== ´N;;\ ´N :== ´I$
$\{|´M = {}^{\tau}N \wedge ´N = {}^{\tau}I|\}$
$\{|´M = {}^{\sigma}N \wedge ´N = {}^{\sigma}M|\}$
$\langle proof \rangle$

lemma $\Gamma \vdash (\{\sigma\} \cap \{|´M = 0 \wedge ´S = 0|\})$
$(ANNO\ \tau.\ (\{\tau\} \cap \{|´A = {}^{\sigma}A \wedge ´I = {}^{\sigma}I \wedge ´M = 0 \wedge ´S = 0|\})$
$WHILE\ ´M \neq ´A$
$INV\ \{|´S = ´M * ´I \wedge ´A = {}^{\tau}A \wedge ´I = {}^{\tau}I|\}$
$DO\ ´S :== ´S + ´I;;\ ´M :== ´M + 1\ OD$
$\{|´S = {}^{\tau}A * {}^{\tau}I|\})$
$\{|´S = {}^{\sigma}A * {}^{\sigma}I|\}$
$\langle proof \rangle$

Instead of annotations one can also directly use previously proven lemmas.

lemma *foo-lemma*: $\forall n\ m.\ \Gamma \vdash \{|´N = n \wedge ´M = m|\}\ ´N :== ´N + 1;;\ ´M :== ´M + 1$
$\{|´N = n + 1 \wedge ´M = m + 1|\}$
$\langle proof \rangle$

lemma $\Gamma \vdash \{|´N = n \wedge ´M = m|\}\ LEMMA\ foo\text{-}lemma$
$´N :== ´N + 1;;\ ´M :== ´M + 1$
$END;;$
$´N :== ´N + 1$
$\{|´N = n + 2 \wedge ´M = m + 1|\}$
$\langle proof \rangle$

lemma $\Gamma\vdash$ $\{\!|\acute{}N = n \wedge \acute{}M = m|\!\}$
LEMMA foo-lemma
$\acute{}N :== \acute{}N + 1;; \acute{}M :== \acute{}M + 1$
END;;
LEMMA foo-lemma
$\acute{}N :== \acute{}N + 1;; \acute{}M :== \acute{}M + 1$
END
$\{\!|\acute{}N = n + 2 \wedge \acute{}M = m + 2|\!\}$
⟨*proof*⟩

lemma $\Gamma\vdash$ $\{\!|\acute{}N = n \wedge \acute{}M = m|\!\}$
$\acute{}N :== \acute{}N + 1;; \acute{}M :== \acute{}M + 1;;$
$\acute{}N :== \acute{}N + 1;; \acute{}M :== \acute{}M + 1$
$\{\!|\acute{}N = n + 2 \wedge \acute{}M = m + 2|\!\}$
⟨*proof*⟩

Just some test on marked, guards

lemma $\Gamma\vdash\{\!|True|\!\}$ *WHILE* $\{\!|P\ \acute{}N\ |\!\}\surd, \{\!|Q\ \acute{}M|\!\}\#, \{\!|R\ \acute{}N|\!\}\longmapsto \acute{}N < \acute{}M$
INV $\{\!|\acute{}N < 2|\!\}$ *DO*
$\acute{}N :== \acute{}M$
OD
$\{\!|hard|\!\}$
⟨*proof*⟩

lemma $\Gamma\vdash_{/\{True\}}$ $\{\!|True|\!\}$ *WHILE* $\{\!|P\ \acute{}N\ |\!\}\surd, \{\!|Q\ \acute{}M|\!\}\#, \{\!|R\ \acute{}N|\!\}\longmapsto \acute{}N < \acute{}M$
INV $\{\!|\acute{}N < 2|\!\}$ *DO*
$\acute{}N :== \acute{}M$
OD
$\{\!|hard|\!\}$
⟨*proof*⟩

term $\Gamma\vdash_{/\{True\}}$ $\{\!|True|\!\}$ $WHILE_g$ $\acute{}N < \acute{}Arr!i$
FIX Z.
INV $\{\!|\acute{}N < 2|\!\}$

DO
$\acute{}N :== \acute{}M$
OD
$\{\!|hard|\!\}$

lemma $\Gamma\vdash_{/\{True\}}$ $\{\!|True|\!\}$ $WHILE_g$ $\acute{}N < \acute{}Arr!i$
FIX Z.
INV $\{\!|\acute{}N < 2|\!\}$
VAR arbitrary
DO
$\acute{}N :== \acute{}M$

OD
{|*hard*|}
⟨*proof*⟩

lemma $\Gamma\vdash_{/\{True\}}$ {|*True*|} *WHILE* {|*P ´N* |}√, {|*Q ´M*|}#, {|*R ´N*|}⟼ *´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*)Γ⊢ {|*´N = n*|} *LOC ´N :== 10*;; *´N :== ´N + 2 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*) $\Gamma\vdash \{|\acute{}N = 5|\}\ \acute{}N :== 2 * \acute{}N\ \{|\acute{}N = 10|\}$
  ⟨*proof*⟩

lemma (**in** *state-space*) $\Gamma\vdash \{\!|True|\!\}\ \acute{}N :== 10\ \{\!|\acute{}N = 10|\!\}$
  ⟨*proof*⟩

lemma (**in** *state-space*) $\Gamma\vdash \{\!|2 * \acute{}N = 10|\!\}\ \acute{}N :== 2 * \acute{}N\ \{\!|\acute{}N = 10|\!\}$
  ⟨*proof*⟩

lemma (**in** *state-space*) $\Gamma\vdash \{\!|\acute{}N = 5|\!\}\ \acute{}N :== 2 * \acute{}N\ \{\!|\acute{}N = 10|\!\}$
  ⟨*proof*⟩

lemma (**in** *state-space*) $\Gamma\vdash \{\!|\acute{}N + 1 = a + 1|\!\}\ \acute{}N :== \acute{}N + 1\ \{\!|\acute{}N = a + 1|\!\}$
  ⟨*proof*⟩

lemma (**in** *state-space*) $\Gamma\vdash \{\!|\acute{}N = a|\!\}\ \acute{}N :== \acute{}N + 1\ \{\!|\acute{}N = a + 1|\!\}$
  ⟨*proof*⟩

lemma (**in** *state-space*)
  shows $\Gamma\vdash \{\!|a = a \wedge b = b|\!\}\ \acute{}M :== a;;\ \acute{}N :== b\ \{\!|\acute{}M = a \wedge \acute{}N = b|\!\}$
  ⟨*proof*⟩

lemma (**in** *state-space*)
  shows $\Gamma\vdash \{\!|True|\!\}\ \acute{}M :== a;;\ \acute{}N :== b\ \{\!|\acute{}M = a \wedge \acute{}N = b|\!\}$
  ⟨*proof*⟩

lemma (**in** *state-space*)
  shows $\Gamma\vdash \{\!|\acute{}M = a \wedge \acute{}N = b|\!\}$
    $\acute{}I :== \acute{}M;;\ \acute{}M :== \acute{}N;;\ \acute{}N :== \acute{}I$
    $\{\!|\acute{}M = b \wedge \acute{}N = a|\!\}$
  ⟨*proof*⟩

We can also perform verification conditions generation step by step by using the *vcg-step* method.

lemma (**in** *state-space*)
  shows $\Gamma\vdash \{\!|\acute{}M = a \wedge \acute{}N = b|\!\}$
    $\acute{}I :== \acute{}M;;\ \acute{}M :== \acute{}N;;\ \acute{}N :== \acute{}I$
    $\{\!|\acute{}M = b \wedge \acute{}N = a|\!\}$
  ⟨*proof*⟩

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 ≠ ´N|}
⟨*proof*⟩

lemma (**in** *state-space*)
  shows $\Gamma\vdash$ {|´M = ´N|} ´M :== ´M + *1* {|´M ≠ ´N|}
⟨*proof*⟩

lemma (**in** *state-space*)
  shows $\Gamma\vdash$ {|´M = ´N|} ´M :== ´M + *1* {|´M ≠ ´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* ∧ ´S = *0*|}
    WHILE ´M ≠ 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* ∧ ´S = *0*|}
      WHILE ´M ≠ 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* ∧ ´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|}
⟨*proof*⟩

lemma (**in** *state-space*)
shows Γ⊢ {|´M = 0 ∧ ´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|}
⟨*proof*⟩

Some more syntactic sugar, the label statement ... • ... as shorthand for the *TRY−CATCH* above, and the *RAISE* for an state-update followed by a *THROW*.

lemma (**in** *state-space*)
shows Γ⊢ {|´M = 0 ∧ ´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*⟩

lemma (**in** *state-space*)
shows Γ⊢ {|´M = 0 ∧ ´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|}
⟨*proof*⟩

lemma (**in** *state-space*)
shows Γ⊢ {|´M = 0 ∧ ´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 Γ⊢{|´I = i|} *LOC ´I;; ´I :== 2 COL* {|´I ≤ 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* => *nat*) => *nat* => *nat*
where
sum f 0 = *0*
| *sum f* (*Suc n*) = *f n* + *sum f n*

syntax
-sum :: *idt* => *nat* => *nat* => *nat*
(‹*SUMM* -<-. -› [*0*, *0*, *10*] *10*)
syntax-consts
-sum == *sum*
translations
SUMM j<k. b == *CONST sum* (λ*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 Γ⊢ {|*True*|}
´S :== 0;; ´I :== 1;;
WHILE ´I ≠ n
DO
´S :== ´S + ´I;;
´I :== ´I + 1
OD
{|*´S* = (*SUMM j<n. j*)|}
(**is** Γ⊢ - (-;; *?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 Γ⊢ ⦃True⦄
          ´S :== 0;; ´I :== 1;;
          WHILE ´I ≠ n
          INV ⦃´S = (SUMM j<´I. j)⦄
          DO
            ´S :== ´S + ´I;;
            ´I :== ´I + 1
          OD
         ⦃´S = (SUMM j<n. j)⦄
⟨proof⟩
```

Certainly, this proof may be done fully automatically as well, provided that the invariant is given beforehand.

```
theorem (in state-space)
  shows Γ⊢ ⦃True⦄
          ´S :== 0;; ´I :== 1;;
          WHILE ´I ≠ n
          INV ⦃´S = (SUMM j<´I. j)⦄
          DO
            ´S :== ´S + ´I;;
            ´I :== ´I + 1
          OD
          ⦃´S = (SUMM j<n. j)⦄
  ⟨proof⟩
```

18.5 SWITCH

```
lemma (in state-space)
  shows Γ⊢ ⦃´N = 5⦄ SWITCH ´B
                     {True} ⇒ ´N :== 6
                   | {False} ⇒ ´N :== 7
                  END
         ⦃´N > 5⦄
⟨proof⟩
```

```
lemma (in state-space)
  shows Γ⊢ ⦃´N = 5⦄ SWITCH ´N
                     {v. v < 5} ⇒ ´N :== 6
                   | {v. v ≥ 5} ⇒ ´N :== 7
                  END
         ⦃´N > 5⦄
⟨proof⟩
```

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* ⇒ *nat*
where
fac 0 = *1* |
fac (*Suc n*) = (*Suc n*) ∗ *fac n*

lemma *fac-simp* [*simp*]: *0* < *i* ⟹ *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 (λ*s*. *s*(|*locals* := *update project-Nat-nat inject-Nat-nat N-′Fac-′* (*K-statefun* (*lookup project-Nat-nat N-′Fac-′* (*locals s*))) (*locals s*)|)) *Fac-′proc* (λ*s t*. *s*(|*globals* := *globals t*|)) (λ*i t*. *´R* :== *lookup project-Nat-nat R-′Fac-′* (*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 ∀σ. Γ,Θ⊢{σ} *PROC Fac*(*´N*,*´R*) {|*´R* = *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):

$[\![\forall Z.\ \Gamma,\Theta \cup (\bigcup_Z \{(P\ Z,\ p,\ Q\ Z,\ A\ Z)\})\vdash_{/F} (P\ Z)\ \mathit{the}\ (\Gamma\ p)\ (Q\ Z),(A\ Z);$
$p \in \mathit{dom}\ \Gamma]\!]$
$\Longrightarrow \forall Z.\ \Gamma,\Theta\vdash_{/F} (P\ Z)\ \mathit{Call}\ p\ (Q\ Z),(A\ Z)$

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 $\forall\sigma.\ \Gamma,\Theta\vdash\{\sigma\}\ \acute{}R :== \mathit{PROC}\ \mathit{Fac}(\acute{}N)\ \{\!|\acute{}R = \mathit{fac}\ {}^{\sigma}N|\!\}$
  ⟨*proof*⟩

Here some Isar style version of the proof

lemma (**in** *Fac-impl*) *Fac-spec2*:

  shows $\forall\sigma.\ \Gamma,\Theta\vdash\{\sigma\}\ \acute{}R :== \mathit{PROC}\ \mathit{Fac}(\acute{}N)\ \{\!|\acute{}R = \mathit{fac}\ {}^{\sigma}N|\!\}$
⟨*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*) *Fac-spec3*:
  shows $\forall\sigma.\ \Gamma,\Theta\vdash\{\sigma\}\ \acute{}R :== \mathit{PROC}\ \mathit{Fac}(\acute{}N)\ \{\!|\acute{}R = \mathit{fac}\ {}^{\sigma}N|\!\}$
  (**is** $\forall\sigma.\ \Gamma,\Theta\vdash(?Pre\ \sigma)\ ?Fac\ (?Post\ \sigma)$)
⟨*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-spec4*:
  shows $\forall\sigma.\ \Gamma,\Theta\vdash\{\sigma\}\ \acute{}R :== \mathit{PROC}\ \mathit{Fac}(\acute{}N)\ \{\!|\acute{}R = \mathit{fac}\ {}^{\sigma}N|\!\}$
  (**is** $\forall\sigma.\ \Gamma,\Theta\vdash(?Pre\ \sigma)\ ?Fac\ (?Post\ \sigma)$)
⟨*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 *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. Thats 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:

⟨*ML*⟩

lemma (**in** *odd-even-clique*)
shows *odd-spec*: $\forall\sigma.\ \Gamma\vdash\{\sigma\}$ *´A :== PROC odd*(*´N*)
$\{\!|(\exists b.\ {}^{\sigma}N = 2 * b + {}^{\prime}A) \wedge {}^{\prime}A < 2\ |\!\}$ (**is** *?P1*)
and *even-spec*: $\forall\sigma.\ \Gamma\vdash\{\sigma\}$ *´A :== PROC even*(*´N*)
$\{\!|(\exists b.\ {}^{\sigma}N + 1 = 2 * b + {}^{\prime}A) \wedge {}^{\prime}A < 2\ |\!\}$ (**is** *?P2*)
⟨*proof*⟩

18.7 Expressions With Side Effects

lemma (**in** *state-space*) **shows** $\Gamma\vdash$ {|*True*|}
´N ≫ *n*. *´N* :== *´N* + *1* ≫
´M ≫ *m*. *´M* :== *´M* + *1* ≫

´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 $CALL\ append(\acute{}p,\acute{}q,\acute{}p{\rightarrow}\acute{}next)$
end
declare $[[hoare\text{-}use\text{-}call\text{-}tr' = true]]$

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 σ, 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 $\{\!|\sigma.\ \ldots|\!\}$ is a shorthand to fix the current state: $\{s.\ \sigma = s\ \ldots\}$.

lemma (**in** *append-impl*) *append-spec*:
  shows $\forall \sigma\ Ps\ Qs.\ \Gamma\vdash$
$$\{\!|\sigma.\ List\ \acute{}p\ \acute{}next\ Ps \wedge\ List\ \acute{}q\ \acute{}next\ Qs \wedge set\ Ps \cap set\ Qs = \{\}|\!\}$$
$$\acute{}p :== PROC\ append(\acute{}p,\acute{}q)$$
$$\{\!|List\ \acute{}p\ \acute{}next\ (Ps@Qs) \wedge (\forall x.\ x{\notin}set\ Ps \longrightarrow \acute{}next\ x = {}^{\sigma}next\ x)|\!\}$$
  $\langle proof\rangle$

The modifies clause is equal to a proper record update specification of the following form.

lemma (**in** *append-impl*) **shows** $\{t.\ t\ may\text{-}only\text{-}modify\text{-}globals\ Z\ in\ [next]\}$
$$=$$
$$\{t.\ \exists next.\ globals\ t{=}update\ id\ id\ next\text{-}'\ (K\text{-}statefun\ next)\ (globals\ Z)\}$$
  $\langle proof\rangle$

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*=*Null* ∧ ´*cont*=*c*|} ´*p* :== *CALL append*(´*p*,*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
∀ σ. Γ⊢ {σ} ´*p* :== *PROC append*(´*p*,´*q*){*t*. *t may-only-modify-globals* σ *in* [*next*]}
⟨*proof*⟩

lemma (**in** *append-impl*)
shows Γ⊢ {|´*p*=*Null* ∧ ´*cont*=*c*|} ´*p*→´*next* :== *CALL append*(´*p*,*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 me may need the modifies clause to prove the functional specifications.

18.8.1 Insertion Sort

primrec *sorted*:: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a list* ⇒ *bool*
where
sorted le [] = *True* |
sorted le (*x*#*xs*) = ((∀ *y*∈*set xs*. *le x y*) ∧ *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*→´*next* :== *Null*
ELSE IF ´*r*→´*cont* ≤ ´*p*→´*cont*
THEN ´*r*→´*next* :== ´*p*;; ´*p*:==´*r*
ELSE ´*p*→´*next* :== *CALL insert*(´*r*,´*p*→´*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 explicitely state an equality in the postcondition. The reason is simple. If the postcondition would talk about *´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. Thats 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*:
$\forall \sigma.\ \Gamma\vdash \{\sigma\}$ *´p :== PROC insert(´r,´p)*$\{t.$ *t may-only-modify-globals* σ *in* [*next*]$\}$
⟨*proof*⟩

lemma (**in** *insert-impl*) *insert-spec*:
$\forall \sigma\ Ps\ .\ \Gamma\vdash$ ⦃σ. *List ´p ´next Ps* $\wedge$ *sorted* $(\le)$ (*map ´cont Ps*) $\wedge$
´r $\neq$ *Null* $\wedge$ *´r* $\notin$ *set Ps*⦄
´p :== PROC insert(´r,´p)
⦃$\exists$ *Qs*. *List ´p ´next Qs* $\wedge$ *sorted* $(\le)$ (*map* $^{\sigma}$*cont Qs*) $\wedge$
set Qs = *insert* $^{\sigma}$*r* (*set Ps*) $\wedge$
($\forall x.\ x \notin$ *set Qs* $\longrightarrow$ *´next x* = $^{\sigma}$*next x*)⦄

⟨*proof*⟩

procedures (**imports** *globals-list*)
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
```

