pGCL for Isabelle

David Cock

March 17, 2025

Contents

1 Overview **1**

2 Introduction to pGCL **3**

2.1 Language Primitives 3

2.1.1 The Basics 3

2.1.2 Assertion and Annotation 4

2.1.3 Probability 4

2.1.4 Nondeterminism 5

2.1.5 Properties of Expectations 5

2.2 Loops 6

2.2.1 Guaranteed Termination 6

2.2.2 Probabilistic Termination 8

2.3 The Monty Hall Problem 9

2.3.1 The State Space 9

2.3.2 The Game 10

2.3.3 A Brute Force Solution 11

2.3.4 A Modular Approach 11

3 Semantic Structures **17**

3.1 Expectations 17

3.1.1 Bounded Functions 18

3.1.2 Non-Negative Functions. 20

3.1.3 Sound Expectations 21

3.1.4 Unitary expectations 24

3.1.5 Standard Expectations 25

3.1.6 Entailment 27

3.1.7 Expectation Conjunction 28

3.1.8 Rules Involving Conjunction. 29

3.1.9 Rules Involving Entailment and Conjunction Together . . 31

3.2 Expectation Transformers 33

3.2.1 Comparing Transformers 35

3.2.2 Healthy Transformers 39

3.2.3 Sublinearity 46

3.2.4 Determinism . . . 56
3.2.5 Modular Reasoning . . . 59
3.2.6 Transforming Standard Expectations . . . 60
3.3 Induction . . . 62
3.3.1 The Lattice of Expectations . . . 62
3.3.2 The Lattice of Transformers . . . 66
3.3.3 Tail Recursion . . . 71

4 The pGCL Language 75
4.1 A Shallow Embedding of pGCL in HOL . . . 75
4.1.1 Core Primitives and Syntax . . . 75
4.1.2 Unfolding rules for non-recursive primitives . . . 79
4.2 Healthiness . . . 81
4.2.1 The Healthiness of the Embedding . . . 81
4.2.2 Healthiness for Loops . . . 94
4.3 Continuity . . . 104
4.3.1 Continuity of Primitives . . . 105
4.3.2 Continuity of a Single Loop Step . . . 122
4.4 Continuity and Induction for Loops . . . 125
4.4.1 The Limit of Iterates . . . 133
4.5 Sublinearity . . . 142
4.5.1 Nonrecursive Primitives . . . 143
4.5.2 Sublinearity for Loops . . . 148
4.6 Determinism . . . 152
4.6.1 Additivity . . . 152
4.6.2 Maximality . . . 153
4.6.3 Determinism . . . 155
4.7 Well-Defined Programs. . . . 156
4.7.1 Strict Implies Liberal . . . 156
4.7.2 Sub-Distributivity of Conjunction . . . 161
4.7.3 The Well-Defined Predicate. . . . 170
4.8 The Loop Rules . . . 173
4.8.1 Liberal and Strict Invariants. . . . 173
4.8.2 Partial Correctness . . . 174
4.8.3 Total Correctness . . . 174
4.8.4 Unfolding . . . 175
4.9 The Algebra of pGCL . . . 176
4.9.1 Program Refinement . . . 176
4.9.2 Simple Identities . . . 177
4.9.3 Deterministic Programs are Maximal . . . 182
4.9.4 The Algebraic Structure of Refinement . . . 184
4.9.5 Data Refinement . . . 187
4.9.6 The Algebra of Data Refinement . . . 191
4.9.7 Structural Rules for Correspondence . . . 198

4.9.8 Structural Rules for Data Refinement 199
4.10 Structured Reasoning 201
4.10.1 Syntactic Decomposition 202
4.10.2 Algebraic Decomposition 206
4.10.3 Hoare triples 207
4.11 Loop Termination 207
4.11.1 Trivial Termination 207
4.11.2 Classical Termination 208
4.11.3 Probabilistic Termination 211
4.12 Automated Reasoning 214

Additional Material **217**
4.13 Miscellaneous Mathematics 217
4.13.1 Truncated Subtraction 223

Chapter 1

Overview

pGCL is both a programming language and a specification language that incorporates both probabilistic and nondeterministic choice, in a unified manner. Program verification is by *refinement* or *annotation* (or both), using either Hoare triples, or weakest-precondition entailment, in the style of GCL [Dijkstra, 1975].

This document is divided into three parts: Chapter 2 gives a tutorial-style introduction to pGCL, and demonstrates the tools provided by the package; Chapter 3 covers the development of the semantic interpretation: *expectation transformers*; and Chapter 4 covers the formalisation of the language primitives, the associated *healthiness* results, and the tools for structured and automated reasoning. This second part follows the technical development of the pGCL theory package, in detail. It is not a great place to start learning pGCL. For that, see either the tutorial or McIver and Morgan [2004].

This formalisation was first presented (as an overview) in Cock [2012]. The language has previously been formalised in HOL4 by Hurd et al. [2005]. Two substantial results using this package were presented in Cock [2013], Cock [2014a] and Cock [2014b].

Chapter 2

Introduction to pGCL

2.1 Language Primitives

theory *Primitives* **imports** *../pGCL* **begin**

Programs in pGCL are probabilistic automata. They can do anything a traditional program can, plus, they may make truly probabilistic choices.

2.1.1 The Basics

Imagine flipping a pair of fair coins: *a* and *b*. Using a record type for the state allows a number of syntactic niceties, which we describe shortly:

datatype *coin* = *Heads* | *Tails*

record *coins* =
 a :: *coin*
 b :: *coin*

The primitive state operation is *Apply*, which takes a state transformer as an argument, constructs the pGCL equivalent. Thus *Apply* (*a-update* (λ-. *Heads*)) sets the value of coin *a* to *Heads*. As records are so common as state types, we introduce syntax to make these update neater: The same program may be defined more simply as *Apply* (*a-update* (λ-. *Heads*)) (note that the syntax translation involved does not apply to Latex output, and thus this lemma appears trivial):

lemma
 Apply ($\lambda s.\ s\ (\!|\ a := Heads\ |\!)$) = ($a := (\lambda s.\ Heads)$)
 by(*simp*)

We can treat the record's fields as the names of *variables*. Note that the right-hand side of an assignment is always a function of the current state. Thus we may use a record accessor directly, for example *Apply* ($\lambda s.\ s(\!|a := b\ s|\!)$), which updates *a* with the current value of *b*. If we wish to formally establish that the previous statement is correct i.e. that in the final state, *a* really will have whatever value *b* had in the initial state, we must first introduce the assertion language.

2.1.2 Assertion and Annotation

Assertions in pGCL are real-valued functions of the state, which are often interpreted as a probability distribution over possible outcomes. These functions are termed *expectations*, for reasons which shortly be clear. Initially, however, we need only consider *standard* expectations: those derived from a binary predicate. A predicate $P{::}'s \Rightarrow bool$ is embedded as $«\ P\ »{::}'s \Rightarrow real$, such that $P\ s \longrightarrow «\ P\ »\ s = 1 \wedge \neg\ P\ s \longrightarrow «\ P\ »\ s = 0$.

An annotation consists of an assertion on the initial state and one on the final state, which for standard expectations may be interpreted as 'if *P* holds in the initial state, then *Q* will hold in the final state'. These are in weakest-precondition form: we assert that the precondition implies the *weakest precondition*: the weakest assertion on the initial state, which implies that the postcondition must hold on the final state. So far, this is identical to the standard approach. Remember, however, that we are working with *real-valued* assertions. For standard expectations, the logic is nevertheless identical, if the implication $\forall\, s.\ P\ s \longrightarrow Q\ s$ is substituted with the equivalent expectation entailment $«\ P\ » \Vdash «\ Q\ »$, $[\![«\ ?P\ » \Vdash «\ ?Q\ »;\ ?P\ ?s]\!] \Longrightarrow\ ?Q\ ?s$. Thus a valid specification of $Apply\ (\lambda s.\ s(\!|a := b\ s|\!))$ is:

lemma
$\bigwedge x.\ «\lambda s.\ b\ s = x» \Vdash wp\ (a := b)\ «\lambda s.\ a\ s = x»$
by(*pvcg*, *simp add*:*o-def*)

Any ordinary computation and its associated annotation can be expressed in this form.

2.1.3 Probability

Next, we introduce the syntax *x* ;; *y* for the sequential composition of *x* and *y*, and also demonstrate that one can operate directly on a real-valued (and thus infinite) state space:

lemma
$«\lambda s{::}real.\ s \neq 0» \Vdash wp\ (Apply\ ((*)\ 2)\ ;;\ Apply\ (\lambda s.\ s\ /\ s))\ «\lambda s.\ s = 1»$
by(*pvcg*, *simp add*:*o-def*)

So far, we haven't done anything that required probabilities, or expectations other than 0 and 1. As an example of both, we show that a single coin toss is fair. We introduce the syntax $x\ {}_p\!\oplus\ y$ for a probabilistic choice between *x* and *y*. This program behaves as *x* with probability *p*, and as *y* with probability $1 - p$. The probability may depend on the state, and is therefore of type $'s \Rightarrow real$. The following annotation states that the probability of heads is exactly 1/2:

definition

flip-a :: *real* ⇒ *coins prog*
where
flip-a p = *a* := (λ-. *Heads*) $_{(\lambda s.\ p)}\oplus$ *a* := (λ-. *Tails*)

lemma
(λ*s*. *1/2*) = *wp* (*flip-a* (*1/2*)) «λ*s*. *a s* = *Heads*»
unfolding *flip-a-def*

Sufficiently small problems can be handled by the simplifier, by symbolic evaluation.

by(*simp add*:*wp-eval o-def*)

2.1.4 Nondeterminism

We can also under-specify a program, using the *nondeterministic choice* operator, $x \sqcap y$. This is interpreted demonically, giving the pointwise *minimum* of the pre-expectations for *x* and *y*: the chance of seeing heads, if your opponent is allowed choose between a pair of coins, one biased 2/3 heads and one 2/3 tails, and then flips it, is *at least* 1/3, but we can make no stronger statement:

lemma
λ*s*. *1/3* ⊩ *wp* (*flip-a* (*2/3*) ⊓ *flip-a* (*1/3*)) «λ*s*. *a s* = *Heads*»
unfolding *flip-a-def*
by *pvcg*

2.1.5 Properties of Expectations

The probabilities of independent events combine as usual, by multiplying: The chance of getting heads on two separate coins is *1* / (*4*::′*a*).

definition
flip-b :: *real* ⇒ *coins prog*
where
flip-b p = *b* := (λ-. *Heads*) $_{(\lambda s.\ p)}\oplus$ *b* := (λ-. *Tails*)

lemma
(λ*s*. *1/4*) = *wp* (*flip-a* (*1/2*) ;; *flip-b* (*1/2*))
«λ*s*. *a s* = *Heads* ∧ *b s* = *Heads*»
unfolding *flip-a-def flip-b-def*
by(*simp add*:*wp-eval o-def*)

If, rather than two coins, we use two dice, we can make some slightly more involved calculations. We see that the weakest pre-expectation of the value on the face of the die after rolling is its *expected value* in the initial state, which justifies the use of the term expectation.

record *dice* =
red :: *nat*
blue :: *nat*

definition *Puniform* :: *'a set* ⇒ (*'a* ⇒ *real*)
where *Puniform S* = (λ*x. if x* ∈ *S then 1* / *card S else 0*)

lemma *Puniform-in*:
 x ∈ *S* ⟹ *Puniform S x* = *1* / *card S*
 by(*simp add*:*Puniform-def*)

lemma *Puniform-out*:
 x ∉ *S* ⟹ *Puniform S x* = *0*
 by(*simp add*:*Puniform-def*)

lemma *supp-Puniform*:
 finite S ⟹ *supp* (*Puniform S*) = *S*
 by(*auto simp*:*Puniform-def supp-def*)

The expected value of a roll of a six-sided die is (*7*::*'a*) / (*2*::*'a*):

lemma
 (λ*s. 7/2*) = *wp* (*bind v at* (λ*s. Puniform* {*1..6*} *v*) *in red* := (λ*-. v*)) *red*
 by(*simp add*:*wp-eval supp-Puniform sum.atLeast-Suc-atMost Puniform-in*)

The expectations of independent variables add:

lemma
 (λ*s. 7*) = *wp* ((*bind v at* (λ*s. Puniform* {*1..6*} *v*) *in red* := (λ*s. v*)) ;;
  (*bind v at* (λ*s. Puniform* {*1..6*} *v*) *in blue* := (λ*s. v*)))
  (λ*s. red s* + *blue s*)
 by(*simp add*:*wp-eval supp-Puniform sum.atLeast-Suc-atMost Puniform-in*)

end

2.2 Loops

theory *LoopExamples* **imports** *../pGCL* **begin**

Reasoning about loops in pGCL is mostly familiar, in particular in the use of invariants. Proving termination for truly probabilistic loops is slightly different: We appeal to a 0–1 law to show that the loop terminates *with probability 1*. In our semantic model, terminating with certainty and with probability 1 are exactly equivalent.

2.2.1 Guaranteed Termination

We start with a completely classical loop, to show that standard techniques apply. Here, we have a program that simply decrements a counter until it hits zero:

definition *countdown* :: *int prog*
where
 countdown = *do* (λ*x. 0* < *x*) ⟶ *Apply* (λ*s. s* − *1*) *od*

Clearly, this loop will only terminate from a state where $0 \leq x$. This is, in fact, also a loop invariant.

definition *inv-count* :: *int* $\Rightarrow$ *bool*
where
inv-count = ($\lambda x.$ $0 \leq x$)

Read *wp-inv G body I* as: *I* is an invariant of the loop $\mu x.$ *body* ;; *x* $_{\text{« } G \text{ »}}\oplus$ *Skip*, or « *G* » && *I* ⊩ *wp body I*.

lemma *wp-inv-count*:
wp-inv ($\lambda x.$ $0 < x$) (*Apply* ($\lambda s.$ $s - 1$)) «*inv-count*»
unfolding *wp-inv-def inv-count-def wp-eval o-def*
proof(*clarify*, *cases*)
fix *x*::*int*
assume $0 \leq x$
then show «$\lambda x.$ $0 < x$» $x * $ «$\lambda x.$ $0 \leq x$» $x \leq$ «$\lambda x.$ $0 \leq x$» $(x - 1)$
by(*simp add*:*embed-bool-def*)
next
fix *x*::*int*
assume $\neg$ $0 \leq x$
then show «$\lambda x.$ $0 < x$» $x * $ «$\lambda x.$ $0 \leq x$» $x \leq$ «$\lambda x.$ $0 \leq x$» $(x - 1)$
by(*simp add*:*embed-bool-def*)
qed

This example is contrived to give us an obvious variant, or measure function: the counter itself.

lemma *term-countdown*:
«*inv-count*» ⊩ *wp countdown* ($\lambda s.$ 1)
unfolding *countdown-def*
proof(*intro loop-term-nat-measure*[**where** *m*=$\lambda x.$ *nat* (*max x 0*)] *wp-inv-count*)
let *?p* = *Apply* ($\lambda x.$ $x - 1$::*int*)

As usual, well-definedness is trivial.

show *well-def ?p*
by(*rule wd-intros*)

A measure of 0 imples termination.

show $\bigwedge x.$ *nat* (*max x 0*) = $0 \longrightarrow \neg$ $0 < x$
by(*auto*)

This is the meat of the proof: that the measure must decrease, whenever the invariant holds. Note that the invariant is essential here, as if $x \leq 0$, the measure will *not* decrease.

This is the kind of proof that the VCG is good at. It leaves a purely logical goal, which we can solve with auto.

show $\bigwedge n.$ «$\lambda x.$ *nat* (*max x 0*) = *Suc n*» && «*inv-count*» ⊩
wp ?p «$\lambda x.$ *nat* (*max x 0*) = *n*»
unfolding *inv-count-def*
by(*pvcg*,

auto simp: *o-def exp-conj-std-split*[*symmetric*]
intro: *implies-entails*)
qed

2.2.2 Probabilistic Termination

Loops need not terminate deterministically: it is sufficient to terminate with probability 1. Here we show the intuitively obvious result that by flipping a coin repeatedly, you will eventually see heads.

type-synonym *coin* = *bool*
definition *Heads* = *True*
definition *Tails* = *False*

definition
flip :: *coin prog*
where
flip = *Apply* (λ-. *Heads*) $_{(\lambda s.\ 1/2)}\oplus$ *Apply* (λ-. *Tails*)

We can't define a measure here, as we did previously, as neither of the two possible states guarantee termination.

definition
wait-for-heads :: *coin prog*
where
wait-for-heads = *do* ((≠) *Heads*) ⟶ *flip od*

Nonetheless, we can show termination .

lemma *wait-for-heads-term*:
λ*s*. *1* ⊩ *wp wait-for-heads* (λ*s*. *1*)
unfolding *wait-for-heads-def*

We use one of the zero-one laws for termination, specifically that if, from every state there is a nonzero probability of satisfying the guard (and thus terminating) in a single step, then the loop terminates from *any* state, with probability 1.

proof(*rule termination-0-1*)
show *well-def flip*
unfolding *flip-def*
by(*auto intro*:*wd-intros*)

We must show that the loop body is deterministic, meaning that it cannot diverge by itself.

show *maximal* (*wp flip*)
unfolding *flip-def* **by**(*auto intro*:*max-intros*)

The verification condition for the loop body is one-step-termination, here shown to hold with probability 1/2. As usual, this result falls to the VCG.

show λ*s*. *1*/*2* ⊩ *wp flip* «𝒩 ((≠) *Heads*)»
unfolding *flip-def*
by(*pvcg*, *simp add*:*o-def Heads-def Tails-def*)

Finally, the one-step escape probability is non-zero.

show $(0::real) < 1/2$ **by**(*simp*)
qed

end

2.3 The Monty Hall Problem

theory *Monty* **imports** *../pGCL* **begin**

We now tackle a more substantial example, allowing us to demonstrate the tools for compositional reasoning and the use of invariants in non-recursive programs. Our example is the well-known Monty Hall puzzle in statistical inference [Selvin, 1975].

The setting is a game show: There is a prize hidden behind one of three doors, and the contestant is invited to choose one. Once the guess is made, the host than opens one of the remaining two doors, revealing a goat and showing that the prize is elsewhere. The contestent is then given the choice of switching their guess to the other unopened door, or sticking to their first guess.

The puzzle is whether the contestant is better off switching or staying put; or indeed whether it makes a difference at all. Most people's intuition suggests that it make no difference, whereas in fact, switching raises the chance of success from 1/3 to 2/3.

2.3.1 The State Space

The game state consists of the prize location, the guess, and the clue (the door the host opens). These are not constrained a priori to the range $\{1, 2, 3\}$, but are simply natural numbers: We instead show that this is in fact an invariant.

record *game* =
prize :: *nat*
guess :: *nat*
clue :: *nat*

The victory condition: The player wins if they have guessed the correct door, when the game ends.

definition *player-wins* :: *game* ⇒ *bool*
where *player-wins g* ≡ *guess g* = *prize g*

Invariants

We prove explicitly that only valid doors are ever chosen.

definition *inv-prize* :: *game* ⇒ *bool*
where *inv-prize g* ≡ *prize g* ∈ $\{1,2,3\}$

definition *inv-clue* :: *game* ⇒ *bool*
where *inv-clue g* ≡ *clue g* ∈ {*1*,*2*,*3*}

definition *inv-guess* :: *game* ⇒ *bool*
where *inv-guess g* ≡ *guess g* ∈ {*1*,*2*,*3*}

2.3.2 The Game

Hide the prize behind door *D*.

definition *hide-behind* :: *nat* ⇒ *game prog*
where *hide-behind D* ≡ *Apply* (*prize-update* (λ*x*. *D*))

Choose door *D*.

definition *guess-behind* :: *nat* ⇒ *game prog*
where *guess-behind D* ≡ *Apply* (*guess-update* (λ*x*. *D*))

Open door *D* and reveal what's behind.

definition *open-door* :: *nat* ⇒ *game prog*
where *open-door D* ≡ *Apply* (*clue-update* (λ*x*. *D*))

Hide the prize behind door 1, 2 or 3, demonically i.e. according to any probability distribution (or none).

definition *hide-prize* :: *game prog*
where *hide-prize* ≡ *hide-behind 1* ⨅ *hide-behind 2* ⨅ *hide-behind 3*

Guess uniformly at random.

definition *make-guess* :: *game prog*
where *make-guess* ≡ *guess-behind 1* $_{(\lambda s.\ 1/3)}\oplus$
guess-behind 2 $_{(\lambda s.\ 1/2)}\oplus$ *guess-behind 3*

Open one of the two doors that *doesn't* hide the prize.

definition *reveal* :: *game prog*
where *reveal* ≡ ⨅*d*∈(λ*s*. {*1*,*2*,*3*} − {*prize s*, *guess s*}). *open-door d*

Switch your guess to the other unopened door.

definition *switch-guess* :: *game prog*
where *switch-guess* ≡ ⨅*d*∈(λ*s*. {*1*,*2*,*3*} − {*clue s*, *guess s*}). *guess-behind d*

The complete game, either with or without switching guesses.

definition *monty* :: *bool* ⇒ *game prog*
where
monty switch ≡ *hide-prize* ;;
make-guess ;;
reveal ;;
(*if switch then switch-guess else Skip*)

2.3.3 A Brute Force Solution

For sufficiently simple programs, we can calculate the exact weakest pre-expectation by unfolding.

lemma *eval-win*[*simp*]:
 $p = g \Longrightarrow$ «*player-wins*» (s(| *prize* := p, *guess* := g, *clue* := c |)) = *1*
 by(*simp add*:*embed-bool-def player-wins-def*)

lemma *eval-loss*[*simp*]:
 $p \neq g \Longrightarrow$ «*player-wins*» (s(| *prize* := p, *guess* := g, *clue* := c |)) = *0*
 by(*simp add*:*embed-bool-def player-wins-def*)

If they stick to their guns, the player wins with $p = 1/3$.

lemma *wp-monty-noswitch*:
 (λs. *1/3*) = *wp* (*monty False*) «*player-wins*»
 unfolding *monty-def hide-prize-def make-guess-def reveal-def*
  hide-behind-def guess-behind-def open-door-def
  switch-guess-def
 by(*simp add*:*wp-eval insert-Diff-if o-def*)

lemma *swap-upd*:
 s(| *prize* := p, *clue* := c, *guess* := g |) =
 s(| *prize* := p, *guess* := g, *clue* := c |)
 by(*simp*)

If they switch, they win with $p = 2/3$. Brute force here takes longer, but is still feasible. On larger programs, this will rapidly become impossible, as the size of the terms (generally) grows exponentially with the length of the program.

lemma *wp-monty-switch-bruteforce*:
 (λs. *2/3*) = *wp* (*monty True*) «*player-wins*»
 unfolding *monty-def hide-prize-def make-guess-def reveal-def*
  hide-behind-def guess-behind-def open-door-def
  switch-guess-def
 — Note that this is getting slow
 by (*simp add*: *wp-eval insert-Diff-if swap-upd o-def cong del*: *INF-cong-simp*)

2.3.4 A Modular Approach

We can solve the problem more efficiently, at the cost of a little more user effort, by breaking up the problem and annotating each step of the game separately. While this is not strictly necessary for this program, it will scale to larger examples, as the work in annotation only increases linearly with the length of the program.

Healthiness

We first establish healthiness for each step. This follows straightforwardly by applying the supplied rulesets.

lemma *wd-hide-prize*:
well-def hide-prize
unfolding *hide-prize-def hide-behind-def*
by(*simp add*:*wd-intros*)

lemma *wd-make-guess*:
well-def make-guess
unfolding *make-guess-def guess-behind-def*
by(*simp add*:*wd-intros*)

lemma *wd-reveal*:
well-def reveal
proof −

Here, we do need a subsidiary lemma: that there is always a 'fresh' door available. The rest of the healthiness proof follows as usual.

have $\bigwedge s.\ \{1, 2, 3\} - \{prize\ s, guess\ s\} \neq \{\}$
by(*auto simp*:*insert-Diff-if*)
thus *?thesis*
unfolding *reveal-def open-door-def*
by(*intro wd-intros*, *auto*)
qed

lemma *wd-switch-guess*:
well-def switch-guess
proof −
have $\bigwedge s.\ \{1, 2, 3\} - \{clue\ s, guess\ s\} \neq \{\}$
by(*auto simp*:*insert-Diff-if*)
thus *?thesis*
unfolding *switch-guess-def guess-behind-def*
by(*intro wd-intros*, *auto*)
qed

lemmas *monty-healthy* =
wd-switch-guess wd-reveal wd-make-guess wd-hide-prize

Annotations

We now annotate each step individually, and then combine them to produce an annotation for the entire program.

hide-prize chooses a valid door.

lemma *wp-hide-prize*:
$(\lambda s.\ 1) \Vdash wp\ hide\text{-}prize$ «*inv-prize*»
unfolding *hide-prize-def hide-behind-def wp-eval o-def*
by(*simp add*:*embed-bool-def inv-prize-def*)

Given the prize invariant, *make-guess* chooses a valid door, and guesses incorrectly with probability at least 2/3.

lemma *wp-make-guess*:
  $(\lambda s.\ 2/3 * «\lambda g.\ inv\text{-}prize\ g»\ s) \Vdash$
   wp make-guess $«\lambda g.\ guess\ g \neq prize\ g \wedge inv\text{-}prize\ g \wedge inv\text{-}guess\ g»$
  unfolding *make-guess-def guess-behind-def wp-eval o-def*
  by(*auto simp*:*embed-bool-def inv-prize-def inv-guess-def*)

lemma *last-one*:
  assumes $a \neq b$ **and** $a \in \{1{::}nat,2,3\}$ **and** $b \in \{1,2,3\}$
  shows $\exists!c.\ \{1,2,3\} - \{b,a\} = \{c\}$
  apply(*simp add*:*insert-Diff-if*)
  using *assms* **by**(*auto intro*:*assms*)

Given the composed invariants, and an incorrect guess, *reveal* will give a clue that is neither the prize, nor the guess.

lemma *wp-reveal*:
  $«\lambda g.\ guess\ g \neq prize\ g \wedge inv\text{-}prize\ g \wedge inv\text{-}guess\ g» \Vdash$
   wp reveal $«\lambda g.\ guess\ g \neq prize\ g \wedge$
        $clue\ g \neq prize\ g \wedge$
        $clue\ g \neq guess\ g \wedge$
        $inv\text{-}prize\ g \wedge inv\text{-}guess\ g \wedge inv\text{-}clue\ g»$
  (**is** *?X* $\Vdash$ *wp reveal ?Y*)
proof(*rule use-premise*, *rule well-def-wp-healthy*[*OF wd-reveal*], *clarify*)
  fix *s*
  assume $guess\ s \neq prize\ s$
    and *inv-prize s*
    and *inv-guess s*
  moreover then obtain *c*
   where *singleton*: $\{Suc\ 0,2,3\} - \{prize\ s, guess\ s\} = \{c\}$
    and $c \neq prize\ s$
    and $c \neq guess\ s$
    and $c \in \{Suc\ 0,2,3\}$
   unfolding *inv-prize-def inv-guess-def*
   by(*force dest*:*last-one elim*!:*ex1E*)
  ultimately show $1 \leq wp\ reveal\ ?Y\ s$
   by(*simp add*:*reveal-def open-door-def wp-eval singleton o-def*
      embed-bool-def inv-prize-def inv-guess-def inv-clue-def)
qed

Showing that the three doors are all district is a largeish first-order problem, for which sledgehammer gives us a reasonable script.

lemma *distinct-game*:
  $[\![\ guess\ g \neq prize\ g;\ clue\ g \neq prize\ g;\ clue\ g \neq guess\ g;$
   $inv\text{-}prize\ g;\ inv\text{-}guess\ g;\ inv\text{-}clue\ g\ ]\!] \Longrightarrow$
  $\{1,\ 2,\ 3\} = \{guess\ g,\ prize\ g,\ clue\ g\}$
  unfolding *inv-prize-def inv-guess-def inv-clue-def*
  apply(*rule set-eqI*)
  apply(*rule iffI*)
   apply(*clarify*)
   apply(*metis* (*full-types*) *empty-iff insert-iff*)

apply(*metis insert-iff*)
done

Given the invariants, switching from the wrong guess gives the right one.

lemma *wp-switch-guess*:
«λg. *guess g* $\neq$ *prize g* $\wedge$ *clue g* $\neq$ *prize g* $\wedge$ *clue g* $\neq$ *guess g* $\wedge$
inv-prize g $\wedge$ *inv-guess g* $\wedge$ *inv-clue g*» $\Vdash$
wp switch-guess «*player-wins*»
proof(*rule use-premise*, *safe*)
from *wd-switch-guess* **show** *healthy* (*wp switch-guess*) **by**(*auto*)

fix *s*
assume *guess s* $\neq$ *prize s* **and** *clue s* $\neq$ *prize s*
and *clue s* $\neq$ *guess s* **and** *inv-prize s*
and *inv-guess s* **and** *inv-clue s*
note *state* = *this*
hence $1 \leq$ *Inf* ((λa. « *player-wins* » (*s*(|*guess* := *a*|))) ‘
({*guess s*, *prize s*, *clue s*} − {*clue s*, *guess s*}))
by(*auto simp*:*insert-Diff-if player-wins-def*)
also from *state*
have ... = *Inf* ((λa. « *player-wins* » (*s*(|*guess* := *a*|))) ‘
({*1*, *2*, *3*} − {*clue s*, *guess s*}))
by(*simp add*:*distinct-game*[*symmetric*])
also have ... = *wp switch-guess* «*player-wins*» *s*
by(*simp add*:*switch-guess-def guess-behind-def wp-eval o-def*)
finally show $1 \leq$ *wp switch-guess* « *player-wins* » *s* **.**
qed

Given componentwise specifications, we can glue them together with calculational reasoning to get our result.

lemma *wp-monty-switch-modular*:
(λs. *2/3*) $\Vdash$ *wp* (*monty True*) «*player-wins*»
proof(*rule wp-validD*) — Work in probabilistic Hoare triples
note *wp-validI*[*OF wp-scale*, *OF wp-hide-prize*, *simplified*]
— Here we apply scaling to match our pre-expectation
also note *wp-validI*[*OF wp-make-guess*]
also note *wp-validI*[*OF wp-reveal*]
also note *wp-validI*[*OF wp-switch-guess*]
finally show {|λs. *2/3*|} *monty True* {|«*player-wins*»|}*p*
unfolding *monty-def*
by(*simp add*:*wd-intros sound-intros monty-healthy*)
qed

Using the VCG

lemmas *scaled-hide* = *wp-scale*[*OF wp-hide-prize*, *simplified*]
declare *scaled-hide*[*pwp*] *wp-make-guess*[*pwp*] *wp-reveal*[*pwp*] *wp-switch-guess*[*pwp*]
declare *wd-hide-prize*[*wd*] *wd-make-guess*[*wd*] *wd-reveal*[*wd*] *wd-switch-guess*[*wd*]

Alternatively, the VCG will get this using the same annotations.

lemma *wp-monty-switch-vcg*:
$(\lambda s.\ 2/3) \Vdash wp\ (monty\ True)$ «*player-wins*»
unfolding *monty-def*
by(*simp*, *pvcg*)

end

Chapter 3

Semantic Structures

3.1 Expectations

theory *Expectations* **imports** *Misc* **begin type-synonym** $'s\ expect = 's \Rightarrow real$

Expectations are a real-valued generalisation of boolean predicates: An expectation on state $'s$ is a function $'s \Rightarrow real$. A predicate P on $'s$ is embedded as an expectation by mapping *True* to 1 and *False* to 0. Under this embedding, implication becomes comparison, as the truth tables demonstrate:

a	b	$a \rightarrow b$	x	y	$x \leq y$
F	F	T	0	0	T
F	T	T	0	1	T
T	F	F	1	0	F
T	T	T	1	1	T

For probabilistic automata, an expectation gives the current expected value of some expression, if it were to be evaluated in the final state. For example, consider the automaton of Figure 3.1, with transition probabilities affixed to edges. Let $P\ b = 2.0$ and $P\ c = 3.0$. Both states b and c are final (accepting) states, and thus the 'final expected value' of P in state b is 2.0 and in state c is 3.0. The expected value from state a is the weighted sum of these, or $0.7 \times 2.0 + 0.3 \times 3.0 = 2.3$.