print-locale *insertSort-impl*

lemma (**in** *insertSort-impl*) *insertSort-modifies*:

shows
∀σ. Γ⊢ {σ} ´p :== *PROC insertSort*(´p)
{t. t *may-only-modify-globals* σ *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 *HoareTotal.annotateI*.

lemma (**in** *insertSort-impl*) *insertSort-body-spec*:
shows ∀σ *Ps*. Γ,Θ⊢ ⦃σ. *List* ´p ´*next Ps* ⦄
´p :== *PROC insertSort*(´p)
⦃∃ *Qs*. *List* ´p ´*next Qs* ∧ *sorted* (≤) (*map* $^{\sigma}$*cont Qs*) ∧
set Qs = *set Ps*⦄
⟨*proof*⟩

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* =
alloc::*ref list*
free::*nat*
next::*ref* ⇒ *ref*
cont::*ref* ⇒ *nat*
hoarestate *locals-list-alloc* =
i::*nat*
first::*ref*
p::*ref*
q::*ref*
r::*ref*
root::*ref*
tmp::*ref*
locale *list-alloc* = *globals-list-alloc* + *locals-list-alloc*

definition *sz* = (*2*::*nat*)

lemma (**in** *list-alloc*)
shows
Γ,Θ⊢ ⦃´i = *0* ∧ ´*first* = *Null* ∧ n∗*sz* ≤ ´*free*⦄
WHILE ´i < n
INV ⦃∃ *Ps*. *List* ´*first* ´*next Ps* ∧ *length Ps* = ´i ∧ ´i ≤ n ∧
set Ps ⊆ *set* ´*alloc* ∧ (n − ´i)∗*sz* ≤ ´*free*⦄
DO
´p :== *NEW sz* [´*cont*:==*0*,´*next*:== *Null*];;

´p→´next :== ´first;;
´first :== ´p;;
´i :== ´i+ 1
OD
{|∃ Ps. List ´first ´next Ps ∧ length Ps = n ∧ set Ps ⊆ set ´alloc|}
⟨proof⟩

lemma (**in** *list-alloc*)
shows
Γ⊢ {|´i = 0 ∧ ´first = Null ∧ n∗sz ≤ ´free|}
WHILE ´i < n
INV {|∃ Ps. List ´first ´next Ps ∧ length Ps = ´i ∧ ´i ≤ n ∧
set Ps ⊆ set ´alloc ∧ (n − ´i)∗sz ≤ ´free|}
DO
´p :== NNEW sz [´cont:==0,´next:== Null];;
´p→´next :== ´first;;
´first :== ´p;;
´i :== ´i+ 1
OD
{|∃ Ps. List ´first ´next Ps ∧ length Ps = n ∧ set Ps ⊆ set ´alloc|}

⟨proof⟩

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 precondtion.

lemma (**in** *list-alloc*) Γ,Θ⊢ {|True|} {|´p≠Null|}⟼ ´p→´next :== ´p {|True|}
⟨proof⟩

lemma (**in** *list-alloc*) Γ,Θ⊢ {} {|´p≠Null|}⟼ ´p→´next :== ´p {|True|}
⟨proof⟩

Let us consider this small program that reverts a list. At first without guards.

lemma (**in** *list-alloc*)
shows
Γ,Θ⊢ {|List ´p ´next Ps ∧ List ´q ´next Qs ∧ set Ps ∩ set Qs = {} ∧
set Ps ⊆ set ´alloc ∧ set Qs ⊆ set ´alloc|}
WHILE ´p ≠ Null
INV {|∃ ps qs. List ´p ´next ps ∧ List ´q ´next qs ∧ set ps ∩ set qs = {} ∧
rev ps @ qs = rev Ps @ Qs ∧
set ps ⊆ set ´alloc ∧ set qs ⊆ set ´alloc|}
DO ´r :== ´p;;
´p :== ´p→ ´next;;

´r→´next :== ´q;;
´q :== ´r OD
⦃List ´q ´next (rev Ps @ Qs) ∧ set Ps⊆ set ´alloc ∧ set Qs ⊆ set ´alloc⦄
⟨proof⟩

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
Γ,Θ⊢ ⦃List ´p ´next Ps ∧ List ´q ´next Qs ∧ set Ps ∩ set Qs = {} ∧
set Ps ⊆ set ´alloc ∧ set Qs ⊆ set ´alloc⦄
WHILE ´p ≠ Null
INV ⦃∃ps qs. List ´p ´next ps ∧ List ´q ´next qs ∧ set ps ∩ set qs = {} ∧
rev ps @ qs = rev Ps @ Qs ∧
set ps ⊆ set ´alloc ∧ set qs ⊆ set ´alloc⦄
DO ´r :== ´p;;
⦃´p≠Null ∧ ´p∈set ´alloc⦄⟼ ´p :== ´p→ ´next;;
⦃´r≠Null ∧ ´r∈set ´alloc⦄⟼ ´r→´next :== ´q;;
´q :== ´r OD
⦃List ´q ´next (rev Ps @ Qs) ∧ set Ps ⊆ set ´alloc ∧ set Qs ⊆ set ´alloc⦄
⟨proof⟩

We can also just prove that no faults will occur, by giving the trivial postcondition.

lemma (**in** *list-alloc*) *rev-noFault*:
shows
Γ,Θ⊢ ⦃List ´p ´next Ps ∧ List ´q ´next Qs ∧ set Ps ∩ set Qs = {} ∧
set Ps ⊆ set ´alloc ∧ set Qs ⊆ set ´alloc⦄
WHILE ´p ≠ Null
INV ⦃∃ps qs. List ´p ´next ps ∧ List ´q ´next qs ∧ set ps ∩ set qs = {} ∧
rev ps @ qs = rev Ps @ Qs ∧
set ps ⊆ set ´alloc ∧ set qs ⊆ set ´alloc⦄
DO ´r :== ´p;;
⦃´p≠Null ∧ ´p∈set ´alloc⦄⟼ ´p :== ´p→ ´next;;
⦃´r≠Null ∧ ´r∈set ´alloc⦄⟼ ´r→´next :== ´q;;
´q :== ´r OD
UNIV,UNIV
⟨proof⟩

lemma (**in** *list-alloc*) *rev-moduloGuards*:

shows
Γ,Θ$\vdash_{/\{True\}}$ ⦃List ´p ´next Ps ∧ List ´q ´next Qs ∧ set Ps ∩ set Qs = {} ∧
set Ps ⊆ set ´alloc ∧ set Qs ⊆ set ´alloc⦄
WHILE ´p ≠ Null
INV ⦃∃ps qs. List ´p ´next ps ∧ List ´q ´next qs ∧ set ps ∩ set qs = {} ∧
rev ps @ qs = rev Ps @ Qs ∧
set ps ⊆ set ´alloc ∧ set qs ⊆ set ´alloc⦄
DO ´r :== ´p;;

$\{\!|$ ´$p \neq Null \wedge$ ´$p \in set$ ´$alloc|\!\}\surd \longmapsto$ ´p :== ´$p\rightarrow$ ´$next$;;
$\{\!|$ ´$r \neq Null \wedge$ ´$r \in set$ ´$alloc|\!\}\surd \longmapsto$ ´$r\rightarrow$´$next$:== ´q;;
´q :== ´r *OD*
$\{\!|$*List* ´*q* ´*next* (*rev Ps* @ *Qs*) $\wedge$ *set Ps* $\subseteq$ *set* ´*alloc* $\wedge$ *set Qs* $\subseteq$ *set* ´*alloc*$|\!\}$
⟨*proof*⟩

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-guards False* (*strip-guards* ($-F$) *c'*)
assumes *c*: *c* = *mark-guards False c'*
shows $\Gamma,\Theta\vdash_{/\{\}}$ *P c Q,A*
⟨*proof*⟩

We can then combine the prove that no fault will occur with the functional prove of the programm without guards to get the full proove by the rule $[\![?\Gamma,?\Theta\vdash_{/?F}\ ?P\ ?c\ ?Q,?A;\ ?\Gamma,?\Theta\vdash\ ?P\ strip\text{-}guards\ (-\ ?F)\ ?c\ UNIV,UNIV]\!] \Longrightarrow\ ?\Gamma,?\Theta\vdash\ ?P\ ?c\ ?Q,?A$

lemma (**in** *list-alloc*)
shows
$\Gamma,\Theta\vdash$ $\{\!|$*List* ´*p* ´*next Ps* $\wedge$ *List* ´*q* ´*next Qs* $\wedge$ *set Ps* $\cap$ *set Qs* = {} $\wedge$
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* = {} $\wedge$
rev ps @ *qs* = *rev Ps* @ *Qs* $\wedge$
set ps $\subseteq$ *set* ´*alloc* $\wedge$ *set qs* $\subseteq$ *set* ´*alloc*$|\!\}$
DO ´*r* :== ´*p*;;
$\{\!|$ ´$p \neq Null \wedge$ ´$p \in set$ ´$alloc|\!\} \longmapsto$ ´p :== ´$p\rightarrow$ ´$next$;;
$\{\!|$ ´$r \neq Null \wedge$ ´$r \in set$ ´$alloc|\!\} \longmapsto$ ´$r\rightarrow$´$next$:== ´q;;
´q :== ´r *OD*
$\{\!|$*List* ´*q* ´*next* (*rev Ps* @ *Qs*) $\wedge$ *set Ps* $\subseteq$ *set* ´*alloc* $\wedge$ *set Qs* $\subseteq$ *set* ´*alloc*$|\!\}$

⟨*proof*⟩

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* ≠ *Null*|})⟼ ´*p*→´*next* ≠ *Null DO SKIP OD*

end

18.10 Cicular Lists

definition
  distPath :: *ref* ⇒ (*ref* ⇒ *ref*) ⇒ *ref* ⇒ *ref list* ⇒ *bool* **where**
  distPath x next y as = (*Path x next y as* ∧ *distinct as*)

lemma *neq-dP*: ⟦*p* ≠ *q*; *Path p h q Ps*; *distinct Ps*⟧ ⟹
 ∃ *Qs*. *p*≠*Null* ∧ *Ps* = *p*#*Qs* ∧ *p* ∉ *set Qs*
⟨*proof*⟩

lemma (**in** *list-alloc*) *circular-list-rev-I*:
  Γ,Θ⊢ {|´*root* = *r* ∧ *distPath* ´*root* ´*next* ´*root* (*r*#*Ps*)|}
   ´*p* :== ´*root*;; ´*q* :== ´*root*→´*next*;;
  WHILE ´*q* ≠ ´*root*
  INV {|∃ *ps qs*. *distPath* ´*p* ´*next* ´*root ps* ∧ *distPath* ´*q* ´*next* ´*root qs* ∧
        ´*root* = *r* ∧ *r*≠*Null* ∧ *r* ∉ *set Ps* ∧ *set ps* ∩ *set qs* = {} ∧
        Ps = (*rev ps*) @ *qs* |}
  DO ´*tmp* :== ´*q*;; ´*q* :== ´*q*→´*next*;; ´*tmp*→´*next* :== ´*p*;; ´*p*:==´*tmp OD*;;
  ´*root*→´*next* :== ´*p*
  {|´*root* = *r* ∧ *distPath* ´*root* ´*next* ´*root* (*r*#*rev Ps*)|}
⟨*proof*⟩

lemma *path-is-list*:⋀*a next b*. ⟦*Path b next a Ps* ; *a* ∉ *set Ps*; *a*≠*Null*⟧
⟹ *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*:
 Γ,Θ⊢
 {|´*p* = *r* ∧ *distPath* ´*p* ´*next* ´*p* (*r*#*Ps*)|}
´*q*:==*Null*;;
WHILE ´*p* ≠ *Null*
INV
 {| ((´*q* = *Null*) ⟶ (∃ *ps*. *distPath* ´*p* ´*next r ps* ∧ *ps* = *r*#*Ps*)) ∧
  ((´*q* ≠ *Null*) ⟶ (∃ *ps qs*. *distPath* ´*q* ´*next r qs* ∧ *List* ´*p* ´*next ps* ∧
           set ps ∩ *set qs* = {} ∧ *rev qs* @ *ps* = *Ps*@[*r*])) ∧
  ¬ (´*p* = *Null* ∧ ´*q* = *Null* ∧ *r* = *Null*)
  |}