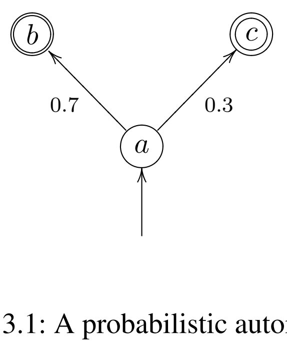

Figure 3.1: A probabilistic automaton

All expectations must be non-negative and bounded i.e. $\forall s.\ 0 \leq P\ s$ and $\exists b.\forall s. P\ s \leq b$. Note that although every expectation must have a bound, there is no bound on all expectations; In particular, the following series has no global bound, although each element is clearly bounded:

$$P_i = \lambda s.\ i \quad \text{where } i \in \mathbb{N}$$

3.1.1 Bounded Functions

definition *bounded-by* :: *real* ⇒ (*'a* ⇒ *real*) ⇒ *bool*
where *bounded-by b P* ≡ ∀ *x*. *P x* ≤ *b*

By instantiating the classical reasoner, both establishing and appealing to boundedness is largely automatic.

lemma *bounded-byI*[*intro*]:
⟦ ⋀*x*. *P x* ≤ *b* ⟧ ⟹ *bounded-by b P*
by (*simp add*:*bounded-by-def*)

lemma *bounded-byI2*[*intro*]:
P ≤ (λ*s*. *b*) ⟹ *bounded-by b P*
by (*blast dest*:*le-funD*)

lemma *bounded-byD*[*dest*]:
bounded-by b P ⟹ *P x* ≤ *b*
by (*simp add*:*bounded-by-def*)

lemma *bounded-byD2*[*dest*]:
bounded-by b P ⟹ *P* ≤ (λ*s*. *b*)
by (*blast intro*:*le-funI*)

A function is bounded if there exists at least one upper bound on it.

definition *bounded* :: (*'a* ⇒ *real*) ⇒ *bool*
where *bounded P* ≡ (∃ *b*. *bounded-by b P*)

In the reals, if there exists any upper bound, then there must exist a least upper bound.

definition *bound-of* :: (*'a* ⇒ *real*) ⇒ *real*
where *bound-of P* ≡ *Sup* (*P* ‘ *UNIV*)

lemma *bounded-bdd-above*[*intro*]:
assumes *bP*: *bounded P*
shows *bdd-above* (*range P*)
proof
fix *x* **assume** *x* ∈ *range P*
with *bP* **show** *x* ≤ *Inf* {*b*. *bounded-by b P*}
unfolding *bounded-def* **by**(*auto intro*:*cInf-greatest*)
qed

The least upper bound has the usual properties:

lemma *bound-of-least*[*intro*]:
assumes *bP*: *bounded-by b P*
shows *bound-of P* $\leq$ *b*
unfolding *bound-of-def*
using *bP* **by**(*intro cSup-least*, *auto*)

lemma *bounded-by-bound-of*[*intro*!]:
fixes *P*::*'a* $\Rightarrow$ *real*
assumes *bP*: *bounded P*
shows *bounded-by* (*bound-of P*) *P*
unfolding *bound-of-def*
using *bP* **by**(*intro bounded-byI cSup-upper bounded-bdd-above*, *auto*)

lemma *bound-of-greater*[*intro*]:
bounded P $\Longrightarrow$ *P x* $\leq$ *bound-of P*
by (*blast intro*:*bounded-byD*)

lemma *bounded-by-mono*:
$[\![$ *bounded-by a P*; *a* $\leq$ *b* $]\!]$ $\Longrightarrow$ *bounded-by b P*
unfolding *bounded-by-def* **by**(*blast intro*:*order-trans*)

lemma *bounded-by-imp-bounded*[*intro*]:
bounded-by b P $\Longrightarrow$ *bounded P*
unfolding *bounded-def* **by**(*blast*)

This is occasionally easier to apply:

lemma *bounded-by-bound-of-alt*:
$[\![$ *bounded P*; *bound-of P* = *a* $]\!]$ $\Longrightarrow$ *bounded-by a P*
by (*blast*)

lemma *bounded-const*[*simp*]:
bounded (λ*x*. *c*)
by (*blast*)

lemma *bounded-by-const*[*intro*]:
c $\leq$ *b* $\Longrightarrow$ *bounded-by b* (λ*x*. *c*)
by (*blast*)

lemma *bounded-by-mono-alt*[*intro*]:
$[\![$ *bounded-by b Q*; *P* $\leq$ *Q* $]\!]$ $\Longrightarrow$ *bounded-by b P*
by (*blast intro*:*order-trans dest*:*le-funD*)

lemma *bound-of-const*[*simp*, *intro*]:
bound-of (λ*x*. *c*) = (*c*::*real*)
unfolding *bound-of-def*
by(*intro antisym cSup-least cSup-upper bounded-bdd-above bounded-const*, *auto*)

lemma *bound-of-leI*:

assumes $\bigwedge x.\ P\ x \le (c{::}real)$
shows *bound-of* $P \le c$
unfolding *bound-of-def*
using *assms* **by**(*intro cSup-least*, *auto*)

lemma *bound-of-mono*[*intro*]:
$[\![\ P \le Q;$ *bounded* P; *bounded* $Q\ ]\!] \Longrightarrow$ *bound-of* $P \le$ *bound-of* Q
by (*blast intro*:*order-trans dest*:*le-funD*)

lemma *bounded-by-o*[*intro*,*simp*]:
$\bigwedge b.$ *bounded-by* $b\ P \Longrightarrow$ *bounded-by* $b\ (P\ o\ f)$
unfolding *o-def* **by**(*blast*)

lemma *le-bound-of*[*intro*]:
$\bigwedge x.$ *bounded* $f \Longrightarrow f\ x \le$ *bound-of* f
by(*blast*)

3.1.2 Non-Negative Functions.

The definitions for non-negative functions are analogous to those for bounded functions.

definition
nneg $::\ ('a \Rightarrow 'b{::}\{zero,order\}) \Rightarrow bool$
where
nneg $P \longleftrightarrow (\forall\, x.\ 0 \le P\ x)$

lemma *nnegI*[*intro*]:
$[\![\ \bigwedge x.\ 0 \le P\ x\ ]\!] \Longrightarrow$ *nneg* P
by (*simp add*:*nneg-def*)

lemma *nnegI2*[*intro*]:
$(\lambda s.\ 0) \le P \Longrightarrow$ *nneg* P
by (*blast dest*:*le-funD*)

lemma *nnegD*[*dest*]:
nneg $P \Longrightarrow 0 \le P\ x$
by (*simp add*:*nneg-def*)

lemma *nnegD2*[*dest*]:
nneg $P \Longrightarrow (\lambda s.\ 0) \le P$
by (*blast intro*:*le-funI*)

lemma *nneg-bdd-below*[*intro*]:
nneg $P \Longrightarrow$ *bdd-below* (*range* P)
by(*auto*)

lemma *nneg-const*[*iff*]:
nneg $(\lambda x.\ c) \longleftrightarrow 0 \le c$
by (*simp add*:*nneg-def*)

lemma *nneg-o*[*intro*,*simp*]:
nneg P $\Longrightarrow$ *nneg* (*P o f*)
by (*force*)

lemma *nneg-bound-nneg*[*intro*]:
$[\![$ *bounded P*; *nneg P* $]\!] \Longrightarrow 0 \leq$ *bound-of P*
by (*blast intro*:*order-trans*)

lemma *nneg-bounded-by-nneg*[*dest*]:
$[\![$ *bounded-by b P*; *nneg P* $]\!] \Longrightarrow 0 \leq (b{::}real)$
by (*blast intro*:*order-trans*)

lemma *bounded-by-nneg*[*dest*]:
fixes $P{::}'s \Rightarrow real$
shows $[\![$ *bounded-by b P*; *nneg P* $]\!] \Longrightarrow 0 \leq b$
by (*blast intro*:*order-trans*)

3.1.3 Sound Expectations

definition *sound* :: $('s \Rightarrow real) \Rightarrow bool$
where *sound P* $\equiv$ *bounded P* $\wedge$ *nneg P*

Combining *nneg* and *Expectations.bounded*, we have *sound* expectations. We set up the classical reasoner and the simplifier, such that showing soundess, or deriving a simple consequence (e.g. *sound P* $\Longrightarrow 0 \leq P\ s$) will usually follow by blast, force or simp.

lemma *soundI*:
$[\![$ *bounded P*; *nneg P* $]\!] \Longrightarrow$ *sound P*
by (*simp add*:*sound-def*)

lemma *soundI2*[*intro*]:
$[\![$ *bounded-by b P*; *nneg P* $]\!] \Longrightarrow$ *sound P*
by(*blast intro*:*soundI*)

lemma *sound-bounded*[*dest*]:
sound P $\Longrightarrow$ *bounded P*
by (*simp add*:*sound-def*)

lemma *sound-nneg*[*dest*]:
sound P $\Longrightarrow$ *nneg P*
by (*simp add*:*sound-def*)

lemma *bound-of-sound*[*intro*]:
assumes *sP*: *sound P*
shows $0 \leq$ *bound-of P*
using *assms* **by**(*auto*)

This proof demonstrates the use of the classical reasoner (specifically blast), to both introduce and eliminate soundness terms.

```
lemma sound-sum[simp,intro]:
  assumes sP: sound P and sQ: sound Q
  shows sound (λs. P s + Q s)
proof
  from sP have ⋀s. P s ≤ bound-of P by(blast)
  moreover from sQ have ⋀s. Q s ≤ bound-of Q by(blast)
  ultimately have ⋀s. P s + Q s ≤ bound-of P + bound-of Q
    by(rule add-mono)
  thus bounded-by (bound-of P + bound-of Q) (λs. P s + Q s)
    by(blast)

  from sP have ⋀s. 0 ≤ P s by(blast)
  moreover from sQ have ⋀s. 0 ≤ Q s by(blast)
  ultimately have ⋀s. 0 ≤ P s + Q s by(simp add:add-mono)
  thus nneg (λs. P s + Q s) by(blast)
qed

lemma mult-sound:
  assumes sP: sound P and sQ: sound Q
  shows sound (λs. P s ∗ Q s)
proof
  from sP have ⋀s. P s ≤ bound-of P by(blast)
  moreover from sQ have ⋀s. Q s ≤ bound-of Q by(blast)
  ultimately have ⋀s. P s ∗ Q s ≤ bound-of P ∗ bound-of Q
    using sP and sQ by(blast intro:mult-mono)
  thus bounded-by (bound-of P ∗ bound-of Q) (λs. P s ∗ Q s) by(blast)

  from sP and sQ show nneg (λs. P s ∗ Q s)
    by(blast intro:mult-nonneg-nonneg)
qed

lemma div-sound:
  assumes sP: sound P and cpos: 0 < c
  shows sound (λs. P s / c)
proof
  from sP and cpos have ⋀s. P s / c ≤ bound-of P / c
    by(blast intro:divide-right-mono less-imp-le)
  thus bounded-by (bound-of P / c) (λs. P s / c) by(blast)
  from assms show nneg (λs. P s / c)
    by(blast intro:divide-nonneg-pos)
qed

lemma tminus-sound:
  assumes sP: sound P and nnc: 0 ≤ c
  shows sound (λs. P s ⊖ c)
proof(rule soundI)
  from sP have ⋀s. P s ≤ bound-of P by(blast)
  with nnc have ⋀s. P s ⊖ c ≤ bound-of P ⊖ c
```

```
    by(blast intro:tminus-left-mono)
  thus bounded (λs. P s ⊖ c) by(blast)
  show nneg (λs. P s ⊖ c) by(blast)
qed

lemma const-sound:
  0 ≤ c ⟹ sound (λs. c)
  by (blast)

lemma sound-o[intro,simp]:
  sound P ⟹ sound (P o f)
  unfolding o-def by(blast)

lemma sc-bounded-by[intro,simp]:
  ⟦ sound P; 0 ≤ c ⟧ ⟹ bounded-by (c ∗ bound-of P) (λx. c ∗ P x)
  by(blast intro!:mult-left-mono)

lemma sc-bounded[intro,simp]:
  assumes sP:  sound P and pos: 0 ≤ c
  shows bounded (λx. c ∗ P x)
  using assms by(blast)

lemma sc-bound[simp]:
  assumes sP: sound P
    and cnn: 0 ≤ c
  shows c ∗ bound-of P = bound-of (λx. c ∗ P x)
proof(cases c = 0)
  case True then show ?thesis by(simp)
next
  case False with cnn have cpos: 0 < c by(auto)
  show ?thesis
  proof (rule antisym)
    from sP and cnn have bounded (λx. c ∗ P x) by(simp)
    hence ⋀x. c ∗ P x ≤ bound-of (λx. c ∗ P x)
      by(rule le-bound-of)
    with cpos have ⋀x. P x ≤ inverse c ∗ bound-of (λx. c ∗ P x)
      by(force intro:mult-div-mono-right)
    hence bound-of P ≤ inverse c ∗ bound-of (λx. c ∗ P x)
      by(blast)
    with cpos show c ∗ bound-of P ≤ bound-of (λx. c ∗ P x)
      by(force intro:mult-div-mono-left)
  next
    from sP and cpos have ⋀x. c ∗ P x ≤ c ∗ bound-of P
      by(blast intro:mult-left-mono less-imp-le)
    thus bound-of (λx. c ∗ P x) ≤ c ∗ bound-of P
      by(blast)
  qed
qed
```

lemma *sc-sound*:
  ⟦ *sound P*; *0* ≤ *c* ⟧ ⟹ *sound* (λ*s*. *c* ∗ *P s*)
  by (*blast intro*:*mult-nonneg-nonneg*)

lemma *bounded-by-mult*:
  assumes *sP*: *sound P* **and** *bP*: *bounded-by a P*
    and *sQ*: *sound Q* **and** *bQ*: *bounded-by b Q*
  shows *bounded-by* (*a* ∗ *b*) (λ*s*. *P s* ∗ *Q s*)
  using *assms* **by**(*intro bounded-byI*, *auto intro*:*mult-mono*)

lemma *bounded-by-add*:
  fixes *P*::′*s* ⇒ *real* **and** *Q*
  assumes *bP*: *bounded-by a P*
    and *bQ*: *bounded-by b Q*
  shows *bounded-by* (*a* + *b*) (λ*s*. *P s* + *Q s*)
  using *assms* **by**(*intro bounded-byI*, *auto intro*:*add-mono*)

lemma *sound-unit*[*intro*!,*simp*]:
  sound (λ*s*. *1*)
  by(*auto*)

lemma *unit-mult*[*intro*]:
  assumes *sP*: *sound P* **and** *bP*: *bounded-by 1 P*
    and *sQ*: *sound Q* **and** *bQ*: *bounded-by 1 Q*
  shows *bounded-by 1* (λ*s*. *P s* ∗ *Q s*)
proof(*rule bounded-byI*)
  fix *s*
  have *P s* ∗ *Q s* ≤ *1* ∗ *1*
    using *assms* **by**(*blast dest*:*bounded-by-mult*)
  thus *P s* ∗ *Q s* ≤ *1* **by**(*simp*)
qed

lemma *sum-sound*:
  assumes *sP*: ∀ *x*∈*S*. *sound* (*P x*)
  shows *sound* (λ*s*. ∑ *x*∈*S*. *P x s*)
proof(*rule soundI2*)
  from *sP* **show** *bounded-by* (∑ *x*∈*S*. *bound-of* (*P x*)) (λ*s*. ∑ *x*∈*S*. *P x s*)
    by(*auto intro*!:*sum-mono*)
  from *sP* **show** *nneg* (λ*s*. ∑ *x*∈*S*. *P x s*)
    by(*auto intro*!:*sum-nonneg*)
qed

3.1.4 Unitary expectations

A unitary expectation is a sound expectation that is additionally bounded by one. This is the domain on which the *liberal* (partial correctness) semantics operates.

definition *unitary* :: ′*s expect* ⇒ *bool*
where *unitary P* ⟷ *sound P* ∧ *bounded-by 1 P*

lemma *unitaryI*[*intro*]:
 ⟦ *sound P*; *bounded-by 1 P* ⟧ $\Longrightarrow$ *unitary P*
 by(*simp add*:*unitary-def*)

lemma *unitaryI2*:
 ⟦ *nneg P*; *bounded-by 1 P* ⟧ $\Longrightarrow$ *unitary P*
 by(*auto*)

lemma *unitary-sound*[*dest*]:
 unitary P $\Longrightarrow$ *sound P*
 by(*simp add*:*unitary-def*)

lemma *unitary-bound*[*dest*]:
 unitary P $\Longrightarrow$ *bounded-by 1 P*
 by(*simp add*:*unitary-def*)

3.1.5 Standard Expectations

definition
 embed-bool :: ($'s \Rightarrow bool$) $\Rightarrow$ $'s \Rightarrow real$ (‹« - »› *1000*)
where
 «*P*» $\equiv$ (λs. *if P s then 1 else 0*)

Standard expectations are the embeddings of boolean predicates, mapping *False* to 0 and *True* to 1. We write « *P* » rather than [*P*] (the syntax employed by McIver and Morgan [2004]) for boolean embedding to avoid clashing with the HOL syntax for lists.

lemma *embed-bool-nneg*[*simp*,*intro*]:
 nneg «*P*»
 unfolding *embed-bool-def* **by**(*force*)

lemma *embed-bool-bounded-by-1*[*simp*,*intro*]:
 bounded-by 1 «*P*»
 unfolding *embed-bool-def* **by**(*force*)

lemma *embed-bool-bounded*[*simp*,*intro*]:
 bounded «*P*»
 by (*blast*)

Standard expectations have a number of convenient properties, which mostly follow from boolean algebra.

lemma *embed-bool-idem*:
 «*P*» *s* $*$ «*P*» *s* = «*P*» *s*
 by (*simp add*:*embed-bool-def*)

lemma *eval-embed-true*[*simp*]:
 P s $\Longrightarrow$ «*P*» *s* = *1*
 by (*simp add*:*embed-bool-def*)

lemma *eval-embed-false*[*simp*]:
$\neg P\ s \Longrightarrow$ «*P*» $s = 0$
by (*simp add*:*embed-bool-def*)

lemma *embed-ge-0*[*simp*,*intro*]:
$0 \leq$ «*G*» *s*
by (*simp add*:*embed-bool-def*)

lemma *embed-le-1*[*simp*,*intro*]:
«*G*» $s \leq 1$
by(*simp add*:*embed-bool-def*)

lemma *embed-le-1-alt*[*simp*,*intro*]:
$0 \leq 1 -$ «*G*» *s*
by(*subst add-le-cancel-right*[**where** *c*=«*G*» *s*, *symmetric*], *simp*)

lemma *expect-1-I*:
$P\ x \Longrightarrow 1 \leq$ «*P*» *x*
by(*simp*)

lemma *standard-sound*[*intro*,*simp*]:
sound «*P*»
by(*blast*)

lemma *embed-o*[*simp*]:
«*P*» *o f* = «*P o f*»
unfolding *embed-bool-def o-def* **by**(*simp*)

Negating a predicate has the expected effect in its embedding as an expectation:

definition *negate* :: $('s \Rightarrow bool) \Rightarrow 's \Rightarrow bool$ (‹$\mathcal{N}$›)
where *negate* $P = (\lambda s.\ \neg\ P\ s)$

lemma *negateI*:
$\neg\ P\ s \Longrightarrow \mathcal{N}\ P\ s$
by (*simp add*:*negate-def*)

lemma *embed-split*:
$f\ s =$ «*P*» $s * f\ s +$ «$\mathcal{N}$ *P*» $s * f\ s$
by (*simp add*:*negate-def embed-bool-def*)

lemma *negate-embed*:
«$\mathcal{N}$ *P*» $s = 1 -$ «*P*» *s*
by (*simp add*:*embed-bool-def negate-def*)

lemma *eval-nembed-true*[*simp*]:
$P\ s \Longrightarrow$ «$\mathcal{N}$ *P*» $s = 0$
by (*simp add*:*embed-bool-def negate-def*)

lemma *eval-nembed-false*[*simp*]:

$\neg P\ s \Longrightarrow$ «$\mathcal{N}$ P» $s = 1$
by (*simp add*:*embed-bool-def negate-def*)

lemma *negate-Not*[*simp*]:
$\mathcal{N}$ *Not* $= (\lambda x.\ x)$
by(*simp add*:*negate-def*)

lemma *negate-negate*[*simp*]:
$\mathcal{N}\ (\mathcal{N}\ P) = P$
by(*simp add*:*negate-def*)

lemma *embed-bool-cancel*:
«G» $s *$ «$\mathcal{N}$ G» $s = 0$
by(*cases G s*, *simp-all*)

3.1.6 Entailment

Entailment on expectations is a generalisation of that on predicates, and is defined by pointwise comparison:

abbreviation *entails* :: $('s \Rightarrow real) \Rightarrow ('s \Rightarrow real) \Rightarrow bool$ (‹- ⊩ -› *50*)
where $P \Vdash Q \equiv P \leq Q$

lemma *entailsI*[*intro*]:
$[\![\bigwedge s.\ P\ s \leq Q\ s]\!] \Longrightarrow P \Vdash Q$
by(*simp add*:*le-funI*)

lemma *entailsD*[*dest*]:
$P \Vdash Q \Longrightarrow P\ s \leq Q\ s$
by(*simp add*:*le-funD*)

lemma *eq-entails*[*intro*]:
$P = Q \Longrightarrow P \Vdash Q$
by(*blast*)

lemma *entails-trans*[*trans*]:
$[\![\ P \Vdash Q;\ Q \Vdash R\ ]\!] \Longrightarrow P \Vdash R$
by(*blast intro*:*order-trans*)

For standard expectations, both notions of entailment coincide. This result justifies the above claim that our definition generalises predicate entailment:

lemma *implies-entails*:
$[\![\ \bigwedge s.\ P\ s \Longrightarrow Q\ s\ ]\!] \Longrightarrow$ «P» ⊩ «Q»
by(*rule entailsI*, *case-tac P s*, *simp-all*)

lemma *entails-implies*:
$\bigwedge s.$ ⟦ «P» ⊩ «Q»; $P\ s$ ⟧ $\Longrightarrow Q\ s$
by(*rule ccontr*, *drule-tac s=s* **in** *entailsD*, *simp*)

3.1.7 Expectation Conjunction

definition
pconj :: *real* ⇒ *real* ⇒ *real* (**infixl** ‹.&› *71*)
where
p .& $q \equiv p + q \ominus 1$

definition
exp-conj :: (*'s* ⇒ *real*) ⇒ (*'s* ⇒ *real*) ⇒ (*'s* ⇒ *real*) (**infixl** ‹&&› *71*)
where *a* && *b* ≡ λ*s*. (*a s* .& *b s*)

Expectation conjunction likewise generalises (boolean) predicate conjunction. We show that the expected properties are preserved, and instantiate both the classical reasoner, and the simplifier (in the case of associativity and commutativity).

lemma *pconj-lzero*[*intro*,*simp*]:
$b \leq 1 \Longrightarrow 0$.& $b = 0$
by(*simp add*:*pconj-def tminus-def*)

lemma *pconj-rzero*[*intro*,*simp*]:
$b \leq 1 \Longrightarrow b$.& $0 = 0$
by(*simp add*:*pconj-def tminus-def*)

lemma *pconj-lone*[*intro*,*simp*]:
$0 \leq b \Longrightarrow 1$.& $b = b$
by(*simp add*:*pconj-def tminus-def*)

lemma *pconj-rone*[*intro*,*simp*]:
$0 \leq b \Longrightarrow b$.& $1 = b$
by(*simp add*:*pconj-def tminus-def*)

lemma *pconj-bconj*:
«*a*» *s* .& «*b*» *s* = «λ*s*. *a s* ∧ *b s*» *s*
unfolding *embed-bool-def pconj-def tminus-def* **by**(*force*)

lemma *pconj-comm*[*ac-simps*]:
a .& *b* = *b* .& *a*
by(*simp add*:*pconj-def ac-simps*)

lemma *pconj-assoc*:
$[\![\, 0 \leq a; a \leq 1; 0 \leq b; b \leq 1; 0 \leq c; c \leq 1 \,]\!] \Longrightarrow$
a .& (*b* .& *c*) = (*a* .& *b*) .& *c*
unfolding *pconj-def tminus-def* **by**(*simp*)

lemma *pconj-mono*:
$[\![\, a \leq b; c \leq d \,]\!] \Longrightarrow a$.& $c \leq b$.& d
unfolding *pconj-def tminus-def* **by**(*simp*)

lemma *pconj-nneg*[*intro*,*simp*]:
$0 \leq a$.& b

unfolding *pconj-def tminus-def* **by**(*auto*)

lemma *min-pconj*:
(*min a b*) .& (*min c d*) $\leq$ *min* (*a* .& *c*) (*b* .& *d*)
by(*cases a* $\leq$ *b*,
(*cases c* $\leq$ *d*,
simp-all add:*min*.*absorb1 min*.*absorb2 pconj-mono*)[],
(*cases c* $\leq$ *d*,
simp-all add:*min*.*absorb1 min*.*absorb2 pconj-mono*))

lemma *pconj-less-one*[*simp*]:
$a + b < 1 \Longrightarrow a$.& $b = 0$
unfolding *pconj-def* **by**(*simp*)

lemma *pconj-ge-one*[*simp*]:
$1 \leq a + b \Longrightarrow a$.& $b = a + b - 1$
unfolding *pconj-def* **by**(*simp*)

lemma *pconj-idem*[*simp*]:
«*P*» *s* .& «*P*» *s* = «*P*» *s*
unfolding *pconj-def* **by**(*cases P s*, *simp-all*)

3.1.8 Rules Involving Conjunction.

lemma *exp-conj-mono-left*:
$P \Vdash Q \Longrightarrow P$ && $R \Vdash Q$ && R
unfolding *exp-conj-def pconj-def*
by(*auto intro*:*tminus-left-mono add-right-mono*)

lemma *exp-conj-mono-right*:
$Q \Vdash R \Longrightarrow P$ && $Q \Vdash P$ && R
unfolding *exp-conj-def pconj-def*
by(*auto intro*:*tminus-left-mono add-left-mono*)

lemma *exp-conj-comm*[*ac-simps*]:
a && *b* = *b* && *a*
by(*simp add*:*exp-conj-def ac-simps*)

lemma *exp-conj-bounded-by*[*intro*,*simp*]:
assumes *bP*: *bounded-by 1 P*
and *bQ*: *bounded-by 1 Q*
shows *bounded-by 1* (*P* && *Q*)
proof(*rule bounded-byI*, *unfold exp-conj-def pconj-def*)
fix *x*
from *bP* **have** $P\ x \leq 1$ **by**(*blast*)
moreover from *bQ* **have** $Q\ x \leq 1$ **by**(*blast*)
ultimately have $P\ x + Q\ x \leq 2$ **by**(*auto*)
thus $P\ x + Q\ x \ominus 1 \leq 1$
unfolding *tminus-def* **by**(*simp*)

```
qed

lemma exp-conj-o-distrib[simp]:
  (P && Q) o f = (P o f) && (Q o f)
  unfolding exp-conj-def o-def by(simp)

lemma exp-conj-assoc:
  assumes unitary P and unitary Q and unitary R
  shows P && (Q && R) = (P && Q) && R
  unfolding exp-conj-def
proof(rule ext)
  fix s
  from assms have 0 ≤ P s by(blast)
  moreover from assms have 0 ≤ Q s by(blast)
  moreover from assms have 0 ≤ R s by(blast)
  moreover from assms have P s ≤ 1 by(blast)
  moreover from assms have Q s ≤ 1 by(blast)
  moreover from assms have R s ≤ 1 by(blast)
  ultimately
  show P s .& (Q s .& R s) = (P s .& Q s) .& R s
    by(simp add:pconj-assoc)
qed

lemma exp-conj-top-left[simp]:
  sound P ⟹ «λ-. True» && P = P
  unfolding exp-conj-def by(force)

lemma exp-conj-top-right[simp]:
  sound P ⟹ P && «λ-. True» = P
  unfolding exp-conj-def by(force)

lemma exp-conj-idem[simp]:
  «P» && «P» = «P»
  unfolding exp-conj-def
  by(rule ext, cases P s, simp-all)

lemma exp-conj-nneg[intro,simp]:
  (λs. 0) ≤ P && Q
  unfolding exp-conj-def
  by(blast intro:le-funI)

lemma exp-conj-sound[intro,simp]:
  assumes s-P: sound P
    and s-Q: sound Q
  shows sound (P && Q)
  unfolding exp-conj-def
proof(rule soundI)
  from s-P and s-Q have ⋀s. 0 ≤ P s + Q s by(blast intro:add-nonneg-nonneg)
  hence ⋀s. P s .& Q s ≤ P s + Q s
```

```
    unfolding pconj-def by(force intro:tminus-less)
  also from assms have ⋀s. ... s ≤ bound-of P + bound-of Q
    by(blast intro:add-mono)
  finally have bounded-by (bound-of P + bound-of Q) (λs. P s .& Q s)
    by(blast)
  thus bounded (λs. P s .& Q s) by(blast)

  show nneg (λs. P s .& Q s)
    unfolding pconj-def tminus-def by(force)
qed

lemma exp-conj-rzero[simp]:
  bounded-by 1 P ⟹ P && (λs. 0) = (λs. 0)
  unfolding exp-conj-def by(force)

lemma exp-conj-1-right[simp]:
  assumes nn: nneg A
  shows A && (λ-. 1) = A
  unfolding exp-conj-def pconj-def tminus-def
proof(rule ext, simp)
  fix s
  from nn have 0 ≤ A s by(blast)
  thus max (A s) 0 = A s by(force)
qed

lemma exp-conj-std-split:
  «λs. P s ∧ Q s» = «P» && «Q»
  unfolding exp-conj-def embed-bool-def pconj-def
  by(auto)
```