DO
  $\acute{}tmp :== \acute{}p;;\ \acute{}p :== \acute{}p \rightarrow \acute{}next;;\ \acute{}tmp \rightarrow \acute{}next :== \acute{}q;;\ \acute{}q :== \acute{}tmp$
OD
$\{\!|\acute{}q = r \wedge distPath\ \acute{}q\ \acute{}next\ \acute{}q\ (r \# rev\ Ps)|\!\}$

⟨*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 (**in** *state-space*) $\Gamma \vdash \{\sigma\}$
$\acute{}I :== \acute{}M;;$
$ANNO\ \tau.\ \{\!|\tau.\ \acute{}I = {}^{\sigma}M|\!\}$
$\acute{}M :== \acute{}N;;\ \acute{}N :== \acute{}I$
$\{\!|\acute{}M = {}^{\tau}N \wedge \acute{}N = {}^{\tau}I|\!\}$
$\{\!|\acute{}M = {}^{\sigma}N \wedge \acute{}N = {}^{\sigma}M|\!\}$
⟨*proof*⟩

context *state-space*
begin
term $ANNO\ (\tau,m,k).\ (\{\!|\tau.\ \acute{}M = m|\!\})\ \acute{}M :== \acute{}N;;\ \acute{}N :== \acute{}I\ \{\!|\acute{}M = {}^{\tau}N\ \&\ \acute{}N = {}^{\tau}I|\!\},\{\}$
end

lemma (**in** *state-space*) $\Gamma \vdash (\{\sigma\} \cap \{\!|\acute{}M = 0 \wedge \acute{}S = 0|\!\})$
$(ANNO\ \tau.\ (\{\tau\} \cap \{\!|\acute{}A = {}^{\sigma}A \wedge \acute{}I = {}^{\sigma}I \wedge \acute{}M = 0 \wedge \acute{}S = 0|\!\})$
$WHILE\ \acute{}M \neq \acute{}A$
$INV\ \{\!|\acute{}S = \acute{}M * \acute{}I \wedge \acute{}A = {}^{\tau}A \wedge \acute{}I = {}^{\tau}I|\!\}$
$DO\ \acute{}S :== \acute{}S + \acute{}I;;\ \acute{}M :== \acute{}M + 1\ OD$
$\{\!|\acute{}S = {}^{\tau}A * {}^{\tau}I|\!\})$
$\{\!|\acute{}S = {}^{\sigma}A * {}^{\sigma}I|\!\}$
⟨*proof*⟩

Just some test on marked, guards

lemma (**in** *state-space*) $\Gamma \vdash \{\!|True|\!\}\ WHILE\ \{\!|P\ \acute{}N\ |\!\}\surd,\ \{\!|Q\ \acute{}M|\!\}\#,\ \{\!|R\ \acute{}N|\!\} \longmapsto \acute{}N < \acute{}M$
$INV\ \{\!|\acute{}N < 2|\!\}\ DO$
$\acute{}N :== \acute{}M$
OD
$\{\!|hard|\!\}$
⟨*proof*⟩

lemma (**in** *state-space*) $\Gamma \vdash_{/\{True\}} \{\!|True|\!\}\ WHILE\ \{\!|P\ \acute{}N\ |\!\}\surd,\ \{\!|Q\ \acute{}M|\!\}\#,\ \{\!|R\ \acute{}N|\!\} \longmapsto \acute{}N < \acute{}M$

```
                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 Γ⊢t {|´M = 0 ∧ ´S = 0|}
          WHILE ´M ≠ a
          INV {|´S = ´M ∗ b ∧ ´M ≤ a|}
          VAR MEASURE a − ´M
          DO ´S :== ´S + b;; ´M :== ´M + 1 OD
          {|´S = a ∗ b|}
⟨proof⟩
```

```
lemma Γ⊢t {|´I ≤ 3|}
     WHILE ´I < 10 INV {|´I≤ 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 Γ⊢t {|´M=0 ∧ ´N=0|}
      WHILE (´M < i)
      INV {|´M ≤ i ∧ (´M ≠ 0 ⟶ ´N = j) ∧ ´N ≤ j|}
      VAR MEASURE (i − ´M)
      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*⟩

primrec *fac*:: *nat* ⇒ *nat*
where
fac 0 = *1* |
fac (*Suc n*) = (*Suc n*) ∗ *fac n*

lemma *fac-simp* [*simp*]: *0* < *i* ⟹ *fac i* = *i* ∗ *fac* (*i* − *1*)
⟨*proof*⟩

procedures
Fac (*N* | *R*) = *IF* ´*N* = *0 THEN* ´*R* :== *1*
ELSE CALL Fac(´*N* − *1*,´*R*);;
´*R* :== ´*N* ∗ ´*R*
FI

lemma (**in** *Fac-impl*) *Fac-spec*:
shows ∀ *n*. Γ⊢$_t$ {|´*N*=*n*|} ´*R* :== *PROC Fac*(´*N*) {|´*R* = *fac n*|}
⟨*proof*⟩

procedures
p91(*R*,*N* | *R*) = *IF 100* < ´*N THEN* ´*R* :== ´*N* − *10*
ELSE ´*R* :== *CALL p91*(´*R*,´*N*+*11*);;
´*R* :== *CALL p91*(´*R*,´*R*) *FI*

p91-spec: ∀ *n*. Γ⊢$_t$ {|´*N*=*n*|} ´*R* :== *PROC p91*(´*R*,´*N*)
{|*if 100* < *n then* ´*R* = *n* − *10 else* ´*R* = *91*|},{}

lemma (**in** *p91-impl*) *p91-spec*:
shows ∀ σ. Γ⊢$_t$ {σ} ´*R* :== *PROC p91*(´*R*,´*N*)
{|*if 100* < ^{σ}*N then* ´*R* = ^{σ}*N* − *10 else* ´*R* = *91*|},{}
⟨*proof*⟩

record *globals-list* =
next-′ :: *ref* ⇒ *ref*
cont-′ :: *ref* ⇒ *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*. $\Gamma\vdash_t$
    ⦃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* = $^{\sigma}$*next x*)⦄
  ⟨*proof*⟩

lemma (**in** *append-impl*)
  shows
  $\forall \sigma$ *Ps Qs*. $\Gamma\vdash_t$
    ⦃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* = $^{\sigma}$*next x*)⦄
  ⟨*proof*⟩

lemma (**in** *append-impl*)
  shows
  append-spec:
  ∀ σ. $\Gamma\vdash_t$ ({σ} ∩ ⦃*islist ´p ´next*⦄) *´p :== PROC append(´p,´q)*
   ⦃∀ *Ps Qs*. *List* $^{\sigma}$*p* $^{\sigma}$*next Ps* ∧ *List* $^{\sigma}$*q* $^{\sigma}$*next Qs* ∧ *set Ps* ∩ *set Qs* = {}
    ⟶
    List ´p ´next (Ps@Qs) ∧ (∀ *x*. *x*∉*set Ps* ⟶ *´next x* = $^{\sigma}$*next x*)⦄
  ⟨*proof*⟩

lemma $\Gamma\vdash$⦃*List ´p ´next Ps*⦄
    ´q :== Null;;
    WHILE ´p ≠ Null INV ⦃∃ *Ps' Qs'*. *List ´p ´next Ps'* ∧ *List ´q ´next Qs'* ∧
                set Ps' ∩ *set Qs'* = {} ∧
                rev Ps' @ *Qs'* = *rev Ps*⦄
     DO
      ´r :== ´p;; *´p :== ´p→´next*;;
      ´r→´next :== ´q;; *´q :== ´r*
    OD;;
    ´p :==´q
    ⦃*List ´p ´next (rev Ps)*⦄

⟨proof⟩

lemma *conjI2*: $[\![Q;\ Q \Longrightarrow P]\!] \Longrightarrow P \wedge Q$
⟨proof⟩

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:
∀ Ps. $\Gamma \vdash_t$ {|List ´p ´next Ps|} ´p :== PROC Rev(´p) {|List ´p ´next (rev Ps)|}
Rev-modifies:
∀ σ. $\Gamma \vdash_{/UNIV}$ {σ} ´p :== PROC Rev(´p) {t. t may-only-modify-globals σ in [next]}

We only need partial correctness of modifies clause!

lemma *upd-hd-next*:
assumes *p-ps*: *List p next* (*p#ps*)
shows *List* (*next p*) (*next*(*p* := *q*)) *ps*
⟨proof⟩

lemma (**in** *Rev-impl*) **shows**
Rev-spec:
∀ Ps. $\Gamma \vdash_t$ {|List ´p ´next Ps|} ´p :== PROC Rev(´p) {|List ´p ´next (rev Ps)|}
⟨proof⟩

lemma (**in** *Rev-impl*) **shows**
Rev-modifies:
∀ σ. $\Gamma \vdash_{/UNIV}$ {σ} ´p :== PROC Rev(´p) {t. t may-only-modify-globals σ in [next]}
⟨proof⟩

lemma $\Gamma \vdash_t$ {|List ´p ´next Ps|}
´q :== Null;;
WHILE ´p ≠ Null INV {|∃ Ps′ Qs′. List ´p ´next Ps′ ∧ List ´q ´next Qs′ ∧
set Ps′ ∩ set Qs′ = {} ∧
rev Ps′ @ Qs′ = 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*⦄) ∧
($\Gamma\vdash_t$ ⦃*True*⦄ *PROC coast*(´*N*,´*M*) ⦃*True*⦄)
⟨*proof*⟩

lemma (**in** *pedal-coast-clique*)
shows ($\Gamma\vdash_t$ ⦃*True*⦄ *PROC pedal*(´*N*,´*M*) ⦃*True*⦄) ∧
($\Gamma\vdash_t$ ⦃*True*⦄ *PROC coast*(´*N*,´*M*) ⦃*True*⦄)
⟨*proof*⟩

lemma (**in** *pedal-coast-clique*)
shows ($\Gamma\vdash_t$ ⦃*True*⦄ *PROC pedal*(´*N*,´*M*) ⦃*True*⦄) ∧
($\Gamma\vdash_t$ ⦃*True*⦄ *PROC coast*(´*N*,´*M*) ⦃*True*⦄)
⟨*proof*⟩

lemma (**in** *pedal-coast-clique*)
shows ($\Gamma\vdash_t$ ⦃*True*⦄ *PROC pedal*(´*N*,´*M*) ⦃*True*⦄) ∧
($\Gamma\vdash_t$ ⦃*True*⦄ *PROC coast*(´*N*,´*M*) ⦃*True*⦄)
⟨*proof*⟩

lemma (**in** *pedal-coast-clique*)
shows ($\Gamma\vdash_t$ ⦃*True*⦄ *PROC pedal*(´*N*,´*M*) ⦃*True*⦄) ∧
($\Gamma\vdash_t$ ⦃*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* ⇒ *ref*
  cont-′ :: *ref* ⇒ *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→´next :== CALL append(´p→´next,´q)
FI

  append-spec:
  ∀ 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* = $^{\sigma}$*next x*)|}

  append-modifies:
  ∀ σ. Γ⊢ {σ} *´p :== PROC append(´p,´q)*{*t*. *t may-only-modify-globals* σ *in* [*next*]}

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* = $^{\sigma}$*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*$\in$*set xs*. *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* $\in$ *set xs*. $\forall$ *y* $\in$ *set ys*. *le x y*))
⟨*proof*⟩

procedures *quickSort*(*p*|*p*) =
```
IF ´p=Null THEN SKIP
 ELSE ´tl :== ´p→´next;;
      ´le :== Null;;
      ´gt :== Null;;
      WHILE ´tl≠Null DO
        ´hd :== ´tl;;
        ´tl :== ´tl→´next;;
        IF ´hd→´cont ≤ ´p→´cont
        THEN ´hd→´next :== ´le;;
             ´le :== ´hd
        ELSE ´hd→´next :== ´gt;;
             ´gt :== ´hd
        FI
      OD;;
      ´le :== CALL quickSort(´le);;
      ´gt :== CALL quickSort(´gt);;
      ´p→´next :== ´gt;;
      ´le :== CALL append(´le,´p);;
      ´p :== ´le
 FI
```

quickSort-spec:
$\forall$ σ *Ps*. Γ⊢ {|σ. *List* ´*p* ´*next Ps*|} ´*p* :== *PROC quickSort*(´*p*)
{|($\exists$ *sortedPs*. *List* ´*p* ´*next sortedPs* $\wedge$
sorted (≤) (*map* $^{\sigma}$*cont sortedPs*) $\wedge$
mset Ps = *mset sortedPs*) $\wedge$
($\forall$ *x*. *x*$\notin$*set Ps* $\longrightarrow$ ´*next x* = $^{\sigma}$*next x*)|}

quickSort-modifies:
$\forall$ σ. Γ⊢ {σ} ´*p* :== *PROC quickSort*(´*p*) {*t*. *t may-only-modify-globals* σ *in* [*next*]}

lemma (**in** *quickSort-impl*) *quickSort-modifies*:
shows
$\forall$ σ. Γ⊢ {σ} ´*p* :== *PROC quickSort*(´*p*) {*t*. *t may-only-modify-globals* σ *in* [*next*]}
⟨*proof*⟩

lemma (**in** *quickSort-impl*) *quickSort-spec*:
shows
$\forall\sigma$ *Ps*. $\Gamma\vdash$ ⦃σ. *List ´p ´next Ps*⦄
´p :== *PROC quickSort*(*´p*)
⦃($\exists$ *sortedPs*. *List ´p ´next sortedPs* $\wedge$
sorted ($\leq$) (*map* $^{\sigma}$*cont sortedPs*) $\wedge$
mset Ps = *mset sortedPs*) $\wedge$
($\forall$ *x*. *x*$\notin$*set Ps* $\longrightarrow$ *´next x* = $^{\sigma}$*next x*)⦄
⟨*proof*⟩

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*] => *basicblock*
(‹(‹*notation*=‹*mixfix LET expression*››*LET* (-)/ *IN* (-))› *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*
⟨*proof*⟩

lemma *Let′-split-conv* [*vcg-simp*]:
(*Let′ x* (λ*p*. (*case-prod* (*f p*) (*g p*)))) =
(*Let′ x* (λ*p*. (*f p*) (*fst* (*g p*)) (*snd* (*g p*))))
⟨*proof*⟩

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 Γ⊢ {|´A = 0|}
     {|´A < 0|} ⟼ BASIC
      LET (a,b,c) = foo ´A
      IN
          ´A :== a,
          ´B :== b,
          ´C :== c
     END
     {|´A = x ∧ ´B = y ∧ ´C = c|}
⟨proof⟩

lemma Γ⊢ {|´A = 0|}
```