3.1.9 Rules Involving Entailment and Conjunction Together

Meta-conjunction distributes over expectaton entailment, becoming expectation conjunction:

```
lemma entails-frame:
  assumes ePR: P ⊩ R
    and eQS: Q ⊩ S
  shows P && Q ⊩ R && S
proof(rule le-funI)
  fix s
  from ePR have P s ≤ R s by(blast)
  moreover from eQS have Q s ≤ S s by(blast)
  ultimately have P s + Q s ≤ R s + S s by(rule add-mono)
  hence P s + Q s ⊖ 1 ≤ R s + S s ⊖ 1 by(rule tminus-left-mono)
  thus (P && Q) s ≤ (R && S) s
    unfolding exp-conj-def pconj-def .
qed
```

This rule allows something very much akin to a case distinction on the pre-expectation.

```
lemma pentails-cases:
  assumes PQe: ⋀x. P x ⊩ Q x
    and exhaust: ⋀s. ∃x. P (x s) s = 1
    and framed: ⋀x. P x && R ⊩ Q x && S
    and sR: sound R and sS: sound S
    and bQ: ⋀x. bounded-by 1 (Q x)
  shows R ⊩ S
proof(rule le-funI)
  fix s
  from exhaust obtain x where P-xs: P x s = 1 by(blast)
  moreover {
    hence 1 = P x s by(simp)
    also from PQe have P x s ≤ Q x s by(blast dest:le-funD)
    finally have Q x s = 1
      using bQ by(blast intro:antisym)
  }
  moreover note le-funD[OF framed[where x=x], where x=s]
  moreover from sR have 0 ≤ R s by(blast)
  moreover from sS have 0 ≤ S s by(blast)
  ultimately show R s ≤ S s by(simp add:exp-conj-def)
qed

lemma unitary-bot[iff]:
  unitary (λs. 0::real)
  by(auto)

lemma unitary-top[iff]:
  unitary (λs. 1::real)
  by(auto)

lemma unitary-embed[iff]:
  unitary «P»
  by(auto)

lemma unitary-const[iff]:
  ⟦ 0 ≤ c; c ≤ 1 ⟧ ⟹ unitary (λs. c)
  by(auto)

lemma unitary-mult:
  assumes uA: unitary A and uB: unitary B
  shows unitary (λs. A s ∗ B s)
proof(intro unitaryI2 nnegI bounded-byI)
  fix s
  from assms have nnA: 0 ≤ A s and nnB: 0 ≤ B s by(auto)
  thus 0 ≤ A s ∗ B s by(rule mult-nonneg-nonneg)
  from assms have A s ≤ 1 and B s ≤ 1 by(auto)
  with nnB have A s ∗ B s ≤ 1 ∗ 1 by(intro mult-mono, auto)
  also have ... = 1 by(simp)
  finally show A s ∗ B s ≤ 1 .
```

qed

lemma *exp-conj-unitary*:
  ⟦ *unitary P*; *unitary Q* ⟧ ⟹ *unitary* (*P* && *Q*)
  by(*intro unitaryI2 nnegI2*, *auto*)

lemma *unitary-comp*[*simp*]:
  unitary P ⟹ *unitary* (*P o f*)
  by(*intro unitaryI2 nnegI bounded-byI*, *auto simp*:*o-def*)

lemmas *unitary-intros* =
  unitary-bot unitary-top unitary-embed unitary-mult exp-conj-unitary
  unitary-comp unitary-const

lemmas *sound-intros* =
  mult-sound div-sound const-sound sound-o sound-sum
  tminus-sound sc-sound exp-conj-sound sum-sound

end

3.2 Expectation Transformers

theory *Transformers* **imports** *Expectations* **begin** **type-synonym** *'s trans* = *'s expect* ⇒ *'s expect*

Transformers are functions from expectations to expectations i.e. (*'s* ⇒ *real*) ⇒ *'s* ⇒ *real*.

The set of *healthy* transformers is the universe into which we place our semantic interpretation of pGCL programs. In its standard presentation, the healthiness condition for pGCL programs is *sublinearity*, for demonic programs, and *superlinearity* for angelic programs. We extract a minimal core property, consisting of monotonicity, feasibility and scaling to form our healthiness property, which holds across all programs. The additional components of sublinearity are broken out separately, and shown later. The two reasons for this are firstly to avoid the effort of establishing sub-(super-)linearity globally, and to allow us to define primitives whose sublinearity, and indeed healthiness, depend on context.

Consider again the automaton of Figure 3.1. Here, the effect of executing the automaton from its initial state (a) until it reaches some final state (b or c) is to *transform* the expectation on final states (P), into one on initial states, giving the *expected* value of the function on termination. Here, the transformation is linear: $P_{\text{prior}}(a) = 0.7 * P_{\text{post}}(b) + 0.3 * P_{\text{post}}(c)$, but this need not be the case.

Consider the automaton of Figure 3.2. Here, we have extended that of Figure 3.1 with two additional states, d and e, and a pair of silent (unlabelled) transitions. From the initial state, e, this automaton is free to transition either to the original starting state (a), and thence behave exactly as the previous automaton did, or to d, which has the same set of available transitions, now with different probabilities.

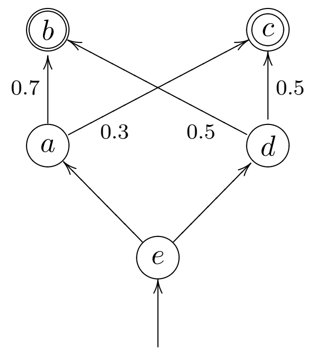

Figure 3.2: A nondeterministic-probabilistic automaton.

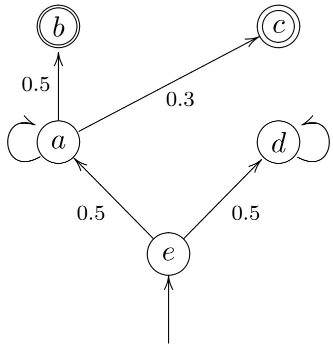

Figure 3.3: A diverging automaton.

Where previously we could state that the automaton would terminate in state b with probability 0.7 (and in c with probability 0.3), this now depends on the outcome of the *nondeterministic* transition from e to either a or d. The most we can now say is that we must reach b with probability *at least* 0.5 (the minimum from either a or d) and c with at least probability 0.3. Note that these probabilities do not sum to one (although the sum will still always be less than one). The associated expectation transformer is now *sub*-linear: $P_{\text{prior}}(e) = 0.5 * P_{\text{post}}(b) + 0.3 * P_{\text{post}}(c)$.

Finally, Figure 3.3 shows the other way in which strict sublinearity arises: divergence. This automaton transitions with probability 0.5 to state d, from which it never escapes. Once there, the probability of reaching any terminating state is zero, and thus the probabilty of terminating from the initial state (e) is no higher than 0.5. If it instead takes the edge to state a, we again see a self loop, and thus in theory an infinite trace. In this case, however, every time the automaton reaches state a, with probability $0.5 + 0.3 = 0.8$, it transitions to a terminating state. An infinite trace of transitions $a \to a \to \ldots$ thus has probability 0, and the automaton terminates with probability 1. We formalise such probabilistic termination arguments in Section 4.11.

Having reached a, the automaton will proceed to b with probability $0.5 * (1/(0.5 + 0.3)) = 0.625$, and to c with probability 0.375. As a is in turn reached half the time, the final probability of ending in b is 0.3125, and in c, 0.1875, which sum to only 0.5. The remaining probability is that the automaton diverges via d. We view nondeterminism and divergence demonically: we take the *least* probability of reaching a given final state, and use it to calculate the expectation. Thus for this automaton, $P_{\text{prior}}(e) = 0.3125 * P_{\text{post}}(b) + 0.1875 * P_{\text{post}}(c)$. The end result is the same as for nondeterminism: a sublinear transformation (the weights sum to less than one). The two outcomes are thus unified in the semantic interpretation, although as we will establish in Section 4.6, the two have slightly different algebraic properties.

This pattern holds for all pGCL programs: probabilistic choices are always linear, while struct sublinearity is introduced both nondeterminism and divergence.

Healthiness, again, is the combination of three properties: feasibility, monotonicity and scaling. Feasibility requires that a transformer take non-negative expectations to non-negative expectations, and preserve bounds. Thus, starting with an expectation bounded between 0 and some bound, b, after applying any number of feasible transformers, the result will still be bounded between 0 and b. This closure property allows us to treat expectations almost as a complete lattice. Specifically, for any b, the set of expectations bounded by b is a complete lattice ($\bot = (\lambda s.0)$, $\top = (\lambda s.b)$), and is closed under the action of feasible transformers, including $\sqcap$ and $\sqcup$, which are themselves feasible. We are thus able to define both least and greatest fixed points on this set, and thus give semantics to recursive programs built from feasible components.

3.2.1 Comparing Transformers

Transformers are compared pointwise, but only on *sound* expectations. From the preorder so generated, we define equivalence by antisymmetry, giving a partial order.

definition
  le-trans :: *'s trans* ⇒ *'s trans* ⇒ *bool*
where
  le-trans t u ≡ ∀ *P*. *sound P* ⟶ *t P* ≤ *u P*

We also need to define relations restricted to *unitary* transformers, for the liberal (wlp) semantics.

definition
  le-utrans :: *'s trans* ⇒ *'s trans* ⇒ *bool*
where
  le-utrans t u ⟷ (∀ *P*. *unitary P* ⟶ *t P* ≤ *u P*)

lemma *le-transI*[*intro*]:

$\llbracket \bigwedge P.\ sound\ P \Longrightarrow t\ P \leq u\ P \rrbracket \Longrightarrow le\text{-}trans\ t\ u$
by(*simp add*:*le-trans-def*)

lemma *le-utransI*[*intro*]:
$\llbracket \bigwedge P.\ unitary\ P \Longrightarrow t\ P \leq u\ P \rrbracket \Longrightarrow le\text{-}utrans\ t\ u$
by(*simp add*:*le-utrans-def*)

lemma *le-transD*[*dest*]:
$\llbracket le\text{-}trans\ t\ u;\ sound\ P \rrbracket \Longrightarrow t\ P \leq u\ P$
by(*simp add*:*le-trans-def*)

lemma *le-utransD*[*dest*]:
$\llbracket le\text{-}utrans\ t\ u;\ unitary\ P \rrbracket \Longrightarrow t\ P \leq u\ P$
by(*simp add*:*le-utrans-def*)

lemma *le-trans-trans*[*trans*]:
$\llbracket le\text{-}trans\ x\ y;\ le\text{-}trans\ y\ z \rrbracket \Longrightarrow le\text{-}trans\ x\ z$
unfolding *le-trans-def* **by**(*blast dest*:*order-trans*)

lemma *le-utrans-trans*[*trans*]:
$\llbracket le\text{-}utrans\ x\ y;\ le\text{-}utrans\ y\ z \rrbracket \Longrightarrow le\text{-}utrans\ x\ z$
unfolding *le-utrans-def* **by**(*blast dest*:*order-trans*)

lemma *le-trans-refl*[*iff*]:
le-trans x x
by(*simp add*:*le-trans-def*)

lemma *le-utrans-refl*[*iff*]:
le-utrans x x
by(*simp add*:*le-utrans-def*)

lemma *le-trans-le-utrans*[*dest*]:
$le\text{-}trans\ t\ u \Longrightarrow le\text{-}utrans\ t\ u$
unfolding *le-trans-def le-utrans-def* **by**(*auto*)

definition
$l\text{-}trans :: {}'s\ trans \Rightarrow {}'s\ trans \Rightarrow bool$
where
$l\text{-}trans\ t\ u \longleftrightarrow le\text{-}trans\ t\ u \land \neg\ le\text{-}trans\ u\ t$

Transformer equivalence is induced by comparison:

definition
$equiv\text{-}trans :: {}'s\ trans \Rightarrow {}'s\ trans \Rightarrow bool$
where
$equiv\text{-}trans\ t\ u \longleftrightarrow le\text{-}trans\ t\ u \land le\text{-}trans\ u\ t$

definition
$equiv\text{-}utrans :: {}'s\ trans \Rightarrow {}'s\ trans \Rightarrow bool$
where

equiv-utrans t u $\longleftrightarrow$ *le-utrans t u* $\wedge$ *le-utrans u t*

lemma *equiv-transI*[*intro*]:
$[\![\bigwedge P.$ *sound P* $\Longrightarrow t\ P = u\ P]\!] \Longrightarrow$ *equiv-trans t u*
unfolding *equiv-trans-def* **by**(*force*)

lemma *equiv-utransI*[*intro*]:
$[\![\bigwedge P.$ *sound P* $\Longrightarrow t\ P = u\ P]\!] \Longrightarrow$ *equiv-utrans t u*
unfolding *equiv-utrans-def* **by**(*force*)

lemma *equiv-transD*[*dest*]:
$[\![$ *equiv-trans t u*; *sound P* $]\!] \Longrightarrow t\ P = u\ P$
unfolding *equiv-trans-def* **by**(*blast intro*:*antisym*)

lemma *equiv-utransD*[*dest*]:
$[\![$ *equiv-utrans t u*; *unitary P* $]\!] \Longrightarrow t\ P = u\ P$
unfolding *equiv-utrans-def* **by**(*blast intro*:*antisym*)

lemma *equiv-trans-refl*[*iff*]:
equiv-trans t t
by(*blast*)

lemma *equiv-utrans-refl*[*iff*]:
equiv-utrans t t
by(*blast*)

lemma *le-trans-antisym*:
$[\![$ *le-trans x y*; *le-trans y x* $]\!] \Longrightarrow$ *equiv-trans x y*
unfolding *equiv-trans-def* **by**(*simp*)

lemma *le-utrans-antisym*:
$[\![$ *le-utrans x y*; *le-utrans y x* $]\!] \Longrightarrow$ *equiv-utrans x y*
unfolding *equiv-utrans-def* **by**(*simp*)

lemma *equiv-trans-comm*[*ac-simps*]:
equiv-trans t u $\longleftrightarrow$ *equiv-trans u t*
unfolding *equiv-trans-def* **by**(*blast*)

lemma *equiv-utrans-comm*[*ac-simps*]:
equiv-utrans t u $\longleftrightarrow$ *equiv-utrans u t*
unfolding *equiv-utrans-def* **by**(*blast*)

lemma *equiv-imp-le*[*intro*]:
equiv-trans t u $\Longrightarrow$ *le-trans t u*
unfolding *equiv-trans-def* **by**(*clarify*)

lemma *equivu-imp-le*[*intro*]:
equiv-utrans t u $\Longrightarrow$ *le-utrans t u*
unfolding *equiv-utrans-def* **by**(*clarify*)

lemma *equiv-imp-le-alt*:
equiv-trans t u ⟹ *le-trans u t*
by(*force simp*:*ac-simps*)

lemma *equiv-uimp-le-alt*:
equiv-utrans t u ⟹ *le-utrans u t*
by(*force simp*:*ac-simps*)

lemma *le-trans-equiv-rsp*[*simp*]:
equiv-trans t u ⟹ *le-trans t v* ⟷ *le-trans u v*
unfolding *equiv-trans-def* **by**(*blast intro*:*le-trans-trans*)

lemma *le-utrans-equiv-rsp*[*simp*]:
equiv-utrans t u ⟹ *le-utrans t v* ⟷ *le-utrans u v*
unfolding *equiv-utrans-def* **by**(*blast intro*:*le-utrans-trans*)