$\{\!|\, ´A < 0 \,|\!\} \longmapsto BASIC$
$LET\ (a,b,c) = foo\ ´A$
IN
$´A :== a,$
$´G :== b + ´B,$
$´H :== c$
END
$\{\!|\, ´A = x \wedge ´G = y \wedge ´H = c \,|\!\}$
⟨*proof*⟩

definition *foo*:: $nat \Rightarrow (nat \times nat \times nat)$
where $foo\ n = (n,n+1,n+2)$

lemma $\Gamma\vdash \{\!|\, ´A = 0 \,|\!\}$
$\{\!|\, ´A < 0 \,|\!\} \longmapsto BASIC$
$LET\ (a,b,c) = foo\ ´A$
IN
$´A :== a,$
$´G :== b + ´B,$
$´H :== c$
END
$\{\!|\, ´A = x \wedge ´G = y \wedge ´H = c \,|\!\}$
⟨*proof*⟩

end

22 Examples for Procedures as Parameters

theory *ProcParEx* **imports** ../*Vcg* **begin**

lemma *DynProcProcPar′*:
assumes *adapt*: $P \subseteq \{s.\ p\ s = q \wedge$
$(\exists Z.\ init\ s \in P'\ Z \wedge$
$(\forall t \in Q'\ Z.\ return\ s\ t \in R\ s\ t) \wedge$
$(\forall t \in A'\ Z.\ return\ s\ t \in A))\}$
assumes *result*: $\forall s\ t.\ \Gamma,\Theta\vdash_{/F} (R\ s\ t)\ result\ s\ t\ Q,A$
assumes *q*: $\forall Z.\ \Gamma,\Theta\vdash_{/F} (P'\ Z)\ Call\ q\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{/F} P\ dynCall\ init\ p\ return\ result\ Q,A$
⟨*proof*⟩

lemma *conseq-exploit-pre′*:
$[\![\forall s \in S.\ \Gamma,\Theta \vdash (\{s\} \cap P)\ c\ Q,A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash (P \cap S) c\ Q,A$
⟨*proof*⟩

lemma *conseq-exploit-pre''*:
$[\![\forall Z.\ \forall s \in S\ Z.\ \ \Gamma,\Theta \vdash (\{s\} \cap P\ Z)\ c\ (Q\ Z),(A\ Z)]\!]$
$\Longrightarrow$
$\forall Z.\ \Gamma,\Theta\vdash\ (P\ Z \cap S\ Z) c\ (Q\ Z),(A\ Z)$
⟨*proof*⟩

lemma *conseq-exploit-pre'''*:
$[\![\forall s \in S.\ \forall Z.\ \Gamma,\Theta \vdash (\{s\} \cap P\ Z)\ c\ (Q\ Z),(A\ Z)]\!]$
$\Longrightarrow$
$\forall Z.\ \Gamma,\Theta\vdash\ (P\ Z \cap S) c\ (Q\ Z),(A\ Z)$
⟨*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: $\forall\sigma.\ \Gamma\vdash\ \{\sigma\}$ *PROC LEQ(´n,´m,´b)* $\{\!|$ *´b* $= ({}^{\sigma}n \le {}^{\sigma}m)|\!\}$
LEQ-modifies: $\forall\sigma.\ \Gamma\vdash\ \{\sigma\}$ *PROC LEQ(´n,´m,´b)* $\{t.$ *t may-only-modify-globals* σ *in* []}

definition *mx*:: *('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ 'a*
where *mx leq a b = (if leq a b then a else b)*

procedures
Max (*compare*, *n*, *m* | *k*) =
´b :== *DYNCALL ´compare*(*´n*,*´m*);;
IF ´b THEN ´k :== *´n ELSE ´k* :== *´m FI*

Max-spec: $\bigwedge$*leq*. ∀ σ. Γ⊢
({σ} ∩ {*s*. (∀ τ. Γ⊢ {τ} *´b* :== *PROC* ^{s}*compare*(*´n*,*´m*) {|*´b* = (*leq* ^{τ}*n* ^{τ}*m*)|}) ∧
(∀ τ. Γ⊢ {τ} *´b* :== *PROC* ^{s}*compare*(*´n*,*´m*) {*t*. *t may-only-modify-globals* τ *in* []})})
PROC Max(*´compare*,*´n*,*´m*,*´k*)
{|*´k* = *mx leq* ^{σ}*n* ^{σ}*m*|}

lemma (**in** *Max-impl*) *Max-spec1*:
shows
∀ σ *leq*. Γ⊢
({σ} ∩ {| (∀ τ. Γ⊢{τ} *´b* :== *PROC ´compare*(*´n*,*´m*) {|*´b* = (*leq* ^{τ}*n* ^{τ}*m*)|}) ∧
(∀ τ. Γ⊢ {τ} *´b* :== *PROC ´compare*(*´n*,*´m*) {*t*. *t may-only-modify-globals* τ *in* []})|})
´k :== *PROC Max*(*´compare*,*´n*,*´m*)
{|*´k* = *mx leq* ^{σ}*n* ^{σ}*m*|}
⟨*proof*⟩

lemma (**in** *Max-impl*) *Max-spec2*:
shows
∀ σ *leq*. Γ⊢
({σ} ∩ {|(∀ τ. Γ⊢ {τ} *´b* :== *PROC ´compare*(*´n*,*´m*) {|*´b* = (*leq* ^{τ}*n* ^{τ}*m*)|}) ∧
(∀ τ. Γ⊢ {τ} *´b* :== *PROC ´compare*(*´n*,*´m*) {*t*. *t may-only-modify-globals* τ *in* []})|})
´k :== *PROC Max*(*´compare*,*´n*,*´m*)
{|*´k* = *mx leq* ^{σ}*n* ^{σ}*m*|}
⟨*proof*⟩

lemma (**in** *Max-impl*) *Max-spec3*:
shows
∀ *n m leq*. Γ⊢
({|*´n*=*n* ∧ *´m*=*m*|} ∩
{|(∀ τ. Γ⊢ {τ} *´b* :== *PROC ´compare*(*´n*,*´m*) {|*´b* = (*leq* ^{τ}*n* ^{τ}*m*)|}) ∧
(∀ τ. Γ⊢ {τ} *´b* :== *PROC ´compare*(*´n*,*´m*) {*t*. *t may-only-modify-globals* τ *in* []})|})
´k :== *PROC Max*(*´compare*,*´n*,*´m*)
{|*´k* = *mx leq n m*|}
⟨*proof*⟩

lemma (**in** *Max-impl*) *Max-spec4*:
shows
∀ *n m leq*. Γ⊢
({|*´n*=*n* ∧ *´m*=*m*|} ∩ {|∀ τ. Γ⊢ {τ} *´b* :== *PROC ´compare*(*´n*,*´m*) {|*´b* = (*leq* ^{τ}*n*

$^{\tau}m)\}\})$
  $\acute{}k$:== $PROC\ Max(\acute{}compare,\acute{}n,\acute{}m)$
 $\{\!|\ \acute{}k = mx\ leq\ n\ m|\!\}$
⟨*proof*⟩

locale *Max-test* = *Max-spec* + *LEQ-spec* + *LEQ-modifies*
lemma (**in** *Max-test*)

shows
$\Gamma\vdash \{\sigma\}\ \acute{}k :== CALL\ Max(LEQ\text{-}'proc,\acute{}n,\acute{}m)\ \{\!|\acute{}k = mx\ (\leq)\ ^{\sigma}n\ ^{\sigma}m|\!\}$
⟨*proof*⟩

lemma (**in** *Max-impl*) *Max-spec5*:
shows
$\forall n\ m\ leq.\ \Gamma\vdash$
$(\{\!|\acute{}n=n \wedge \acute{}m=m|\!\} \cap \{\!|\forall n'\ m'.\ \Gamma\vdash \{\!|\acute{}n=n' \wedge \acute{}m=m'|\!\}\ \acute{}b :== PROC\ \acute{}compare(\acute{}n,\acute{}m)\ \{\!|\acute{}b = (leq\ n'\ m')|\!\}|\!\})$
  $\acute{}k :== PROC\ Max(\acute{}compare,\acute{}n,\acute{}m)$
 $\{\!|\acute{}k = mx\ leq\ n\ m|\!\}$
term $\{\!|\{s.\ ^{s}n = n' \wedge\ ^{s}m = m'\} = X|\!\}$
⟨*proof*⟩

lemma (**in** *LEQ-impl*)
LEQ-spec: $\forall n\ m.\ \Gamma\vdash \{\!|\acute{}n=n \wedge \acute{}m=m|\!\}\ \ PROC\ LEQ(\acute{}n,\acute{}m,\acute{}b)\ \{\!|\acute{}b = (n \leq m)|\!\}$
⟨*proof*⟩

locale *Max-test'* = *Max-impl* + *LEQ-impl*
lemma (**in** *Max-test'*)
shows
$\forall n\ m.\ \Gamma\vdash \{\!|\acute{}n=n \wedge \acute{}m=m|\!\}\ \acute{}k :== CALL\ Max(LEQ\text{-}'proc,\acute{}n,\acute{}m)\ \{\!|\acute{}k = mx\ (\leq)\ n\ m|\!\}$
⟨*proof*⟩

end

23 Examples for Procedures as Parameters using Statespaces

theory *ProcParExSP* **imports** ../*Vcg* **begin**

lemma *DynProcProcPar'*:
assumes *adapt*: $P \subseteq \{s.\ p\ s = q \wedge$
$(\exists Z.\ init\ s \in P'\ Z \wedge$
$(\forall t \in Q'\ Z.\ return\ s\ t \in R\ s\ t) \wedge$
$(\forall t \in A'\ Z.\ return\ s\ t \in A))\}$

assumes *result*: $\forall s\ t.\ \Gamma,\Theta\vdash_{/F}\ (R\ s\ t)\ result\ s\ t\ Q,A$
assumes *q*: $\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ Call\ q\ (Q'\ Z),(A'\ Z)$
shows $\Gamma,\Theta\vdash_{/F}\ P\ dynCall\ init\ p\ return\ result\ Q,A$
⟨*proof*⟩

lemma *conseq-exploit-pre′*:
$[\![\forall s \in S.\ \Gamma,\Theta \vdash (\{s\} \cap P)\ c\ Q,A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash\ (P \cap S)c\ Q,A$
⟨*proof*⟩

lemma *conseq-exploit-pre″*:
$[\![\forall Z.\ \forall s \in S\ Z.\ \ \Gamma,\Theta \vdash (\{s\} \cap P\ Z)\ c\ (Q\ Z),(A\ Z)]\!]$
$\Longrightarrow$
$\forall Z.\ \Gamma,\Theta\vdash\ (P\ Z \cap S\ Z)c\ (Q\ Z),(A\ Z)$
⟨*proof*⟩

lemma *conseq-exploit-pre‴*:
$[\![\forall s \in S.\ \forall Z.\ \Gamma,\Theta \vdash (\{s\} \cap P\ Z)\ c\ (Q\ Z),(A\ Z)]\!]$
$\Longrightarrow$
$\forall Z.\ \Gamma,\Theta\vdash\ (P\ Z \cap S)c\ (Q\ Z),(A\ Z)$
⟨*proof*⟩

procedures *compare*(*i*::*nat*,*j*::*nat*|*r*::*bool*) *NoBody*

print-locale! *compare-signature*

context *compare-impl*
begin
declare [[*hoare-use-call-tr′* = *false*]]
term *´r* :== *CALL compare*(*´i*,*´j*)
declare [[*hoare-use-call-tr′* = *true*]]
end

procedures
LEQ (*i*::*nat*,*j*::*nat* | *r*::*bool*) *´r* :== *´i* ≤ *´j*
LEQ-spec: $\forall \sigma.\ \Gamma\vdash \{\sigma\}$ *PROC LEQ*(*´i*,*´j*,*´r*) $\{\!|\,´r = ({}^{\sigma}i \le {}^{\sigma}j)|\!\}$

LEQ-modifies: $\forall \sigma.\ \Gamma\vdash \{\sigma\}$ *PROC LEQ*(*´i*,*´j*,*´r*) $\{t.\ t$ *may-only-modify-globals* σ *in* $[]\}$

definition *mx*:: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a* ⇒ '*a* ⇒ '*a*
  where *mx leq a b* = (*if leq a b then a else b*)

procedures (**imports** *compare-signature*)
  Max (*compare*::*string*, *n*::*nat*, *m*::*nat* | *k*::*nat*)
  where *b*::*bool*
  in
  ´b :== *DYNCALL ´compare*(*´n*,*´m*);;
   IF ´b THEN ´k :== *´n ELSE ´k* :== *´m FI*

  Max-spec: ⋀*leq*. ∀ *σ*. Γ⊢
  ({*σ*} ∩ {*s*. (∀ *τ*. Γ⊢ {*τ*} *´r* :== *PROC* ^{s}*compare*(*´i*,*´j*) {|*´r* = (*leq* $^\tau$*i* $^\tau$*j*)|}) ∧
            (∀ *τ*. Γ⊢ {*τ*} *´r* :== *PROC* ^{s}*compare*(*´i*,*´j*) {*t*. *t may-only-modify-globals*
τ in []})})
    PROC Max(*´compare*,*´n*,*´m*,*´k*)
  {|*´k* = *mx leq* $^\sigma$*n* $^\sigma$*m*|}

context *Max-spec*
begin
thm *Max-spec*
end
context *Max-impl*
begin
term *´b* :== *DYNCALL ´compare*(*´n*,*´m*)
declare [[*hoare-use-call-tr'* = *false*]]
term *´b* :== *DYNCALL ´compare*(*´n*,*´m*)
declare [[*hoare-use-call-tr'* = *true*]]
end

lemma (**in** *Max-impl*) *Max-spec1*:
shows
∀ *σ leq*. Γ⊢
  ({*σ*} ∩ {| (∀ *τ*. Γ⊢{*τ*} *´r* :== *PROC ´compare*(*´i*,*´j*) {|*´r* = (*leq* $^\tau$*i* $^\tau$*j*)|}) ∧
     (∀ *τ*. Γ⊢ {*τ*} *´r* :== *PROC ´compare*(*´i*,*´j*) {*t*. *t may-only-modify-globals τ in*
[]})|})
    ´k :== *PROC Max*(*´compare*,*´n*,*´m*)
  {|*´k* = *mx leq* $^\sigma$*n* $^\sigma$*m*|}
⟨*proof*⟩

lemma (**in** *Max-impl*) *Max-spec2*:
shows
∀ *σ leq*. Γ⊢
  ({*σ*} ∩ {|(∀ *τ*. Γ⊢ {*τ*} *´r* :== *PROC ´compare*(*´i*,*´j*) {|*´r* = (*leq* $^\tau$*i* $^\tau$*j*)|}) ∧
     (∀ *τ*. Γ⊢ {*τ*} *´r* :== *PROC ´compare*(*´i*,*´j*) {*t*. *t may-only-modify-globals τ in*

[]})⦄)
   ´k :== PROC Max(´compare,´n,´m)
  ⦃´k = mx leq $^\sigma$n $^\sigma$m⦄
⟨proof⟩

lemma (**in** *Max-impl*) *Max-spec3*:
shows
∀ n m leq. Γ⊢
  (⦃´n=n ∧ ´m=m⦄ ∩
   ⦃(∀ τ. Γ⊢ {τ} ´r :== PROC ´compare(´i,´j) ⦃´r = (leq $^\tau$i $^\tau$j)⦄) ∧
     (∀ τ. Γ⊢ {τ} ´r :== PROC ´compare(´i,´j) {t. t may-only-modify-globals τ in
[]})⦄)
   ´k :== PROC Max(´compare,´n,´m)
  ⦃´k = mx leq n m⦄
⟨proof⟩

lemma (**in** *Max-impl*) *Max-spec4*:
shows
∀ n m leq. Γ⊢
  (⦃´n=n ∧ ´m=m⦄ ∩ ⦃∀ τ. Γ⊢ {τ} ´r :== PROC ´compare(´i,´j) ⦃´r = (leq $^\tau$i $^\tau$
j)⦄⦄)
   ´k :== PROC Max(´compare,´n,´m)
  ⦃´k = mx leq n m⦄
⟨proof⟩

print-locale *Max-spec*

locale *Max-test* = *Max-spec* **where**
    i-'compare-' = i-'LEQ-' **and**
    j-'compare-' = j-'LEQ-' **and**
    r-'compare-' = r-'LEQ-'
    + *LEQ-spec* + *LEQ-modifies*

lemma (**in** *Max-test*)
  shows
  Γ⊢ {σ} ´k :== CALL Max(LEQ-'proc,´n,´m) ⦃´k = mx (≤) $^\sigma$n $^\sigma$m⦄
⟨proof⟩

lemma (**in** *Max-impl*) *Max-spec5*:
shows
∀ n m leq. Γ⊢
  (⦃´n=n ∧ ´m=m⦄ ∩ ⦃∀ n′ m′. Γ⊢ ⦃´i=n′ ∧ ´j=m′⦄ ´r :== PROC ´compare(´i,´
j) ⦃´r = (leq n′ m′)⦄⦄)

´k :== *PROC Max*(´*compare*,´*n*,´*m*)
⦃´k = *mx leq n m*⦄
⟨*proof*⟩

lemma (**in** *LEQ-impl*)
LEQ-spec: ∀ *n m*. Γ⊢ ⦃´i=n ∧ ´j=m⦄ *PROC LEQ*(´i,´j,´r) ⦃´r = (n ≤ m)⦄
⟨*proof*⟩

print-locale *Max-impl*
locale *Max-test'* = *Max-impl* **where**
i-'compare-' = *i-'LEQ-'* **and**
j-'compare-' = *j-'LEQ-'* **and**
r-'compare-' = *r-'LEQ-'*
+ *LEQ-impl*
lemma (**in** *Max-test'*)
shows
∀ *n m*. Γ⊢ ⦃´n=n ∧ ´m=m⦄ ´k :== *CALL Max*(*LEQ-'proc*,´n,´m) ⦃´k = *mx* (≤) *n m*⦄
⟨*proof*⟩

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 (λ*s*. *call* (*upd* (*fst* (*cl s*)) ∘ *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*: $P \subseteq \{s.\ \exists P'\ Q'\ A'.\ \Gamma,\Theta\vdash_{/F}\ P'\ (callClosure\ upd\ (cl\ s))\ Q',A' \wedge$
$init\ s \in P' \wedge$
$(\forall t \in Q'.\ return\ s\ t \in R\ s\ t) \wedge$
$(\forall t \in A'.\ return\ s\ t \in A)\}$
assumes *res*: $\forall s\ t.\ \Gamma,\Theta\vdash_{/F}\ (R\ s\ t)\ (c\ s\ t)\ Q,A$
shows
$\Gamma,\Theta\vdash_{/F}\ P\ (dynCallClosure\ init\ upd\ cl\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *in-subsetD*: $[\![P \subseteq P';\ x \in P]\!] \Longrightarrow x \in P'$
⟨*proof*⟩

lemma *dynCallClosureFix*:
assumes *adapt*: $P \subseteq \{s.\ \exists Z.\ cl'=cl\ s \wedge$
$init\ s \in P'\ Z \wedge$
$(\forall t \in Q'\ Z.\ return\ s\ t \in R\ s\ t) \wedge$
$(\forall t \in A'\ Z.\ return\ s\ t \in A)\}$
assumes *res*: $\forall s\ t.\ \Gamma,\Theta\vdash_{/F}\ (R\ s\ t)\ (c\ s\ t)\ Q,A$
assumes *spec*: $\forall Z.\ \Gamma,\Theta\vdash_{/F}\ (P'\ Z)\ (callClosure\ upd\ cl')\ (Q'\ Z),(A'\ Z)$
shows
$\Gamma,\Theta\vdash_{/F}\ P\ (dynCallClosure\ init\ upd\ cl\ return\ c)\ Q,A$
⟨*proof*⟩

lemma *conseq-extract-pre*:
$[\![\forall s \in P.\ \Gamma,\Theta\vdash_{/F}\ (\{s\})\ c\ Q,A]\!]$
$\Longrightarrow$
$\Gamma,\Theta\vdash_{/F}\ P\ c\ Q,A$
⟨*proof*⟩

lemma *app-closure-sound*:
assumes *adapt*: $P \subseteq \{s.\ \exists P'\ Q'\ A'.\ \forall n.\ \Gamma,\Theta\models n:_{/F}\ P'\ (callClosure\ upd\ (e',p))$
$Q',A' \wedge$
$upd\ x\ s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A\}$
assumes *ap*: $upd\ e = upd\ e' \circ upd\ x$
shows $\Gamma,\Theta\models n:_{/F}\ P\ (callClosure\ upd\ (e,p))\ Q,A$
⟨*proof*⟩

lemma *app-closure*:
assumes *adapt*: $P \subseteq \{s.\ \exists P'\ Q'\ A'.\ \Gamma,\Theta\vdash_{/F}\ P'\ (callClosure\ upd\ (e',p))\ Q',A' \wedge$

$upd\ x\ s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A\}$
assumes *ap*: $upd\ e = upd\ e' \circ upd\ x$
shows $\Gamma,\Theta\vdash_{/F} P\ (callClosure\ upd\ (e,p))\ Q,A$
⟨*proof*⟩

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 $\Gamma,\Theta\vdash_{/F} P\ (callClosure\ upd\ (e,p))\ Q,A$
⟨*proof*⟩

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* ≡ (*es*@*fst c*,*snd c*)

lemma *gen-upd-app*: $\bigwedge es'$. *gen-upd var* (*es*@*es'*) = *gen-upd var es'* ∘ *gen-upd var es*
⟨*proof*⟩

lemma *gen-upd-ap*:
gen-upd var (*fst* (*ap es* (*es'*,*p*))) = *gen-upd var es'* ∘ *gen-upd var es*
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

end

theory *ClosureEx*
imports ../*Vcg* ../*Simpl-Heap Closure*
begin

record *globals* =
cnt-' :: *ref* ⇒ *nat*

alloc-' :: *ref list*
free-' :: *nat*
record *'g vars* = *'g state* +
p-':: *ref*
r-':: *nat*
n-':: *nat*
m-':: *nat*
c-':: (*string* × *ref*) *list* × *string*
d-':: (*string* × *ref*) *list* × *string*
e-':: (*string* × *nat*) *list* × *string*

definition var_n = [''n''↦ (λx. n-'-update (λ-. x)),
''m''↦ (λx. m-'-update (λ-. x))]
definition upd_n = *gen-upd* var_n

lemma upd_n-*ap*: upd_n (*fst* (*ap es* (*es'*,*p*))) = upd_n *es'* ∘ upd_n *es*
⟨*proof*⟩

lemma
Γ⊢{|´n=n_0 ∧ (∀ i j. Γ⊢ {|´n=i ∧ ´m=j|} callClosure upd_n ´e {|´r=i + j|})|}
´e :== (ap [(''n'',´n)] ´e)
{|∀ j. Γ⊢ {|´m=j|} callClosure upd_n ´e {|´r=n_0 + j|}|}
⟨*proof*⟩

definition *var* = [''p''↦ (λx. p-'-update (λ-. x))]
definition *upd* = *gen-upd var*

procedures *Inc*(*p*|*r*) =
´p→´cnt :== ´p→´cnt + *1*;;
´r :== ´p→´cnt

lemma (**in** *Inc-impl*)
∀ i p. Γ⊢ {|´p→´cnt = i|} ´r :== *PROC Inc*(´p) {|´r=i+*1* ∧ ´p→´cnt = i+*1*|}
⟨*proof*⟩

procedures (**imports** *Inc-signature*) *NewCounter*(|*c*) =
´p :== *NEW 1* [´cnt :== *0*];;
´c :== ([(''p'',´p)],*Inc-'proc*)

locale *NewCounter-impl'* = *NewCounter-impl* + *Inc-impl*
lemma (**in** *NewCounter-impl'*)
shows
∀ alloc. Γ⊢ {|*1* ≤ ´free|} ´c :== *PROC NewCounter*()
{|∃ p. p→´cnt = *0* ∧
(∀ i. Γ⊢ {|p→´cnt = i|} callClosure upd ´c {|´r=i+*1* ∧ p→´cnt = i+*1*|})|}

⟨*proof*⟩

lemma (**in** *NewCounter-impl'*)

shows
$\forall$ *alloc*. Γ⊢ {|*1* ≤ ´*free*|} ´*c* :== *PROC NewCounter*()
{|∃ *p*. *p*→´*cnt* = *0* ∧
(∀ *i*. Γ⊢ {|*p*→´*cnt* = *i*|} *callClosure upd* ´*c* {|´*r*=*i*+*1* ∧ *p*→´*cnt* = *i*+*1*|})|}
⟨*proof*⟩

lemma (**in** *NewCounter-impl'*)
shows *NewCounter-spec*:
∀ *alloc*. Γ⊢ {|*1* ≤ ´*free* ∧ ´*alloc*=*alloc*|} ´*c* :== *PROC NewCounter*()
{|∃ *p*. *p* ∉ *set alloc* ∧ *p* ∈ *set* ´*alloc* ∧ *p* ≠ *Null* ∧ *p*→´*cnt* = *0* ∧
(∀ *i*. Γ⊢ {|*p*→´*cnt* = *i*|} *callClosure upd* ´*c* {|´*r*=*i*+*1* ∧ *p*→´*cnt* = *i*+*1*|})|}
⟨*proof*⟩

lemma Γ⊢{|∃ *p*. *p* ≠ *Null* ∧ *p*→´*cnt* = *i* ∧
(∀ *i*. Γ⊢ {|*p*→´*cnt* = *i*|} *callClosure upd* ´*c* {|´*r*=*i*+*1* ∧ *p*→´*cnt* = *i*+*1*|})|}
dynCallClosure (λ*s*. *s*) *upd c-'* (λ*s t*. *s*(|*globals* := *globals t*|))
(λ*s t*. *Basic* (λ*u*. *u*(|*r-'* := *r-' t*|)))
{|´*r*=*i*+*1*|}
⟨*proof*⟩

declare [[*hoare-trace* = *1*]]

⟨*ML*⟩
lemma (**in** *NewCounter-impl'*)
shows Γ⊢ {|*1* ≤ ´*free*|}
´*c* :== *CALL NewCounter* ();;
dynCallClosure (λ*s*. *s*) *upd c-'* (λ*s t*. *s*(|*globals* := *globals t*|))
(λ*s t*. *Basic* (λ*u*. *u*(|*r-'* := *r-' t*|)))
{|´*r*=*1*|}
⟨*proof*⟩

lemma (**in** *NewCounter-impl'*)
shows Γ⊢ {|*1* ≤ ´*free*|}
´*c* :== *CALL NewCounter* ();;
dynCallClosure (λ*s*. *s*) *upd c-'* (λ*s t*. *s*(|*globals* := *globals t*|))
(λ*s t*. *Basic* (λ*u*. *u*(|*r-'* := *r-' t*|)));;
dynCallClosure (λ*s*. *s*) *upd c-'* (λ*s t*. *s*(|*globals* := *globals t*|))
(λ*s t*. *Basic* (λ*u*. *u*(|*r-'* := *r-' t*|)))
{|´*r*=*2*|}
⟨*proof*⟩

lemma (**in** *NewCounter-impl'*)
shows $\Gamma\vdash$ $\{\!|1 \leq$ ´*free*$|\!\}$
´*c* :== *CALL NewCounter* ();;
´*d* :== ´*c*;;
dynCallClosure ($\lambda s.\ s$) *upd c-'* ($\lambda s\ t.\ s(\!|globals := globals\ t|\!)$)
($\lambda s\ t.$ *Basic* ($\lambda u.\ u(\!|n\text{-}' := r\text{-}'\ t|\!)$));;
dynCallClosure ($\lambda s.\ s$) *upd d-'* ($\lambda s\ t.\ s(\!|globals := globals\ t|\!)$)
($\lambda s\ t.$ *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) =
(*Seq* ($lift_c$ *prj inject* c_1) ($lift_c$ *prj inject* c_2)) |
$lift_c$ *prj inject* (*Cond b* c_1 c_2) =
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*) |
$lift_c$ *prj inject* (*Call p*) = *Call p* |
$lift_c$ *prj inject* (*DynCom c*) = *DynCom* ($\lambda s.$ $lift_c$ *prj inject* (*c* (*prj s*))) |
$lift_c$ *prj inject* (*Guard f g c*) = *Guard f* ($lift_s$ *prj g*) ($lift_c$ *prj inject c*) |
$lift_c$ *prj inject Throw* = *Throw* |
$lift_c$ *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*)
⟨*proof*⟩

lemma $lift_c$-*Basic*:
($lift_c$ *prj inject c* = *Basic lf*) = ($\exists f.$ *c* = *Basic f* $\wedge$ *lf* = $lift_f$ *prj inject f*)
⟨*proof*⟩

lemma $lift_c$-*Spec*:
($lift_c$ *prj inject c* = *Spec lr*) = ($\exists r.$ *c* = *Spec r* $\wedge$ *lr* = $lift_r$ *prj inject r*)
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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'*)
⟨*proof*⟩

lemma $lift_c$-*Call*:
($lift_c$ *prj inject c* = *Call p*) = (*c* = *Call p*)
⟨*proof*⟩

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*))))
⟨*proof*⟩

lemma $lift_c$-*Guard*:

$(lift_c\ prj\ inject\ c = Guard\ f\ lg\ lc') =$
$(\exists g\ c'.\ c = Guard\ f\ g\ c' \wedge lg = lift_s\ prj\ g \wedge$
$lc' = lift_c\ prj\ inject\ c')$
⟨*proof*⟩

lemma *lift*$_c$*-Throw*:
$(lift_c\ prj\ inject\ c = Throw) = (c = Throw)$
⟨*proof*⟩

lemma *lift*$_c$*-Catch*:
$(lift_c\ prj\ inject\ c = Catch\ lc_1\ lc_2) =$
$(\exists\ c_1\ c_2.\ c = Catch\ c_1\ c_2 \wedge$
$lc_1 = lift_c\ prj\ inject\ c_1 \wedge lc_2 = lift_c\ prj\ inject\ c_2\ )$
⟨*proof*⟩

definition *xstate-map*:: $('S \Rightarrow 's) \Rightarrow ('S,'f)\ xstate \Rightarrow ('s,'f)\ xstate$
where
xstate-map g x = (*case x of*
Normal s ⇒ *Normal* (*g s*)
| *Abrupt s* ⇒ *Abrupt* (*g s*)
| *Fault f* ⇒ *Fault f*
| *Stuck* ⇒ *Stuck*)

lemma *xstate-map-simps* [*simp*]:
xstate-map g (*Normal s*) = *Normal* (*g s*)
xstate-map g (*Abrupt s*) = *Abrupt* (*g s*)
xstate-map g (*Fault f*) = (*Fault f*)
xstate-map g Stuck = *Stuck*
⟨*proof*⟩

lemma *xstate-map-Normal-conv*:
xstate-map g S = *Normal s* = $(\exists s'.\ S{=}Normal\ s' \wedge s = g\ s')$
⟨*proof*⟩

lemma *xstate-map-Abrupt-conv*:
xstate-map g S = *Abrupt s* = $(\exists s'.\ S{=}Abrupt\ s' \wedge s = g\ s')$
⟨*proof*⟩

lemma *xstate-map-Fault-conv*:
xstate-map g S = *Fault f* = (*S*=*Fault f*)
⟨*proof*⟩

lemma *xstate-map-Stuck-conv*:
xstate-map g S = *Stuck* = (*S*=*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 $project_x$::$('S,'f)$ *xstate* $\Rightarrow$ $('s,'f)$ *xstate*
fixes $lift_e$::$('s,'p,'f)$ *body* $\Rightarrow$ $('S,'p,'f)$ *body*
fixes $lift_c$:: $('s,'p,'f)$ *com* $\Rightarrow$ $('S,'p,'f)$ *com*
fixes $lift_f$:: $('s \Rightarrow 's) \Rightarrow ('S \Rightarrow 'S)$
fixes $lift_s$:: $'s$ *set* $\Rightarrow$ $'S$ *set*
fixes $lift_r$:: $('s \times 's)$ *set* $\Rightarrow$ $('S \times 'S)$ *set*
assumes *proj-inj-commute*: $\bigwedge S\ s.$ *project* (*inject* S s) = s
defines $lift_c \equiv$ *Compose*.$lift_c$ *project inject*
defines $project_x \equiv$ *xstate-map project*
defines $lift_e \equiv (\lambda\Gamma\ p.$ *map-option* $lift_c$ $(\Gamma\ p))$
defines $lift_f \equiv$ *Compose*.$lift_f$ *project inject*
defines $lift_s \equiv$ *Compose*.$lift_s$ *project*
defines $lift_r \equiv$ *Compose*.$lift_r$ *project inject*

lemma (**in** *lift-state-space*) $lift_f$*-simp*:
$lift_f$ $f \equiv \lambda S.$ *inject* S (f (*project* S))
⟨*proof*⟩

lemma (**in** *lift-state-space*) $lift_s$*-simp*:
$lift_s$ $A \equiv \{S.$ *project* $S \in A\}$
⟨*proof*⟩

lemma (**in** *lift-state-space*) $lift_r$*-simp*:
$lift_r$ $R \equiv \{(S,T).$ (*project* S,*project* T) $\in R \wedge T$=*inject* S (*project* T)$\}$
⟨*proof*⟩

lemma (**in** *lift-state-space*) $lift_c$*-Skip-simp* [*simp*]:
$lift_c$ *Skip* = *Skip*