lemma *equiv-trans-le-trans*[*trans*]:
⟦ *equiv-trans t u*; *le-trans u v* ⟧ ⟹ *le-trans t v*
by(*simp*)

lemma *equiv-utrans-le-utrans*[*trans*]:
⟦ *equiv-utrans t u*; *le-utrans u v* ⟧ ⟹ *le-utrans t v*
by(*simp*)

lemma *le-trans-equiv-rsp-right*[*simp*]:
equiv-trans t u ⟹ *le-trans v t* ⟷ *le-trans v u*
unfolding *equiv-trans-def* **by**(*blast intro*:*le-trans-trans*)

lemma *le-utrans-equiv-rsp-right*[*simp*]:
equiv-utrans t u ⟹ *le-utrans v t* ⟷ *le-utrans v u*
unfolding *equiv-utrans-def* **by**(*blast intro*:*le-utrans-trans*)

lemma *le-trans-equiv-trans*[*trans*]:
⟦ *le-trans t u*; *equiv-trans u v* ⟧ ⟹ *le-trans t v*
by(*simp*)

lemma *le-utrans-equiv-utrans*[*trans*]:
⟦ *le-utrans t u*; *equiv-utrans u v* ⟧ ⟹ *le-utrans t v*
by(*simp*)

lemma *equiv-trans-trans*[*trans*]:
assumes *xy*: *equiv-trans x y*
and *yz*: *equiv-trans y z*
shows *equiv-trans x z*
proof(*rule le-trans-antisym*)
from *xy* **have** *le-trans x y* **by**(*blast*)
also from *yz* **have** *le-trans y z* **by**(*blast*)
finally show *le-trans x z* **.**

from *yz* **have** *le-trans z y* **by**(*force simp*:*ac-simps*)
also from *xy* **have** *le-trans y x* **by**(*force simp*:*ac-simps*)
finally show *le-trans z x* **.**
qed

lemma *equiv-utrans-trans*[*trans*]:
assumes *xy*: *equiv-utrans x y*
and *yz*: *equiv-utrans y z*
shows *equiv-utrans x z*
proof(*rule le-utrans-antisym*)
from *xy* **have** *le-utrans x y* **by**(*blast*)
also from *yz* **have** *le-utrans y z* **by**(*blast*)
finally show *le-utrans x z* **.**
from *yz* **have** *le-utrans z y* **by**(*force simp*:*ac-simps*)
also from *xy* **have** *le-utrans y x* **by**(*force simp*:*ac-simps*)
finally show *le-utrans z x* **.**
qed

lemma *equiv-trans-equiv-utrans*[*dest*]:
equiv-trans t u $\Longrightarrow$ *equiv-utrans t u*
by(*auto*)

3.2.2 Healthy Transformers

Feasibility

definition *feasible* :: $(('a \Rightarrow real) \Rightarrow ('a \Rightarrow real)) \Rightarrow bool$
where *feasible t* $\longleftrightarrow$ ($\forall$ *P b*. *bounded-by b P* $\wedge$ *nneg P* $\longrightarrow$
bounded-by b (*t P*) $\wedge$ *nneg* (*t P*))

A *feasible* transformer preserves non-negativity, and bounds. A *feasible* transformer always takes its argument 'closer to 0' (or leaves it where it is). Note that any particular value of the expectation may increase, but no element of the new expectation may exceed any bound on the old. This is thus a relatively weak condition.

lemma *feasibleI*[*intro*]:
$[\![$ $\bigwedge$*b P*. $[\![$ *bounded-by b P*; *nneg P* $]\!]$ $\Longrightarrow$ *bounded-by b* (*t P*);
$\bigwedge$*b P*. $[\![$ *bounded-by b P*; *nneg P* $]\!]$ $\Longrightarrow$ *nneg* (*t P*) $]\!]$ $\Longrightarrow$ *feasible t*
by(*force simp*:*feasible-def*)

lemma *feasible-boundedD*[*dest*]:
$[\![$ *feasible t*; *bounded-by b P*; *nneg P* $]\!]$ $\Longrightarrow$ *bounded-by b* (*t P*)
by(*simp add*:*feasible-def*)

lemma *feasible-nnegD*[*dest*]:
$[\![$ *feasible t*; *bounded-by b P*; *nneg P* $]\!]$ $\Longrightarrow$ *nneg* (*t P*)
by(*simp add*:*feasible-def*)

lemma *feasible-sound*[*dest*]:

```
  ⟦ feasible t; sound P ⟧ ⟹ sound (t P)
  by(rule soundI, unfold sound-def, (blast)+)

lemma feasible-pr-0[simp]:
  fixes t::('s ⇒ real) ⇒ 's ⇒ real
  assumes ft: feasible t
  shows t (λx. 0) = (λx. 0)
proof(rule ext, rule antisym)
  fix s

  have bounded-by 0 (λ-::'s. 0::real) by(blast)
  with ft have bounded-by 0 (t (λ-. 0)) by(blast)
  thus t (λ-. 0) s ≤ 0 by(blast)

  have nneg (λ-::'s. 0::real) by(blast)
  with ft have nneg (t (λ-. 0)) by(blast)
  thus 0 ≤ t (λ-. 0) s by(blast)
qed

lemma feasible-id:
  feasible (λx. x)
  unfolding feasible-def by(blast)

lemma feasible-bounded-by[dest]:
  ⟦ feasible t; sound P; bounded-by b P ⟧ ⟹ bounded-by b (t P)
  by(auto)

lemma feasible-fixes-top:
  feasible t ⟹ t (λs. 1) ≤ (λs. (1::real))
  by(drule bounded-byD2[OF feasible-bounded-by], auto)

lemma feasible-fixes-bot:
  assumes ft: feasible t
  shows t (λs. 0) = (λs. 0)
proof(rule antisym)
  have sb: sound (λs. 0) by(auto)
  with ft show (λs. 0) ≤ t (λs. 0) by(auto)
  thm bound-of-const
  from sb have bounded-by (bound-of (λs. 0::real)) (λs. 0) by(auto)
  hence bounded-by 0 (λs. 0::real) by(simp add:bound-of-const)
  with ft have bounded-by 0 (t (λs. 0)) by(auto)
  thus t (λs. 0) ≤ (λs. 0) by(auto)
qed

lemma feasible-unitaryD[dest]:
  assumes ft: feasible t and uP: unitary P
  shows unitary (t P)
proof(rule unitaryI)
  from uP have sound P by(auto)
```

with *ft* **show** *sound* $(t\ P)$ **by**$(auto)$
from *assms* **show** *bounded-by 1* $(t\ P)$ **by**$(auto)$
qed

Monotonicity

definition
$mono\text{-}trans :: (('s \Rightarrow real) \Rightarrow ('s \Rightarrow real)) \Rightarrow bool$
where
$mono\text{-}trans\ t \equiv \forall P\ Q.\ (sound\ P \wedge sound\ Q \wedge P \leq Q) \longrightarrow t\ P \leq t\ Q$

Monotonicity allows us to compose transformers, and thus model sequential computation. Recall the definition of predicate entailment (Section 3.1.6) as less-than-or-equal. The statement $Q \Vdash t\ R$ means that Q is everywhere below $t\ R$. For standard expectations (Section 3.1.5), this simply means that Q *implies* $t\ R$, the *weakest precondition* of R under t.

Given another, monotonic, transformer u, we have that $u\ Q \Vdash u\ (t\ R)$, or that the weakest precondition of Q under u entails that of R under the composition $u \circ t$. If we additionally know that $P \Vdash u\ Q$, then by transitivity we have $P \Vdash u\ (t\ R)$. We thus derive a probabilistic form of the standard rule for sequential composition: $[\![mono\text{-}trans\ t;\ P \Vdash u\ Q;\ Q \Vdash t\ R]\!] \Longrightarrow P \Vdash u\ (t\ R)$.

lemma *mono-transI*[*intro*]:
$[\![\bigwedge P\ Q.\ [\![sound\ P;\ sound\ Q;\ P \leq Q]\!] \Longrightarrow t\ P \leq t\ Q]\!] \Longrightarrow mono\text{-}trans\ t$
by(*simp add:mono-trans-def*)

lemma *mono-transD*[*dest*]:
$[\![mono\text{-}trans\ t;\ sound\ P;\ sound\ Q;\ P \leq Q]\!] \Longrightarrow t\ P \leq t\ Q$
by(*simp add:mono-trans-def*)

Scaling

A healthy transformer commutes with scaling by a non-negative constant.

definition
$scaling :: (('s \Rightarrow real) \Rightarrow ('s \Rightarrow real)) \Rightarrow bool$
where
$scaling\ t \equiv \forall P\ c\ x.\ sound\ P \wedge 0 \leq c \longrightarrow c * t\ P\ x = t\ (\lambda x.\ c * P\ x)\ x$

The *scaling* and feasibility properties together allow us to treat transformers as a complete lattice, when operating on bounded expectations. The action of a transformer on such a bounded expectation is completely determined by its action on *unitary* expectations (those bounded by 1): $t\ P\ s = bound\text{-}of\ P * t\ (\lambda s.\ P\ s\ /\ bound\text{-}of\ P)\ s$. Feasibility in turn ensures that the lattice of unitary expectations is closed under the action of a healthy transformer. We take advantage of this fact in Section 3.3, in order to define the fixed points of healthy transformers.

lemma *scalingI*[*intro*]:
$[\![\bigwedge P\ c\ x.\ [\![sound\ P;\ 0 \leq c]\!] \Longrightarrow c * t\ P\ x = t\ (\lambda x.\ c * P\ x)\ x]\!] \Longrightarrow scaling\ t$

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

lemma scalingD[dest]:
  ⟦ scaling t; sound P; 0 ≤ c ⟧ ⟹ c ∗ t P x = t (λx. c ∗ P x) x
  by(simp add:scaling-def)

lemma right-scalingD:
  assumes st: scaling t
    and sP: sound P
    and nnc: 0 ≤ c
  shows t P s ∗ c = t (λs. P s ∗ c) s
proof −
  have t P s ∗ c = c ∗ t P s by(simp add:algebra-simps)
  also from assms have ... = t (λs. c ∗ P s) s by(rule scalingD)
  also have ... = t (λs. P s ∗ c) s by(simp add:algebra-simps)
  finally show ?thesis .
qed
```

Healthiness

Healthy transformers are feasible and monotonic, and respect scaling

```
definition
  healthy :: (('s ⇒ real) ⇒ ('s ⇒ real)) ⇒ bool
where
  healthy t ⟷ feasible t ∧ mono-trans t ∧ scaling t

lemma healthyI[intro]:
  ⟦ feasible t; mono-trans t; scaling t ⟧ ⟹ healthy t
  by(simp add:healthy-def)

lemmas healthy-parts = healthyI[OF feasibleI mono-transI scalingI]

lemma healthy-monoD[dest]:
  healthy t ⟹ mono-trans t
  by(simp add:healthy-def)

lemmas healthy-monoD2 = mono-transD[OF healthy-monoD]

lemma healthy-feasibleD[dest]:
  healthy t ⟹ feasible t
  by(simp add:healthy-def)

lemma healthy-scalingD[dest]:
  healthy t ⟹ scaling t
  by(simp add:healthy-def)

lemma healthy-bounded-byD[intro]:
  ⟦ healthy t; bounded-by b P; nneg P ⟧ ⟹ bounded-by b (t P)
  by(blast)
```

lemma *healthy-bounded-byD2*:
⟦ *healthy t*; *bounded-by b P*; *sound P* ⟧ ⟹ *bounded-by b* (*t P*)
by(*blast*)

lemma *healthy-boundedD*[*dest*,*simp*]:
⟦ *healthy t*; *sound P* ⟧ ⟹ *bounded* (*t P*)
by(*blast*)

lemma *healthy-nnegD*[*dest*,*simp*]:
⟦ *healthy t*; *sound P* ⟧ ⟹ *nneg* (*t P*)
by(*blast intro*!:*feasible-nnegD*)

lemma *healthy-nnegD2*[*dest*,*simp*]:
⟦ *healthy t*; *bounded-by b P*; *nneg P* ⟧ ⟹ *nneg* (*t P*)
by(*blast*)

lemma *healthy-sound*[*intro*]:
⟦ *healthy t*; *sound P* ⟧ ⟹ *sound* (*t P*)
by(*rule soundI*, *blast*, *blast intro*:*feasible-nnegD*)

lemma *healthy-unitary*[*intro*]:
⟦ *healthy t*; *unitary P* ⟧ ⟹ *unitary* (*t P*)
by(*blast intro*!:*unitaryI dest*:*unitary-bound healthy-bounded-byD*)

lemma *healthy-id*[*simp*,*intro*!]:
healthy id
by(*simp add*:*healthyI feasibleI mono-transI scalingI*)

lemmas *healthy-fixes-bot* = *feasible-fixes-bot*[*OF healthy-feasibleD*]

Some additional results on *le-trans*, specific to *healthy* transformers.

lemma *le-trans-bot*[*intro*,*simp*]:
healthy t ⟹ *le-trans* (λ*P s*. *0*) *t*
by(*blast intro*:*le-funI*)

lemma *le-trans-top*[*intro*,*simp*]:
healthy t ⟹ *le-trans t* (λ*P s*. *bound-of P*)
by(*blast intro*!:*le-transI*[*OF le-funI*])

lemma *healthy-pr-bot*[*simp*]:
healthy t ⟹ *t* (λ*s*. *0*) = (λ*s*. *0*)
by(*blast intro*:*feasible-pr-0*)

The first significant result is that healthiness is preserved by equivalence:

lemma *healthy-equivI*:
fixes *t*::(′*s* ⇒ *real*) ⇒ ′*s* ⇒ *real* **and** *u*
assumes *equiv*: *equiv-trans t u*
and *healthy*: *healthy t*

```
 shows healthy u
proof
 have le-t-u: le-trans t u by(blast intro:equiv)
 have le-u-t: le-trans u t by(simp add:equiv-imp-le ac-simps equiv)
 from equiv have eq-u-t: equiv-trans u t by(simp add:ac-simps)

 show feasible u
 proof
  fix b and P::'s ⇒ real
  assume bP: bounded-by b P and nP: nneg P
  hence sP: sound P by(blast)
  with healthy have ⋀s. 0 ≤ t P s by(blast)
  also from sP and le-t-u have ⋀s. ... s ≤ u P s by(blast)
  finally show nneg (u P) by(blast)

  from sP and le-u-t have ⋀s. u P s ≤ t P s by(blast)
  also from healthy and sP and bP have ⋀s. t P s ≤ b by(blast)
  finally show bounded-by b (u P) by(blast)
 qed

 show mono-trans u
 proof
  fix P::'s ⇒ real and Q::'s ⇒ real
  assume sP: sound P and sQ: sound Q
    and le: P ⊩ Q
  from sP and le-u-t have u P ⊩ t P by(blast)
  also from sP and sQ and le and healthy have t P ⊩ t Q by(blast)
  also from sQ and le-t-u have t Q ⊩ u Q by(blast)
  finally show u P ⊩ u Q .
 qed

 show scaling u
 proof
  fix P::'s ⇒ real and c::real and x::'s
  assume sound: sound P
    and pos:  0 ≤ c

  hence bounded-by (c ∗ bound-of P) (λx. c ∗ P x)
   by(blast intro!:mult-left-mono dest!:less-imp-le)
  hence sc-bounded: bounded (λx. c ∗ P x)
   by(blast)
  moreover from sound and pos have sc-nneg: nneg (λx. c ∗ P x)
   by(blast intro:mult-nonneg-nonneg less-imp-le)
  ultimately have sc-sound: sound (λx. c ∗ P x) by(blast)

  show c ∗ u P x = u (λx. c ∗ P x) x
  proof −
   from sound have c ∗ u P x = c ∗ t P x
    by(simp add:equiv-transD[OF eq-u-t])
```