$\langle proof \rangle$

lemma (**in** *lift-state-space*) $lift_c$*-Basic-simp* [*simp*]:
$lift_c\ (Basic\ f) = Basic\ (lift_f\ f)$
$\langle proof \rangle$

lemma (**in** *lift-state-space*) $lift_c$*-Spec-simp* [*simp*]:
$lift_c\ (Spec\ r) = Spec\ (lift_r\ r)$
$\langle proof \rangle$

lemma (**in** *lift-state-space*) $lift_c$*-Seq-simp* [*simp*]:
$lift_c\ (Seq\ c_1\ c_2) =$
$(Seq\ (lift_c\ c_1)\ (lift_c\ c_2))$
$\langle proof \rangle$

lemma (**in** *lift-state-space*) $lift_c$*-Cond-simp* [*simp*]:
$lift_c\ (Cond\ b\ c_1\ c_2) =$
$Cond\ (lift_s\ b)\ (lift_c\ c_1)\ (lift_c\ c_2)$
$\langle proof \rangle$

lemma (**in** *lift-state-space*) $lift_c$*-While-simp* [*simp*]:
$lift_c\ (While\ b\ c) =$
$While\ (lift_s\ b)\ (lift_c\ c)$
$\langle proof \rangle$

lemma (**in** *lift-state-space*) $lift_c$*-Call-simp* [*simp*]:
$lift_c\ (Call\ p) = Call\ p$
$\langle proof \rangle$

lemma (**in** *lift-state-space*) $lift_c$*-DynCom-simp* [*simp*]:
$lift_c\ (DynCom\ c) = DynCom\ (\lambda s.\ lift_c\ (c\ (project\ s)))$
$\langle proof \rangle$

lemma (**in** *lift-state-space*) $lift_c$*-Guard-simp* [*simp*]:
$lift_c\ (Guard\ f\ g\ c) = Guard\ f\ (lift_s\ g)\ (lift_c\ c)$
$\langle proof \rangle$

lemma (**in** *lift-state-space*) $lift_c$*-Throw-simp* [*simp*]:
$lift_c\ Throw = Throw$
$\langle proof \rangle$

lemma (**in** *lift-state-space*) $lift_c$*-Catch-simp* [*simp*]:
$lift_c\ (Catch\ c_1\ c_2) =$
$Catch\ (lift_c\ c_1)\ (lift_c\ c_2)$
$\langle proof \rangle$

lemma (**in** *lift-state-space*) $project_x$*-def'*:
$$
\begin{aligned}
project_x\ s \equiv (&case\ s\ of \\
&\ Normal\ s \Rightarrow Normal\ (project\ s) \\
&|\ Abrupt\ s \Rightarrow Abrupt\ (project\ s) \\
&|\ Fault\ f \Rightarrow Fault\ f \\
&|\ Stuck \Rightarrow Stuck)
\end{aligned}
$$
$\langle proof \rangle$

lemma (**in** *lift-state-space*) $lift_e$*-def'*:
$lift_e\ \Gamma\ p \equiv (case\ \Gamma\ p\ of\ Some\ bdy \Rightarrow Some\ (lift_c\ bdy)\ |\ None \Rightarrow None)$
$\langle proof \rangle$

The problem is that $lift_c\ project\ inject \circ \Gamma$ is quite a strong premise. The problem is that Γ is a function here. A map would be better. We only have to lift those procedures in the domain of Γ: Γ p = *Some bdy* ⟶ Γ′ p = *Some* lift_c *project inject bdy*. We then can com up with theorems that allow us to extend the domains of Γ and preserve validity.

lemma (**in** *lift-state-space*)
{(S,T). ∃ t. (*project* S,t) ∈ r ∧ T=*inject* S t}
⊆ {(S,T). (*project* S,*project* T) ∈ r ∧ T=*inject* S (*project* T)}
⟨*proof*⟩

lemma (**in** *lift-state-space*)
{(S,T). (*project* S,*project* T) ∈ r ∧ T=*inject* S (*project* T)}
⊆ {(S,T). ∃ t. (*project* S,t) ∈ r ∧ T=*inject* S t}
⟨*proof*⟩

lemma (**in** *lift-state-space*) *lift-exec*:
assumes *exec-lc*: (lift_e Γ)⊢⟨lc,s⟩ ⇒ t
shows ⋀c. ⟦ lift_c c = lc⟧ ⟹
Γ⊢⟨c,$\mathit{project}_x$ s⟩ ⇒ $\mathit{project}_x$ t
⟨*proof*⟩

lemma (**in** *lift-state-space*) *lift-exec′*:
assumes *exec-lc*: (lift_e Γ)⊢⟨lift_c c,s⟩ ⇒ t
shows Γ⊢⟨c,$\mathit{project}_x$ s⟩ ⇒ $\mathit{project}_x$ t
⟨*proof*⟩

lemma (**in** *lift-state-space*) *lift-valid*:
assumes *valid*: Γ$\models_{/F}$ P c Q,A
shows
(lift_e Γ)$\models_{/F}$ (lift_s P) (lift_c c) (lift_s Q),(lift_s A)
⟨*proof*⟩

lemma (**in** *lift-state-space*) *lift-hoarep*:
assumes *deriv*: Γ,{}$\vdash_{/F}$ P c Q,A
shows
(lift_e Γ),{}$\vdash_{/F}$ (lift_s P) (lift_c c) (lift_s Q),(lift_s A)
⟨*proof*⟩

lemma (**in** *lift-state-space*) *lift-hoarep′*:
∀ Z. Γ,{}$\vdash_{/F}$ (P Z) c (Q Z),(A Z) ⟹
∀ Z. (lift_e Γ),{}$\vdash_{/F}$ (lift_s (P Z)) (lift_c c)
(lift_s (Q Z)),(lift_s (A Z))
⟨*proof*⟩

lemma (**in** *lift-state-space*) *lift-termination*:
assumes *termi*: $\Gamma\vdash c\downarrow s$
shows $\bigwedge S.\ project_x\ S = s \Longrightarrow$
  $lift_e\ \Gamma \vdash (lift_c\ c)\downarrow S$
  ⟨*proof*⟩

lemma (**in** *lift-state-space*) *lift-termination'*:
assumes *termi*: $\Gamma\vdash c\downarrow project_x\ S$
shows $lift_e\ \Gamma \vdash (lift_c\ c)\downarrow S$
  ⟨*proof*⟩

lemma (**in** *lift-state-space*) *lift-validt*:
  assumes *valid*: $\Gamma\models_{t/F}\ P\ c\ Q,A$
  shows $(lift_e\ \Gamma)\models_{t/F}\ (lift_s\ P)\ (lift_c\ c)\ (lift_s\ Q),(lift_s\ A)$
⟨*proof*⟩

lemma (**in** *lift-state-space*) *lift-hoaret*:
  assumes *deriv*: $\Gamma,\{\}\vdash_{t/F}\ P\ c\ Q,A$
  shows
   $(lift_e\ \Gamma),\{\}\vdash_{t/F}\ (lift_s\ P)\ (lift_c\ c)\ (lift_s\ Q),(lift_s\ A)$
⟨*proof*⟩

locale *lift-state-space-ext* = *lift-state-space* +
  assumes *inj-proj-commute*: $\bigwedge S.\ inject\ S\ (project\ S) = S$
  assumes *inject-last*: $\bigwedge S\ s\ t.\ inject\ (inject\ S\ s)\ t = inject\ S\ t$

lemma (**in** *lift-state-space-ext*) *lift-exec-inject-same*:
assumes *exec-lc*: $(lift_e\ \Gamma)\vdash\langle lc,s\rangle \Rightarrow t$
shows $\bigwedge c.\ [\![lift_c\ c = lc;\ t \notin (Fault\ `\ UNIV) \cup \{Stuck\}]\!] \Longrightarrow$
      $state\ t = inject\ (state\ s)\ (project\ (state\ t))$
⟨*proof*⟩

lemma (**in** *lift-state-space-ext*) *valid-inject-project*:
 assumes *noFaultStuck*:
  $\Gamma\vdash\langle c,Normal\ (project\ \sigma)\rangle \Rightarrow\notin(Fault\ `\ UNIV \cup \{Stuck\})$
 shows $lift_e\ \Gamma\models_{/F}\ \{\sigma\}\ lift_c\ c$
      $\{t.\ t=inject\ \sigma\ (project\ t)\},\ \{t.\ t=inject\ \sigma\ (project\ t)\}$
⟨*proof*⟩

lemma (**in** *lift-state-space-ext*) *lift-exec-inject-same'*:
assumes *exec-lc*: $(lift_e\ \Gamma)\vdash\langle lift_c\ c,S\rangle \Rightarrow T$
shows $\bigwedge c.\ [\![T \notin (Fault\ `\ UNIV) \cup \{Stuck\}]\!] \Longrightarrow$
      $state\ T = inject\ (state\ S)\ (project\ (state\ T))$
  ⟨*proof*⟩

lemma (**in** *lift-state-space-ext*) *valid-lift-modifies*:
  assumes *valid*: $\forall s.\ \Gamma\models_{/F}\ \{s\}\ c\ (Modif\ s),(ModifAbr\ s)$
  shows $(lift_e\ \Gamma)\models_{/F}\ \{S\}\ (lift_c\ c)$
  $\{T.\ T \in lift_s\ (Modif\ (project\ S)) \wedge T{=}inject\ S\ (project\ T)\}$,
  $\{T.\ T \in lift_s\ (ModifAbr\ (project\ S)) \wedge T{=}inject\ S\ (project\ T)\}$
⟨*proof*⟩

lemma (**in** *lift-state-space-ext*) *hoare-lift-modifies*:
  assumes *deriv*: $\forall \sigma.\ \Gamma,\{\}\vdash_{/F}\ \{\sigma\}\ c\ (Modif\ \sigma),(ModifAbr\ \sigma)$
  shows $\forall \sigma.\ (lift_e\ \Gamma),\{\}\vdash_{/F}\ \{\sigma\}\ (lift_c\ c)$
  $\{T.\ T \in lift_s\ (Modif\ (project\ \sigma)) \wedge T{=}inject\ \sigma\ (project\ T)\}$,
  $\{T.\ T \in lift_s\ (ModifAbr\ (project\ \sigma)) \wedge T{=}inject\ \sigma\ (project\ T)\}$
⟨*proof*⟩

lemma (**in** *lift-state-space-ext*) *hoare-lift-modifies'*:
  assumes *deriv*: $\forall \sigma.\ \Gamma,\{\}\vdash_{/F}\ \{\sigma\}\ c\ (Modif\ \sigma),(ModifAbr\ \sigma)$
  shows $\forall \sigma.\ (lift_e\ \Gamma),\{\}\vdash_{/F}\ \{\sigma\}\ (lift_c\ c)$
  $\{T.\ T \in lift_s\ (Modif\ (project\ \sigma)) \wedge$
  $(\exists T'.\ T{=}inject\ \sigma\ T')\}$,
  $\{T.\ T \in lift_s\ (ModifAbr\ (project\ \sigma)) \wedge$
  $(\exists T'.\ T{=}inject\ \sigma\ T')\}$
⟨*proof*⟩

25.2 Renaming Procedures

primrec *rename*:: $('p \Rightarrow 'q) \Rightarrow ('s,'p,'f)\ com \Rightarrow ('s,'q,'f)\ com$
where
rename N Skip = Skip |
rename N (*Basic f*) = *Basic f* |
rename N (*Spec r*) = *Spec r* |
rename N (*Seq* c_1 c_2) = (*Seq* (*rename N* c_1) (*rename N* c_2)) |
rename N (*Cond b* c_1 c_2) = *Cond b* (*rename N* c_1) (*rename N* c_2) |
rename N (*While b c*) = *While b* (*rename N c*) |
rename N (*Call p*) = *Call* (*N p*) |
rename N (*DynCom c*) = *DynCom* (λ*s. rename N* (*c s*)) |
rename N (*Guard f g c*) = *Guard f g* (*rename N c*) |
rename N Throw = Throw |
rename N (*Catch* c_1 c_2) = *Catch* (*rename N* c_1) (*rename N* c_2)

lemma *rename-Skip*: *rename h c = Skip* = (*c=Skip*)
  ⟨*proof*⟩

lemma *rename-Basic*:
  (*rename h c = Basic f*) = (*c=Basic f*)
  ⟨*proof*⟩

lemma *rename-Spec*:

$(rename\ h\ c = Spec\ r) = (c{=}Spec\ r)$
⟨proof⟩

lemma *rename-Seq*:
$(rename\ h\ c = Seq\ rc_1\ rc_2) =$
$(\exists\ c_1\ c_2.\ c = Seq\ c_1\ c_2 \wedge$
$rc_1 = rename\ h\ c_1 \wedge rc_2 = rename\ h\ c_2\ )$
⟨proof⟩

lemma *rename-Cond*:
$(rename\ h\ c = Cond\ b\ rc_1\ rc_2) =$
$(\exists\ c_1\ c_2.\ c = Cond\ b\ c_1\ c_2\ \wedge rc_1 = rename\ h\ c_1 \wedge rc_2 = rename\ h\ c_2\ )$
⟨proof⟩

lemma *rename-While*:
$(rename\ h\ c = While\ b\ rc') = (\exists\ c'.\ c = While\ b\ c' \wedge rc' = rename\ h\ c')$
⟨proof⟩

lemma *rename-Call*:
$(rename\ h\ c = Call\ q) = (\exists\ p.\ c = Call\ p \wedge q{=}h\ p)$
⟨proof⟩

lemma *rename-DynCom*:
$(rename\ h\ c = DynCom\ rc) = (\exists\ C.\ c{=}DynCom\ C \wedge rc = (\lambda s.\ rename\ h\ (C\ s)))$
⟨proof⟩

lemma *rename-Guard*:
$(rename\ h\ c = Guard\ f\ g\ rc') =$
$(\exists\ c'.\ c = Guard\ f\ g\ c' \wedge rc' = rename\ h\ c')$
⟨proof⟩

lemma *rename-Throw*:
$(rename\ h\ c = Throw) = (c = Throw)$
⟨proof⟩

lemma *rename-Catch*:
$(rename\ h\ c = Catch\ rc_1\ rc_2) =$
$(\exists\ c_1\ c_2.\ c = Catch\ c_1\ c_2 \wedge rc_1 = rename\ h\ c_1 \wedge rc_2 = rename\ h\ c_2\ )$
⟨proof⟩

lemma *exec-rename-to-exec*:
assumes Γ: $\forall\ p\ bdy.\ \Gamma\ p = Some\ bdy \longrightarrow \Gamma'\ (h\ p) = Some\ (rename\ h\ bdy)$
assumes *exec*: $\Gamma' \vdash \langle rc,s\rangle \Rightarrow t$
shows $\bigwedge c.\ rename\ h\ c = rc \Longrightarrow\ \exists\ t'.\ \Gamma \vdash \langle c,s\rangle \Rightarrow t' \wedge (t'{=}Stuck \vee t'{=}t)$
⟨proof⟩

lemma *exec-rename-to-exec'*:

assumes Γ: ∀ p *bdy*. Γ p = *Some bdy* ⟶ Γ′ (N p) = *Some* (*rename N bdy*)
assumes *exec*: Γ′⊢⟨*rename N c*,*s*⟩ ⇒ t
shows ∃ t'. Γ⊢⟨c,s⟩ ⇒ t' ∧ (t'=*Stuck* ∨ t'=t)
⟨*proof*⟩

lemma *valid-to-valid-rename*:
assumes Γ: ∀ p *bdy*. Γ p = *Some bdy* ⟶ Γ′ (N p) = *Some* (*rename N bdy*)
assumes *valid*: Γ$\models_{/F}$ P c Q,A
shows Γ′$\models_{/F}$ P (*rename N c*) Q,A
⟨*proof*⟩

lemma *hoare-to-hoare-rename*:
assumes Γ: ∀ p *bdy*. Γ p = *Some bdy* ⟶ Γ′ (N p) = *Some* (*rename N bdy*)
assumes *deriv*: Γ,{}$\vdash_{/F}$ P c Q,A
shows Γ′,{}$\vdash_{/F}$ P (*rename N c*) Q,A
⟨*proof*⟩

lemma *hoare-to-hoare-rename′*:
assumes Γ: ∀ p *bdy*. Γ p = *Some bdy* ⟶ Γ′ (N p) = *Some* (*rename N bdy*)
assumes *deriv*: ∀ Z. Γ,{}$\vdash_{/F}$ (P Z) c (Q Z),(A Z)
shows ∀ Z. Γ′,{}$\vdash_{/F}$ (P Z) (*rename N c*) (Q Z),(A Z)
⟨*proof*⟩

lemma *terminates-to-terminates-rename*:
assumes Γ: ∀ p *bdy*. Γ p = *Some bdy* ⟶ Γ′ (N p) = *Some* (*rename N bdy*)
assumes *termi*: Γ⊢ c ↓ s
assumes *noStuck*: Γ⊢ ⟨c,s⟩ ⇒∉{*Stuck*}
shows Γ′⊢ *rename N c* ↓ s
⟨*proof*⟩

lemma *validt-to-validt-rename*:
assumes Γ: ∀ p *bdy*. Γ p = *Some bdy* ⟶ Γ′ (N p) = *Some* (*rename N bdy*)
assumes *valid*: Γ$\models_{t/F}$ P c Q,A
shows Γ′$\models_{t/F}$ P (*rename N c*) Q,A
⟨*proof*⟩

lemma *hoaret-to-hoaret-rename*:
assumes Γ: ∀ p *bdy*. Γ p = *Some bdy* ⟶ Γ′ (N p) = *Some* (*rename N bdy*)
assumes *deriv*: Γ,{}$\vdash_{t/F}$ P c Q,A
shows Γ′,{}$\vdash_{t/F}$ P (*rename N c*) Q,A
⟨*proof*⟩

lemma *hoaret-to-hoaret-rename′*:
assumes Γ: ∀ p *bdy*. Γ p = *Some bdy* ⟶ Γ′ (N p) = *Some* (*rename N bdy*)
assumes *deriv*: ∀ Z. Γ,{}$\vdash_{t/F}$ (P Z) c (Q Z),(A Z)

shows $\forall Z.\ \Gamma',\{\}\vdash_{t/F}$ $(P\ Z)$ $(rename\ N\ c)$ $(Q\ Z),(A\ Z)$
⟨*proof*⟩

lemma $lift_c$*-whileAnno* [*simp*]: $lift_c$ *prj inject* (*whileAnno b I V c*) =
whileAnno ($lift_s$ *prj b*)
($lift_s$ *prj I*) ($lift_r$ *prj inject V*) ($lift_c$ *prj inject c*)
⟨*proof*⟩

lemma $lift_c$*-block* [*simp*]: $lift_c$ *prj inject* (*block init bdy return c*) =
block ($lift_f$ *prj inject init*) ($lift_c$ *prj inject bdy*)
(λ*s*. ($lift_f$ *prj inject* (*return* (*prj s*))))
(λ*s t*. $lift_c$ *prj inject* (*c* (*prj s*) (*prj t*)))
⟨*proof*⟩

lemma $lift_c$*-call* [*simp*]: $lift_c$ *prj inject* (*call init p return c*) =
call ($lift_f$ *prj inject init*) *p*
(λ*s*. ($lift_f$ *prj inject* (*return* (*prj s*))))
(λ*s t*. $lift_c$ *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* (λ*s t*. *rename h* (*c s t*))
⟨*proof*⟩

lemma *rename-call* [*simp*]: *rename h* (*call init p return c*) =
call init (*h p*) *return* (λ*s 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:
$\forall \sigma.\ \Gamma\vdash_{/UNIV} \{\sigma\}$ ´sl-q :== PROC Rev(´p) {t. t may-only-modify-globals σ in [next]}
⟨proof⟩

lemma (**in** *Rev-impl*) **shows**
Rev-spec:
$\forall Ps.\ \Gamma\vdash$ {|List ´p ´next Ps|} ´sl-q :== PROC 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(|*strnext-*′ := *next-*′ *g*|)

definition *inject-str*::*state* ⇒ *state-list* ⇒ *state* **where**
inject-str S s = *S*(|*globals* := *inject-globals-str* (*globals S*) (*globals s*),
str-′ := *state-list.p-*′ *s*, *q-*′ := *sl-q-*′ *s*,
r-′ := *state-list.r-*′ *s*|)

lemma *globals-inject-project-str-commutes*:
inject-globals-str G (*project-globals-str G*) = *G*
⟨*proof*⟩

lemma *inject-project-str-commutes*: *inject-str S* (*project-str S*) = *S*
⟨*proof*⟩

lemma *globals-project-inject-str-commutes*:
project-globals-str (*inject-globals-str G g*) = *g*
⟨*proof*⟩

lemma *project-inject-str-commutes*: *project-str* (*inject-str S s*) = *s*
⟨*proof*⟩

lemma *globals-inject-str-last*:
inject-globals-str (*inject-globals-str G g*) *g*′ = *inject-globals-str G g*′
⟨*proof*⟩

lemma *inject-str-last*:
inject-str (*inject-str S s*) *s*′ = *inject-str S s*′
⟨*proof*⟩

definition
$lift_e$ = (λΓ *p*. *map-option* ($lift_c$ *project-str inject-str*) (Γ *p*))
print-locale *lift-state-space*
interpretation *ex*: *lift-state-space project-str inject-str*
xstate-map project-str $lift_e$ $lift_c$ *project-str inject-str*
$lift_f$ *project-str inject-str* $lift_s$ *project-str*
$lift_r$ *project-str inject-str*
⟨*proof*⟩

interpretation *ex*: *lift-state-space-ext project-str inject-str*
xstate-map project-str $lift_e$ $lift_c$ *project-str inject-str*
$lift_f$ *project-str inject-str* $lift_s$ *project-str*
$lift_r$ *project-str inject-str*

⟨*proof*⟩

lemmas *Rev-lift-spec* = *ex.lift-hoarep'* [*OF Rev-impl.Rev-spec*,*simplified* $lift_s$*-def project-str-def project-globals-str-def*,*simplified*, *of* - ''*Rev*'']
print-theorems

definition $\mathcal{N}$ *p' p* = (*if p*=''*Rev*'' *then p' else* '''')

procedures *RevStr*(*str*|*q*) = *rename* ($\mathcal{N}$ *RevStr-'proc*)
(*lift*$_c$ *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''*:
∀ *Ps. lift*$_e$ [''*Rev*'' ↦ *Rev-body.Rev-body*]
⊢ {|*List ´str ´strnext Ps*|} *Call* ''*Rev*'' {|*List ´q ´strnext* (*rev Ps*)|}
⟨*proof*⟩

lemma (**in** *RevStr-impl*) $\mathcal{N}$*-ok*:
∀ *p bdy*. (*lift*$_e$ [''*Rev*'' ↦ *Rev-body.Rev-body*]) *p* = *Some bdy* ⟶
Γ ($\mathcal{N}$ *RevStr-'proc p*) = *Some* (*rename* ($\mathcal{N}$ *RevStr-'proc*) *bdy*)
⟨*proof*⟩

context *RevStr-impl*
begin
thm *hoare-to-hoare-rename'*[*OF* - *Rev-lift-spec''*, *OF* $\mathcal{N}$*-ok*,
simplified $\mathcal{N}$*-def*, *simplified*]
end

lemmas (**in** *RevStr-impl*) *RevStr-spec* =
hoare-to-hoare-rename' [*OF* - *Rev-lift-spec''*, *OF* $\mathcal{N}$*-ok*,
simplified $\mathcal{N}$*-def*, *simplified*]

lemma (**in** *RevStr-impl*) *RevStr-spec'*:
∀ *Ps*. Γ⊢ {|*List ´str ´strnext Ps*|} *´q* :== *PROC RevStr*(*´str*)
{|*List ´q ´strnext* (*rev Ps*)|}
⟨*proof*⟩

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 $\mathcal{N}$*-ok*, *of ″Rev″*, *simplified* $\mathcal{N}$*-def Rev-clique-def*,*simplified*]
end

lemma (**in** *RevStr-impl*) *RevStr-modifies*:
$\forall\sigma.\ \Gamma\vdash_{/UNIV} \{\sigma\}$ ´*str* :== *PROC RevStr*(´*str*)
{*t*. *t may-only-modify-globals* σ *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 'satespace' 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 $A\text{-}'$ 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 $A\text{-}'$ in a state s is written as $s{\cdot}A\text{-}'$, and update with a value *term v* as $s\langle A\text{-}' := v\rangle$.

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*) $\Gamma\vdash \{\!|\,\acute{}N = 5|\!\}\ \acute{}N :== 2 * \acute{}N\ \{\!|\,\acute{}N = 10|\!\}$
  ⟨*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 $\{\!|\,\acute{}N = 5|\!\}$ internally gets expanded to $\{s.\ locals\ s\ {\cdot}N\text{-}' = 5\}$ written in ordinary set comprehension notation of Isabelle. It describes the set of states where the $N\text{-}'$ 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 $\Gamma,\{\}\vdash \{\!|\,\acute{}N = 5|\!\}\ \acute{}N :== 2 * \acute{}N\ \{\!|\,\acute{}N = 10|\!\},\{\}$.

We can step through verification condition generation by the method *vcg-step*.

lemma (**in** *vars*) $\Gamma,\{\}\vdash \{\!|\,\acute{}N = 5|\!\}\ \acute{}N :== 2 * \acute{}N\ \{\!|\,\acute{}N = 10|\!\}$
  ⟨*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*) $\Gamma\vdash \{\!|\,\acute{}N = 5|\!\}\ \acute{}N :== 2 * \acute{}N\ \{\!|\,\acute{}N = 10|\!\}$
  ⟨*proof*⟩

The next example shows how we deal with the while loop. Note the invariant annotation.

lemma (**in** *vars*)
$\Gamma,\{\}\vdash \{\!|\,\acute{}M = 0 \wedge \acute{}S = 0|\!\}$
$\quad WHILE\ \acute{}M \neq a$
$\quad INV\ \{\!|\,\acute{}S = \acute{}M * b|\!\}$
$\quad DO\ \acute{}S :== \acute{}S + b;;\ \acute{}M :== \acute{}M + 1\ OD$
$\quad \{\!|\,\acute{}S = a * b|\!\}$
$\langle proof \rangle$

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**
$\acute{}R :== \acute{}N * \acute{}N$

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 Γ *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 Γ 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:

$call\ (\lambda s.\ s(\!|locals := locals\ s\langle N\text{-}'Square\text{-}' := locals\ s{\cdot}I\text{-}'Square\text{-}'\rangle|\!))$
$Square\text{-}'proc\ (\lambda s\ t.\ s(\!|globals := globals\ t|\!))$
$(\lambda i\ t.\ {}^{\acute{}}R :== locals\ t{\cdot}R\text{-}'Square\text{-}')$

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*:

$$[\![\forall Z.\ \Gamma,\Theta\vdash_{/F} (P\ Z)\ the\ (\Gamma\ p)\ (Q\ Z),(A\ Z);\ p \in dom\ \Gamma]\!] \Longrightarrow \forall Z.\ \Gamma,\Theta\vdash_{/F} (P\ Z)\ Call\ p\ (Q\ Z),(A\ Z)$$

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 $\forall n.\ \Gamma\vdash \{\!|{}^{\acute{}}N = n|\!\}\ \ {}^{\acute{}}R :== PROC\ Square({}^{\acute{}}N)\ \{\!|{}^{\acute{}}R = n * n|\!\}$

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):

$$[\![\forall Z.\ \Gamma,\Theta \cup (\bigcup_Z \{(P\ Z,\ p,\ Q\ Z,\ A\ Z)\})\vdash_{/F} (P\ Z)\ the\ (\Gamma\ p)\ (Q\ Z),(A\ Z);\ p \in dom\ \Gamma]\!] \Longrightarrow \forall Z.\ \Gamma,\Theta\vdash_{/F} (P\ Z)\ Call\ p\ (Q\ Z),(A\ Z)$$

The verification condition generator infers the specification out of the context Θ 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*():

$$
\begin{array}{l}
\mathit{call} \\
\ (\lambda s.\ s(\!|\mathit{locals} := \mathit{locals}\ s\langle p\text{-}'\mathit{append}\text{-}' := \mathit{locals}\ s{\cdot}p\text{-}'\mathit{append}\text{-}', \\
\qquad\qquad q\text{-}'\mathit{append}\text{-}' := \mathit{locals}\ s{\cdot}q\text{-}'\mathit{append}\text{-}'\rangle|\!)) \\
\ \mathit{append}\text{-}'\mathit{proc}\ (\lambda s\ t.\ s(\!|\mathit{globals} := \mathit{globals}\ t|\!)) \\
\ (\lambda i\ t.\ ´p :== \mathit{locals}\ t{\cdot}p\text{-}'\mathit{append}\text{-}')
\end{array}
$$

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 σ, 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 $\{\!|\sigma.\ \ldots|\!\}$ is a shorthand to fix the current state: $\{s.\ \sigma = s\ \ldots\}$. Moreover $^{\sigma}x$ abbreviates the lookup of variable x in the state σ.
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 then on the heap. The basic abstractions are:

$$
\begin{array}{l}
\mathit{Path}\ x\ h\ y\ [\,] = (x = y) \\
\mathit{Path}\ x\ h\ y\ (p \cdot ps) = (x = p \wedge x \neq \mathit{Null} \wedge \mathit{Path}\ (h\ x)\ h\ y\ ps)
\end{array}
$$

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*.
  $\Gamma\vdash$ ⦃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* = $^{\sigma}$*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 ∀ σ. $\Gamma\vdash_{/UNIV}$ {σ} *´p* :== *PROC append*(*´p*,*´q*)
    {*t*. *t may-only-modify-globals* σ *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 verfication 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*]}

$=$
$\{t.\ \exists\, next.\ globals\ t{=}update\ id\ id\ next\text{-}'\ (K\text{-}statefun\ next)\ (globals\ Z)\}$
⟨*proof*⟩

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*⦄
⟨*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 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:

$((x,\ y) \in measure\ f) = (f\ x < f\ y)$
$(((a,\ b),\ x,\ y) \in r <\!*lex*\!> s) = ((a,\ x) \in r \lor a = x \land (b,\ y) \in s)$
$((x,\ y) \in inv\text{-}image\ r\ f) = ((f\ x,\ f\ y) \in r)$

Another useful construction is <∗*mlex*∗> which is a combination of a measure and a lexicographic product:

$((x,\ y) \in f <\!*mlex*\!> r) = (f\ x < f\ y \lor f\ x = f\ y \land (x,\ y) \in r)$

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*)
$\Gamma\vdash_t$ $\{\!|\,\acute{}M = 0 \wedge \acute{}S = 0|\!\}$
  WHILE ´M $\neq$ *a*
  INV $\{\!|\,\acute{}S = \acute{}M * b \wedge \acute{}M \leq a|\!\}$
  VAR MEASURE a − ´M
  DO ´S :== ´S + b;; ´M :== ´M + 1 OD
  $\{\!|\,\acute{}S = a * b|\!\}$
⟨*proof*⟩

The variant annotation is preceded by *VAR*. The capital *MEASURE* is a shorthand for *measure* ($\lambda s.\ a - {}^{s}M$). Analogous there is a capital <∗*MLEX*∗>.

lemma (**in** *Fac-impl*) *Fac-spec′*:
shows $\forall\sigma.\ \Gamma\vdash_t \{\sigma\}$ *´R :== PROC Fac(´N)* $\{\!|\,\acute{}R = fac\ {}^{\sigma}N|\!\}$
⟨*proof*⟩

lemma (**in** *append-impl*) *append-spec2*:
shows $\forall\sigma\ Ps\ Qs.\ \Gamma\vdash_t$
$\{\!|\sigma.\ List\ \acute{}p\ \acute{}next\ Ps \wedge\ List\ \acute{}q\ \acute{}next\ Qs \wedge set\ Ps \cap set\ Qs = \{\}|\!\}$
  ´p :== PROC append(´p,´q)
$\{\!|List\ \acute{}p\ \acute{}next\ (Ps@Qs) \wedge (\forall x.\ x \notin set\ Ps \longrightarrow \acute{}next\ x = {}^{\sigma}next\ x)|\!\}$
⟨*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:

$islist\ p\ h = (\exists ps.\ List\ p\ h\ ps)$
$list\ p\ h = (THE\ ps.\ List\ p\ h\ ps)$

lemma $List\ p\ h\ ps = (islist\ p\ h \wedge ps = list\ p\ h)$

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.

⟨*ML*⟩

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 $(\forall\sigma.\ \Gamma\vdash_t \{\sigma\}\ PROC\ pedal(\acute{}N,\acute{}M)\ UNIV) \wedge$
$(\forall\sigma.\ \Gamma\vdash_t \{\sigma\}\ PROC\ coast(\acute{}N,\acute{}M)\ UNIV)$
⟨*proof*⟩

We can achieve the same effect without <∗*mlex*∗> by using the ordinary lexicographic product <∗*lex*∗>, *inv-image* and *measure*

lemma (**in** *pedal-coast-clique*)
shows $(\forall\sigma.\ \Gamma\vdash_t \{\sigma\}\ PROC\ pedal(\acute{}N,\acute{}M)\ UNIV) \wedge$
$(\forall\sigma.\ \Gamma\vdash_t \{\sigma\}\ PROC\ coast(\acute{}N,\acute{}M)\ UNIV)$
⟨*proof*⟩

By doing some arithmetic we can express the termination condition with a single measure function.

lemma (**in** *pedal-coast-clique*)
shows $(\forall\sigma.\ \Gamma\vdash_t \{\sigma\}\ PROC\ pedal(\acute{}N,\acute{}M)\ UNIV) \wedge$
$(\forall\sigma.\ \Gamma\vdash_t \{\sigma\}\ PROC\ coast(\acute{}N,\acute{}M)\ UNIV)$
⟨*proof*⟩

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 $\acute{}M :==_g (\acute{}M + 1)\ div\ \acute{}N$ gets expanded to guarded command ($False$, $\{\!|in\text{-}range\ (\acute{}M + 1) \wedge \acute{}N \neq 0 \wedge in\text{-}range\ ((\acute{}M + 1)\ div\ \acute{}N)|\!\}$)$\longmapsto \acute{}M :== (\acute{}M + 1)\ div\ \acute{}N$. Here *in-range* is uninterpreted by now.

lemma (**in** *vars*) $\Gamma\vdash\{\!|True|\!\}\ \acute{}M :==_g (\acute{}M + 1)\ div\ \acute{}N\ \{\!|True|\!\}$
⟨*proof*⟩

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: $\ldots :==_g \ldots$
- If: $IF_g \ldots$
- While: $WHILE_g \ldots$
- Call: $CALL_g \ldots$ or $\ldots :== CALL_g \ldots$

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* ⇒ '*a* ⇒ *bool*) ⇒ '*a list* ⇒ *bool*
where
sorted le [] = *True* |
sorted le (*x*#*xs*) = ((∀ *y*∈*set xs*. *le x y*) ∧ *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 ≤ ´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*:
  $\forall \sigma.\ \Gamma\vdash_{/UNIV} \{\sigma\}$ *´p :== PROC insert(´r,´p)*

$\{t.\ t\ \mathit{may\text{-}only\text{-}modify\text{-}globals}\ \sigma\ \mathit{in}\ [\mathit{next}]\}$
⟨*proof*⟩

lemma (**in** *insert-impl*) *insert-spec*:
$\forall \sigma\ Ps\ .\ \Gamma\vdash$
$\{\!|\sigma.\ \mathit{List}\ ´p\ ´\mathit{next}\ Ps \wedge \mathit{sorted}\ (\leq)\ (\mathit{map}\ ´\mathit{cont}\ Ps) \wedge$
$´r \neq \mathit{Null} \wedge ´r \notin \mathit{set}\ Ps|\!\}$
$´p :== \mathit{PROC}\ \mathit{insert}(´r,´p)$
$\{\!|\exists\ Qs.\ \mathit{List}\ ´p\ ´\mathit{next}\ Qs \wedge \mathit{sorted}\ (\leq)\ (\mathit{map}\ {}^{\sigma}\mathit{cont}\ \ Qs) \wedge$
$\mathit{set}\ Qs = \mathit{insert}\ {}^{\sigma}r\ (\mathit{set}\ Ps) \wedge$
$(\forall x.\ x \notin \mathit{set}\ Qs \longrightarrow ´\mathit{next}\ x = {}^{\sigma}\mathit{next}\ x)|\!\}$
⟨*proof*⟩

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 ${}^{\sigma}\mathit{cont}$, 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. Thats 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:

$$\mathit{HoarePartial.annotateI}\colon \frac{\Gamma,\Theta\vdash_{/F} P\ \mathit{anno}\ Q,A \qquad c = \mathit{anno}}{\Gamma,\Theta\vdash_{/F} P\ c\ Q,A}$$

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:

$$\mathit{HoarePartial.annotate\text{-}normI}\colon \frac{\begin{array}{c}\Gamma,\Theta\vdash_{/F} P\ \mathit{anno}\ Q,A\\ \mathit{Language.normalize}\ c = \mathit{Language.normalize}\ \mathit{anno}\end{array}}{\Gamma,\Theta\vdash_{/F} P\ c\ Q,A}$$

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 may-only-modify-globals* σ *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 ∀ σ *Ps*.
Γ⊢ ⦃σ. *List* ´*p* ´*next Ps* ⦄
´*p* :== *PROC insertSort*(´*p*)
⦃∃ *Qs*. *List* ´*p* ´*next Qs* ∧ *sorted* (≤) (*map* $^{\sigma}$*cont Qs*) ∧
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*,

$'p$, $'f$) *com*. All entities formally depend on an auxiliary (logical) variable of type $'a$. 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:
$[\![P \subseteq \{s \mid \exists Z.\ s \in P'\ Z \wedge Q'\ Z \subseteq Q \wedge A'\ Z \subseteq A\};\ \forall Z.\ \Gamma,\Theta\vdash_{/F} (P'\ Z)\ c\ Z\ (Q'\ Z),(A'\ Z);\ \forall Z.\ c\ Z = c\ \mathit{undefined}]\!] \Longrightarrow \Gamma,\Theta\vdash_{/F} P\ \mathit{specAnno}\ P'\ c\ Q'\ A'\ Q,A$

The side-condition $\forall Z.\ c\ Z = c\ \mathit{undefined}$ 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*) $\Gamma\vdash \{\sigma\}$
$\quad$ $\acute{}I$:== $\acute{}M$;;
$\quad$ *ANNO* τ. $\{\!|\tau.\ \acute{}I = {}^{\sigma}M|\!\}$
$\quad\quad$ $\acute{}M$:== $\acute{}N$;; $\acute{}N$:== $\acute{}I$
$\quad\quad$ $\{\!|\acute{}M = {}^{\tau}N \wedge \acute{}N = {}^{\tau}I|\!\}$
$\quad$ $\{\!|\acute{}M = {}^{\sigma}N \wedge \acute{}N = {}^{\sigma}M|\!\}$

With the annotation we can name an intermediate state τ. Since the postcondition refers to σ we have to link the information about the equivalence of ${}^{\tau}I$ and ${}^{\sigma}M$ in the specification in order to be able to derive the postcondition.

⟨*proof*⟩

lemma (**in** *vars*)
$\Gamma\vdash \{\sigma\}$
$\acute{}I$:== $\acute{}M$;;
ANNO τ. $\{\!|\tau.\ \acute{}I = {}^{\sigma}M|\!\}$
$\quad$ $\acute{}M$:== $\acute{}N$;; $\acute{}N$:== $\acute{}I$
$\quad$ $\{\!|\acute{}M = {}^{\tau}N \wedge \acute{}N = {}^{\tau}I|\!\}$
$\{\!|\acute{}M = {}^{\sigma}N \wedge \acute{}N = {}^{\sigma}M|\!\}$
⟨*proof*⟩

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 τ. You can introduce multiple variables by using a tuple:

lemma (**in** *vars*)
$\Gamma\vdash \{\sigma\}$
$\acute{}I$:== $\acute{}M$;;

$ANNO\ (n,i,m).\ \{\!|\ ´I = {}^{\sigma}M \land ´N{=}n \land ´I{=}i \land ´M{=}m |\!\}$
$´M :== ´N;;\ ´N :== ´I$
$\{\!|\ ´M = n \land ´N = i |\!\}$
$\{\!|\ ´M = {}^{\sigma}N \land ´N = {}^{\sigma}M |\!\}$
⟨*proof*⟩

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*:
$\forall\, n\ m.\ \Gamma \vdash \{\!|\ ´N = n \land ´M = m |\!\}\ ´N :== ´N + 1;;\ ´M :== ´M + 1$
$\{\!|\ ´N = n + 1 \land ´M = m + 1 |\!\}$
⟨*proof*⟩

lemma (**in** *vars*)
$\Gamma \vdash \{\!|\ ´N = n \land ´M = m |\!\}$
LEMMA foo-lemma
$´N :== ´N + 1;;\ ´M :== ´M + 1$
END;;
$´N :== ´N + 1$
$\{\!|\ ´N = n + 2 \land ´M = m + 1 |\!\}$
⟨*proof*⟩

lemma (**in** *vars*)
$\Gamma \vdash \{\!|\ ´N = n \land ´M = m |\!\}$
LEMMA foo-lemma
$´N :== ´N + 1;;\ ´M :== ´M + 1$
END;;
LEMMA foo-lemma
$´N :== ´N + 1;;\ ´M :== ´M + 1$
END
$\{\!|\ ´N = n + 2 \land ´M = m + 2 |\!\}$
⟨*proof*⟩

lemma (**in** *vars*)
$\Gamma \vdash \{\!|\ ´N = n \land ´M = m |\!\}$
$´N :== ´N + 1;;\ ´M :== ´M + 1;;$
$´N :== ´N + 1;;\ ´M :== ´M + 1$
$\{\!|\ ´N = n + 2 \land ´M = m + 2 |\!\}$
⟨*proof*⟩

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*:

$$[\![P \subseteq \{s \mid \exists Z.\ s \in I\ Z \wedge (\forall t.\ t \in I\ Z \cap -\ b \longrightarrow t \in Q)\};\ \forall Z\ \sigma.\ \Gamma,\Theta\vdash_{t/F} (\{\sigma\} \cap I\ Z \cap b)\ c\ Z\ (\{t \mid (t,\sigma) \in V\ Z\} \cap I\ Z),A;\ \forall Z.\ c\ Z = c\ \mathit{undefined};\ \forall Z.\ \mathit{wf}\ (V\ Z)]\!] \Longrightarrow \Gamma,\Theta\vdash_{t/F}\ P\ \mathit{whileAnnoFix}\ b\ I\ V\ c\ Q,A$$

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*)
  $\Gamma\vdash_t$ {| ´*M*=*0* ∧ ´*N*=*0* |}
    WHILE (´*M* < *i*)
    INV {| ´*M* ≤ *i* ∧ (´*M* ≠ *0* ⟶ ´*N* = *j*) ∧ ´*N* ≤ *j* |}
    VAR MEASURE (*i* − ´*M*)

```
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* σ *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 $proc_1$ and $proc_2$ to be "leaf" procedures, which do not call any other procedure. Procedure *proc* directly calls $proc_1$ and $proc_2$.

lemma (**in** $proc_1$*-impl*) $proc_1$*-modifies*:
shows ...

After the proof of $proc_1$*-modifies*, the **in** directive stores the lemma in the locale $proc_1$*-impl*. When we later on include $proc_1$*-impl* or prove another theorem in locale $proc_1$*-impl* the lemma $proc_1$*-modifies* will already be available as fact.

lemma (**in** $proc_1$*-impl*) $proc_1$*-spec*:
shows ...

lemma (**in** $proc_2$*-impl*) $proc_2$*-modifies*:
shows ...

lemma (**in** $proc_2$*-impl*) $proc_2$*-spec*:
shows ...

lemma (**in** *proc-impl*) *proc-modifies*:
shows ...

Note that we do not explicitly include anything about $proc_1$ or $proc_2$ 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 $proc_1$*-impl* and $proc_2$*-impl* are imported. This has the neat effect that all theorems that are proven in $proc_1$*-impl* and $proc_2$*-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 inheritance. 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* = $^{\sigma}$*next 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 *´p :== NULL* for a procedure *init*.

procedures *init* (|*p*) =
  ´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 *′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 *´p :== NULL* in the first place by adding a typing annotation:

procedures *init′* (|*p*) =
  ´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 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.