```
    also have ... = t (λx. c ∗ P x) x
      using healthy and sound and pos
      by(blast intro: scalingD)

    also from sc-sound and equiv have ... = u (λx. c ∗ P x) x
      by(blast intro:fun-cong)

    finally show ?thesis .
  qed
 qed
qed

lemma healthy-equiv:
 equiv-trans t u ⟹ healthy t ⟷ healthy u
 by(rule iffI, rule healthy-equivI, assumption+,
   simp add:healthy-equivI ac-simps)

lemma healthy-scale:
 fixes t::('s ⇒ real) ⇒ 's ⇒ real
 assumes ht: healthy t and nc: 0 ≤ c and bc: c ≤ 1
 shows healthy (λP s. c ∗ t P s)
proof
 show feasible (λP s. c ∗ t P s)
 proof
  fix b and P::'s ⇒ real
  assume nnP: nneg P and bP: bounded-by b P

  from ht nnP bP have ⋀s. t P s ≤ b by(blast)
  with nc have ⋀s. c ∗ t P s ≤ c ∗ b by(blast intro:mult-left-mono)
  also {
   from nnP and bP have 0 ≤ b by(auto)
   with bc have c ∗ b ≤ 1 ∗ b by(blast intro:mult-right-mono)
   hence c ∗ b ≤ b by(simp)
  }
  finally show bounded-by b (λs. c ∗ t P s) by(blast)

  from ht nnP bP have ⋀s. 0 ≤ t P s by(blast)
  with nc have ⋀s. 0 ≤ c ∗ t P s by(rule mult-nonneg-nonneg)
  thus nneg (λs. c ∗ t P s) by(blast)
 qed
 show mono-trans (λP s. c ∗ t P s)
 proof
  fix P::'s ⇒ real and Q
  assume sP: sound P and sQ: sound Q and le: P ⊩ Q
  with ht have ⋀s. t P s ≤ t Q s by(auto intro:le-funD)
  with nc have ⋀s. c ∗ t P s ≤ c ∗ t Q s
   by(blast intro:mult-left-mono)
  thus λs. c ∗ t P s ⊩ λs. c ∗ t Q s by(blast)
```

qed
from *ht* **show** *scaling* $(\lambda P\ s.\ c * t\ P\ s)$
by(*auto simp*:*scalingD healthy-scalingD ht*)
qed

lemma *healthy-top*[*iff*]:
healthy $(\lambda P\ s.\ \textit{bound-of}\ P)$
by(*auto intro*!:*healthy-parts*)

lemma *healthy-bot*[*iff*]:
healthy $(\lambda P\ s.\ 0)$
by(*auto intro*!:*healthy-parts*)

This weaker healthiness condition is for the liberal (wlp) semantics. We only insist that the transformer preserves *unitarity* (bounded by 1), and drop scaling (it is unnecessary in establishing the lattice structure here, unlike for the strict semantics).

definition
nearly-healthy :: $(('s \Rightarrow real) \Rightarrow ('s \Rightarrow real)) \Rightarrow bool$
where
nearly-healthy t $\longleftrightarrow (\forall P.\ unitary\ P \longrightarrow unitary\ (t\ P)) \wedge$
$(\forall P\ Q.\ unitary\ P \longrightarrow unitary\ Q \longrightarrow P \Vdash Q \longrightarrow t\ P \Vdash t\ Q)$

lemma *nearly-healthyI*[*intro*]:
$[\![\bigwedge P.\ unitary\ P \Longrightarrow unitary\ (t\ P);$
$\bigwedge P\ Q.\ [\![unitary\ P;\ unitary\ Q;\ P \Vdash Q]\!] \Longrightarrow t\ P \Vdash t\ Q]\!] \Longrightarrow$ *nearly-healthy t*
by(*simp add*:*nearly-healthy-def*)

lemma *nearly-healthy-monoD*[*dest*]:
$[\![$ *nearly-healthy t*; $P \Vdash Q;\ unitary\ P;\ unitary\ Q]\!] \Longrightarrow t\ P \Vdash t\ Q$
by(*simp add*:*nearly-healthy-def*)

lemma *nearly-healthy-unitaryD*[*dest*]:
$[\![$ *nearly-healthy t*; *unitary P* $]\!] \Longrightarrow unitary\ (t\ P)$
by(*simp add*:*nearly-healthy-def*)

lemma *healthy-nearly-healthy*[*dest*]:
assumes *ht*: *healthy t*
shows *nearly-healthy t*
by(*intro nearly-healthyI*, *auto intro*:*mono-transD*[*OF healthy-monoD*, *OF ht*] *ht*)

lemmas *nearly-healthy-id*[*iff*] =
healthy-nearly-healthy[*OF healthy-id*, *unfolded id-def*]

3.2.3 Sublinearity

As already mentioned, the core healthiness property (aside from feasibility and continuity) for transformers is *sublinearity*: The transformation of a quasi-linear combination of sound expectations is greater than the same combination applied to the transformation of the expectations themselves. The term $x \ominus y$ represents *truncated subtraction* i.e. *max* $(x - y)$ *0* (see Section 4.13.1).

definition *sublinear* ::
$(('s \Rightarrow real) \Rightarrow ('s \Rightarrow real)) \Rightarrow bool$
where
$sublinear\ t \longleftrightarrow (\forall a\ b\ c\ P\ Q\ s.\ (sound\ P \wedge sound\ Q \wedge 0 \leq a \wedge 0 \leq b \wedge 0 \leq c) \longrightarrow$
$a * t\ P\ s + b * t\ Q\ s \ominus c$
$\leq t\ (\lambda s'.\ a * P\ s' + b * Q\ s' \ominus c)\ s)$

lemma *sublinearI*[*intro*]:
$[\![\bigwedge a\ b\ c\ P\ Q\ s.\ [\![sound\ P;\ sound\ Q;\ 0 \leq a;\ 0 \leq b;\ 0 \leq c]\!] \Longrightarrow$
$a * t\ P\ s + b * t\ Q\ s \ominus c \leq$
$t\ (\lambda s'.\ a * P\ s' + b * Q\ s' \ominus c)\ s]\!] \Longrightarrow sublinear\ t$
by(*simp add*:*sublinear-def*)

lemma *sublinearD*[*dest*]:
$[\![sublinear\ t;\ sound\ P;\ sound\ Q;\ 0 \leq a;\ 0 \leq b;\ 0 \leq c]\!] \Longrightarrow$
$a * t\ P\ s + b * t\ Q\ s \ominus c \leq$
$t\ (\lambda s'.\ a * P\ s' + b * Q\ s' \ominus c)\ s$
by(*simp add*:*sublinear-def*)

It is easier to see the relevance of sublinearity by breaking it into several component properties, as in the following sections.

Sub-additivity

definition *sub-add* ::
$(('s \Rightarrow real) \Rightarrow ('s \Rightarrow real)) \Rightarrow bool$
where
$sub\text{-}add\ t \longleftrightarrow (\forall P\ Q\ s.\ (sound\ P \wedge sound\ Q) \longrightarrow$
$t\ P\ s + t\ Q\ s \leq t\ (\lambda s'.\ P\ s' + Q\ s')\ s)$

Sub-additivity, together with scaling (Section 3.2.2) gives the *linear* portion of sublinearity. Together, these two properties are equivalent to *convexity*, as Figure 3.4 illustrates by analogy.

Here P is an affine function (expectation) *real* $\Rightarrow$ *real*, restricted to some finite interval. In practice the state space (the left-hand type) is typically discrete and multi-dimensional, but on the reals we have a convenient geometrical intuition. The lines tP and uP represent the effect of two healthy transformers (again affine). Neither monotonicity nor scaling are represented, but both are feasible: Both lines are bounded above by the greatest value of P.

The curve Q is the pointwise minimum of tP and tQ, written $tP \sqcap tQ$. This is, not coincidentally, the syntax for a binary nondeterministic choice in pGCL: The probability that some property is established by the choice between programs a and b cannot be guaranteed to be any higher than either the probability under a, or that under b.

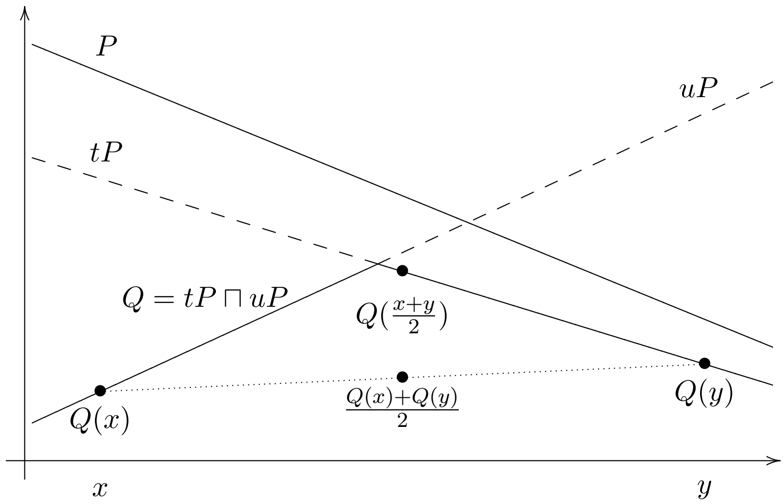

Figure 3.4: A graphical depiction of sub-additivity as convexity.

The original curve, P, is trivially convex—it is linear. Also, both t and u, and the operator $\sqcap$ preserve convexity. A probabilistic choice will also preserve it. The preservation of convexity is a property of sub-additive transformers that respect scaling. Note the form of the definition of convexity:

$$\forall x, y. \frac{Q(x) + Q(y)}{2} \leq Q(\frac{x+y}{2})$$

Were we to replace Q by some sub-additive transformer v, and x and y by expectations R and S, the equivalent expression:

$$\frac{vR + vS}{2} \leq v(\frac{R+S}{2})$$

Can be rewritten, using scaling, to:

$$\frac{1}{2}(vR + vS) \leq \frac{1}{2}v(R+S)$$

Which holds everywhere exactly when v is sub-additive i.e.:

$$vR + vS \leq v(R+S)$$

lemma *sub-addI*[*intro*]:
$[\![\bigwedge P\ Q\ s.\ [\![\mathit{sound}\ P; \mathit{sound}\ Q]\!] \Longrightarrow$
$t\ P\ s + t\ Q\ s \leq t\ (\lambda s'.\ P\ s' + Q\ s')\ s]\!] \Longrightarrow \mathit{sub\text{-}add}\ t$
by(*simp add*:*sub-add-def*)

lemma *sub-addI2*:

```
  ⟦⋀P Q. ⟦ sound P; sound Q ⟧ ⟹
        λs. t P s + t Q s ⊩ t (λs. P s + Q s)⟧ ⟹
   sub-add t
  by(auto)

lemma sub-addD[dest]:
  ⟦ sub-add t; sound P; sound Q ⟧ ⟹ t P s + t Q s ≤ t (λs'. P s' + Q s') s
  by(simp add:sub-add-def)

lemma equiv-sub-add:
  fixes t::('s ⇒ real) ⇒ 's ⇒ real
  assumes eq: equiv-trans t u
     and sa: sub-add t
  shows sub-add u
proof
  fix P::'s ⇒ real and Q::'s ⇒ real and s::'s
  assume sP: sound P and sQ: sound Q

  with eq have u P s + u Q s = t P s + t Q s
   by(simp add:equiv-transD)
  also from sP sQ sa have t P s + t Q s ≤ t (λs. P s + Q s) s
   by(auto)
  also {
   from sP sQ have sound (λs. P s + Q s) by(auto)
   with eq have t (λs. P s + Q s) s = u (λs. P s + Q s) s
     by(simp add:equiv-transD)
  }
  finally show u P s + u Q s ≤ u (λs. P s + Q s) s .
qed
```

Sublinearity and feasibility imply sub-additivity.

```
lemma sublinear-subadd:
  fixes t::('s ⇒ real) ⇒ 's ⇒ real
  assumes slt: sublinear t
     and ft: feasible t
  shows sub-add t
proof
  fix P::'s ⇒ real and Q::'s ⇒ real and s::'s
  assume sP: sound P and sQ: sound Q

  with ft have sound (t P) sound (t Q) by(auto)
  hence 0 ≤ t P s and 0 ≤ t Q s by(auto)
  hence 0 ≤ t P s + t Q s by(auto)
  hence ... = ...⊖ 0 by(simp)

  also from sP sQ
  have ... ≤ t (λs. P s + Q s ⊖ 0) s
   by(rule sublinearD[OF slt, where a=1 and b=1 and c=0, simplified])
```

```
  also {
    from sP sQ have ⋀s. 0 ≤ P s and ⋀s. 0 ≤ Q s by(auto)
    hence ⋀s. 0 ≤ P s + Q s by(auto)
    hence t (λs. P s + Q s ⊖ 0) s = t (λs. P s + Q s) s
      by(simp)
  }

  finally show t P s + t Q s ≤ t (λs. P s + Q s) s .
qed
```

A few properties following from sub-additivity:

```
lemma standard-negate:
  assumes ht: healthy t
      and sat: sub-add t
  shows t «P» s + t «𝒩 P» s ≤ 1
proof −
  from sat have t «P» s + t «𝒩 P» s ≤ t (λs. «P» s + «𝒩 P» s) s by(auto)
  also have ... = t (λs. 1) s by(simp add:negate-embed)
  also {
    from ht have bounded-by 1 (t (λs. 1)) by(auto)
    hence t (λs. 1) s ≤ 1 by(auto)
  }
  finally show ?thesis .
qed

lemma sub-add-sum:
  fixes t::'s trans and S::'a set
  assumes sat: sub-add t
      and ht: healthy t
      and sP: ⋀x. sound (P x)
  shows (λx. ∑y∈S. t (P y) x) ≤ t (λx. ∑y∈S. P y x)
proof(cases infinite S, simp-all add:ht)
  assume fS: finite S
  show ?thesis
  proof(rule finite-induct[OF fS le-funI le-funI], simp-all)
    fix s::'s
    from ht have sound (t (λs. 0)) by(auto)
    thus 0 ≤ t (λs. 0) s by(auto)

    fix F::'a set and x::'a
    assume IH: λa. ∑y∈F. t (P y) a ⊩ t (λx. ∑y∈F. P y x)
    hence t (P x) s + (∑y∈F. t (P y) s) ≤
          t (P x) s + t (λx. ∑y∈F. P y x) s
      by(auto intro:add-left-mono)
    also from sat sP
    have ... ≤ t (λxa. P x xa + (∑y∈F. P y xa)) s
      by(auto intro!:sub-addD[OF sat] sum-sound)
    finally
    show t (P x) s + (∑y∈F. t (P y) s) ≤
```

```
    t (λxa. P x xa + (∑y∈F. P y xa)) s .
 qed
qed

lemma sub-add-guard-split:
 fixes t::'s::finite trans and P::'s expect and s::'s
 assumes sat: sub-add t
   and ht: healthy t
   and sP: sound P
 shows (∑y∈{s. G s}.  P y ∗ t « λz. z = y » s) +
      (∑y∈{s. ¬G s}. P y ∗ t « λz. z = y » s) ≤ t P s
proof −
 have {s. G s} ∩ {s. ¬G s} = {} by(blast)
 hence (∑y∈{s. G s}.  P y ∗ t « λz. z = y » s) +
      (∑y∈{s. ¬G s}. P y ∗ t « λz. z = y » s) =
      (∑y∈({s. G s} ∪ {s. ¬G s}). P y ∗ t « λz. z = y » s)
  by(auto intro: sum.union-disjoint[symmetric])
 also {
  have {s. G s} ∪ {s. ¬G s} = UNIV by (blast)
  hence (∑y∈({s. G s} ∪ {s. ¬G s}). P y ∗ t « λz. z = y » s) =
        (λx. ∑y∈UNIV. P y ∗ t (λx. «λz. z = y» x) x) s
   by(simp)
 }
 also {
  from sP have ⋀y. 0 ≤ P y by(auto)
  with healthy-scalingD[OF ht]
  have (λx. ∑y∈UNIV. P y ∗ t (λx. «λz. z = y» x) x) s =
       (λx. ∑y∈UNIV. t (λx. P y ∗ «λz. z = y» x) x) s
   by(simp add:scalingD)
 }
 also {
  from sat ht sP
  have (λx. ∑y∈UNIV. t (λx. P y ∗ «λz. z = y» x) x) ≤
       t (λx. ∑y∈UNIV. P y ∗ «λz. z = y» x)
   by(intro sub-add-sum sound-intros, auto)
  hence (λx. ∑y∈UNIV. t (λx. P y ∗ «λz. z = y» x) x) s ≤
       t (λx. ∑y∈UNIV. P y ∗ «λz. z = y» x) s by(auto)
 }
 also {
  have rw1: (λx. ∑y∈UNIV. P y ∗ «λz. z = y» x) =
          (λx. ∑y∈UNIV. if y = x then P y else 0)
   by (rule ext [OF sum.cong]) auto
  also from sP have ... ⊩ P
   by(cases finite (UNIV::'s set), auto simp:sum.delta)
  finally have leP: λx. ∑y∈UNIV. P y ∗ « λz. z = y » x ⊩ P .
  moreover have sound (λx. ∑y∈UNIV. P y ∗ «λz. z = y» x)
  proof(intro soundI2 bounded-byI nnegI sum-nonneg ballI)
   fix x
   from leP have (∑y∈UNIV. P y ∗ « λz. z = y » x) ≤ P x by(auto)
```

```
    also from sP have ... ≤ bound-of P by(auto)
    finally show (∑ y∈UNIV. P y ∗ « λz. z = y » x) ≤ bound-of P .
    fix y
    from sP show 0 ≤ P y ∗ « λz. z = y » x
      by(auto intro:mult-nonneg-nonneg)
  qed
  ultimately have t (λx. ∑ y∈UNIV. P y ∗ «λz. z = y» x) s ≤ t P s
    using sP by(auto intro:le-funD[OF mono-transD, OF healthy-monoD, OF ht])
 }
 finally show ?thesis .
qed
```

Sub-distributivity

```
definition sub-distrib ::
  (('s ⇒ real) ⇒ ('s ⇒ real)) ⇒ bool
where
  sub-distrib t ⟷ (∀ P s. sound P ⟶ t P s ⊖ 1 ≤ t (λs'. P s' ⊖ 1) s)

lemma sub-distribI[intro]:
  ⟦ ⋀P s. sound P ⟹ t P s ⊖ 1 ≤ t (λs'. P s' ⊖ 1) s ⟧ ⟹ sub-distrib t
  by(simp add:sub-distrib-def)

lemma sub-distribI2:
  ⟦ ⋀P. sound P ⟹ λs. t P s ⊖ 1 ⊩ t (λs. P s ⊖ 1) ⟧ ⟹ sub-distrib t
  by(auto)

lemma sub-distribD[dest]:
  ⟦ sub-distrib t; sound P ⟧ ⟹ t P s ⊖ 1 ≤ t (λs'. P s' ⊖ 1) s
  by(simp add:sub-distrib-def)

lemma equiv-sub-distrib:
  fixes t::('s ⇒ real) ⇒ 's ⇒ real
  assumes eq: equiv-trans t u
    and sd: sub-distrib t
  shows sub-distrib u
proof
  fix P::'s ⇒ real and s::'s
  assume sP: sound P

  with eq have u P s ⊖ 1 = t P s ⊖ 1 by(simp add:equiv-transD)
  also from sP sd have ... ≤ t (λs. P s ⊖ 1) s by(auto)
  also from sP eq have ... = u (λs. P s ⊖ 1) s
    by(simp add:equiv-transD tminus-sound)
  finally show u P s ⊖ 1 ≤ u (λs. P s ⊖ 1) s .
qed
```

Sublinearity implies sub-distributivity:

```
lemma sublinear-sub-distrib:
```

```
  fixes t::('s ⇒ real) ⇒ 's ⇒ real
  assumes slt: sublinear t
  shows sub-distrib t
proof
  fix P::'s ⇒ real and s::'s
  assume sP: sound P
  moreover have sound (λ-. 0) by(auto)
  ultimately show t P s ⊖ 1 ≤ t (λs. P s ⊖ 1) s
    by(rule sublinearD[OF slt, where a=1 and b=0 and c=1, simplified])
qed
```

Healthiness, sub-additivity and sub-distributivity imply sublinearity. This is how we usually show sublinearity.

```
lemma sd-sa-sublinear:
  fixes t::('s ⇒ real) ⇒ 's ⇒ real
  assumes sdt: sub-distrib t and sat: sub-add t and ht: healthy t
  shows sublinear t
proof
  fix P::'s ⇒ real and Q::'s ⇒ real and s::'s
  and a::real and b::real and c::real
  assume sP: sound P and sQ: sound Q
     and nna: 0 ≤ a and nnb: 0 ≤ b and nnc: 0 ≤ c

  from ht sP sQ nna nnb
  have saP: sound (λs. a ∗ P s) and staP: sound (λs. a ∗ t P s)
   and sbQ: sound (λs. b ∗ Q s) and stbQ: sound (λs. b ∗ t Q s)
    by(auto intro:sc-sound)
  hence sabPQ:  sound (λs. a ∗ P s + b ∗ Q s)
    and stabPQ: sound (λs. a ∗ t P s + b ∗ t Q s)
    by(auto intro:sound-sum)

  from ht sP sQ nna nnb
  have a ∗ t P s + b ∗ t Q s = t (λs. a ∗ P s) s + t (λs. b ∗ Q s) s
    by(simp add:scalingD healthy-scalingD)
  also from saP sbQ sat
  have t (λs. a ∗ P s) s + t (λs. b ∗ Q s) s ≤
       t (λs. a ∗ P s + b ∗ Q s) s by(blast)
  finally
  have notm: a ∗ t P s + b ∗ t Q s ≤ t (λs. a ∗ P s + b ∗ Q s) s .

  show a ∗ t P s + b ∗ t Q s ⊖ c ≤ t (λs'. a ∗ P s' + b ∗ Q s' ⊖ c) s
  proof(cases c = 0)
    case True note z = this
    from stabPQ have ⋀s. 0 ≤ a ∗ t P s + b ∗ t Q s by(auto)
    moreover from sabPQ
    have ⋀s. 0 ≤ a ∗ P s + b ∗ Q s by(auto)
    ultimately show ?thesis by(simp add:z notm)
  next
    case False note nz = this
```

```
from nz and nnc have nni: 0 ≤ inverse c by(auto)

have ⋀s. (inverse c ∗ a) ∗ P s + (inverse c ∗ b) ∗ Q s =
      inverse c ∗ (a ∗ P s + b ∗ Q s)
 by(simp add: divide-simps)
with sabPQ and nni
have si: sound (λs. (inverse c ∗ a) ∗ P s + (inverse c ∗ b) ∗ Q s)
 by(auto intro:sc-sound)
hence sim: sound (λs. (inverse c ∗ a) ∗ P s + (inverse c ∗ b) ∗ Q s ⊖ 1)
 by(auto intro!:tminus-sound)

from nz
have a ∗ t P s + b ∗ t Q s ⊖ c =
    (c ∗ inverse c) ∗ a ∗ t P s +
    (c ∗ inverse c) ∗ b ∗ t Q s ⊖ c
 by(simp)
also
have ... = c ∗ (inverse c ∗ a ∗ t P s) +
       c ∗ (inverse c ∗ b ∗ t Q s) ⊖ c
 by(simp add:field-simps)
also from nnc
have ... = c ∗ (inverse c ∗ a ∗ t P s + inverse c ∗ b ∗ t Q s ⊖ 1)
 by(simp add:distrib-left tminus-left-distrib)
also {
  have X: ⋀s. (inverse c ∗ a) ∗ t P s + (inverse c ∗ b) ∗ t Q s =
         inverse c ∗ (a ∗ t P s + b ∗ t Q s) by(simp add: divide-simps)
  also from nni and notm
  have inverse c ∗ (a ∗ t P s + b ∗ t Q s) ≤
      inverse c ∗ (t (λs. a ∗ P s + b ∗ Q s) s)
   by(blast intro:mult-left-mono)
  also from nni ht sabPQ
  have ... = t (λs. (inverse c ∗ a) ∗ P s + (inverse c ∗ b) ∗ Q s) s
   by(simp add:scalingD[OF healthy-scalingD, OF ht] algebra-simps)
  finally
  have (inverse c ∗ a) ∗ t P s + (inverse c ∗ b) ∗ t Q s ⊖ 1 ≤
      t (λs. (inverse c ∗ a) ∗ P s + (inverse c ∗ b) ∗ Q s) s ⊖ 1
   by(rule tminus-left-mono)
  also {
    from sdt si
    have t (λs. (inverse c ∗ a) ∗ P s + (inverse c ∗ b) ∗ Q s) s ⊖ 1 ≤
        t (λs. (inverse c ∗ a) ∗ P s + (inverse c ∗ b) ∗ Q s ⊖ 1) s
     by(blast)
  }
  finally
  have c ∗ (inverse c ∗ a ∗ t P s + inverse c ∗ b ∗ t Q s ⊖ 1) ≤
      c ∗ t (λs. inverse c ∗ a ∗ P s + inverse c ∗ b ∗ Q s ⊖ 1) s
   using nnc by(blast intro:mult-left-mono)
}
also from nnc ht sim
```

```
have c ∗ t (λs. inverse c ∗ a ∗ P s + inverse c ∗ b ∗ Q s ⊖ 1) s
    = t (λs. c ∗ (inverse c ∗ a ∗ P s + inverse c ∗ b ∗ Q s ⊖ 1)) s
  by(simp add:scalingD healthy-scalingD)
also from nnc
have ... = t (λs. c ∗ (inverse c ∗ a ∗ P s) +
               c ∗ (inverse c ∗ b ∗ Q s) ⊖ c) s
  by(simp add:distrib-left tminus-left-distrib)
also have ... = t (λs. (c ∗ inverse c) ∗ a ∗ P s +
                  (c ∗ inverse c) ∗ b ∗ Q s ⊖ c) s
  by(simp add:field-simps)
finally
show a ∗ t P s + b ∗ t Q s ⊖ c ≤ t (λs'. a ∗ P s' + b ∗ Q s' ⊖ c) s
  using nz by(simp)
qed
qed
```

Sub-conjunctivity

```
definition
  sub-conj :: (('s ⇒ real) ⇒ 's ⇒ real) ⇒ bool
where
  sub-conj t ≡ ∀ P Q. (sound P ∧ sound Q) ⟶
               t P && t Q ⊩ t (P && Q)

lemma sub-conjI[intro]:
  ⟦ ⋀P Q. ⟦ sound P; sound Q ⟧ ⟹
        t P && t Q ⊩ t (P && Q) ⟧ ⟹ sub-conj t
  unfolding sub-conj-def by(simp)

lemma sub-conjD[dest]:
  ⟦ sub-conj t; sound P; sound Q ⟧ ⟹ t P && t Q ⊩ t (P && Q)
  unfolding sub-conj-def by(simp)

lemma sub-conj-wp-twice:
  fixes f::'s ⇒ (('s ⇒ real) ⇒ 's ⇒ real)
  assumes all: ∀ s. sub-conj (f s)
  shows sub-conj (λP s. f s P s)
proof(rule sub-conjI, rule le-funI)
  fix P::'s ⇒ real and Q::'s ⇒ real and s
  assume sP: sound P and sQ: sound Q

  have ((λs. f s P s) && (λs. f s Q s)) s = (f s P && f s Q) s
    by(simp add:exp-conj-def)
  also {
    from all have sub-conj (f s) by(blast)
    with sP and sQ have (f s P && f s Q) s ≤ f s (P && Q) s
      by(blast)
  }
  finally show ((λs. f s P s) && (λs. f s Q s)) s ≤ f s (P && Q) s .
```

qed

Sublinearity implies sub-conjunctivity:

lemma *sublinear-sub-conj*:
fixes t::$(\,'s \Rightarrow real) \Rightarrow \,'s \Rightarrow real$
assumes *slt*: *sublinear t*
shows *sub-conj t*
proof(*rule sub-conjI*, *rule le-funI*, *unfold exp-conj-def pconj-def*)
fix P::$'s \Rightarrow real$ **and** Q::$'s \Rightarrow real$**and** s::$'s$
assume *sP*: *sound P* **and** *sQ*: *sound Q*
thus $t\ P\ s + t\ Q\ s \ominus 1 \leq t\ (\lambda s.\ P\ s + Q\ s \ominus 1)\ s$
by(*rule sublinearD*[*OF slt*, **where** *a=1* **and** *b=1* **and** *c=1*, *simplified*])
qed

Sublinearity under equivalence

Sublinearity is preserved by equivalence.

lemma *equiv-sublinear*:
$[\![$ *equiv-trans t u*; *sublinear t*; *healthy t* $]\!] \Longrightarrow$ *sublinear u*
by(*iprover intro*:*sd-sa-sublinear healthy-equivI*
dest:*equiv-sub-distrib equiv-sub-add*
sublinear-sub-distrib sublinear-subadd
healthy-feasibleD)

3.2.4 Determinism

Transformers which are both additive, and maximal among those that satisfy feasibility are *deterministic*, and will turn out to be maximal in the refinement order.

Additivity

Full additivity is not generally satisfied. It holds for (sub-)probabilistic transformers however.

definition
additive :: $((\,'a \Rightarrow real) \Rightarrow \,'a \Rightarrow real) \Rightarrow bool$
where
$additive\ t \equiv \forall\, P\ Q.\ (sound\ P \wedge sound\ Q) \longrightarrow$
$t\ (\lambda s.\ P\ s + Q\ s) = (\lambda s.\ t\ P\ s + t\ Q\ s)$

lemma *additiveD*:
$[\![$ *additive t*; *sound P*; *sound Q* $]\!] \Longrightarrow t\ (\lambda s.\ P\ s + Q\ s) = (\lambda s.\ t\ P\ s + t\ Q\ s)$
by(*simp add*:*additive-def*)

lemma *additiveI*[*intro*]:
$[\![\ \bigwedge P\ Q\ s.\ [\![$ *sound P*; *sound Q* $]\!] \Longrightarrow t\ (\lambda s.\ P\ s + Q\ s)\ s = t\ P\ s + t\ Q\ s\ ]\!] \Longrightarrow$
additive t
unfolding *additive-def* **by**(*blast*)

Additivity is strictly stronger than sub-additivity.

lemma *additive-sub-add*:
additive t $\Longrightarrow$ *sub-add t*
by(*simp add*:*sub-addI additiveD*)

The additivity property extends to finite summation.

lemma *additive-sum*:
fixes *S*::*'s set*
assumes *additive*: *additive t*
and *healthy*: *healthy t*
and *finite*: *finite S*
and *sPz*: $\bigwedge z.$ *sound* (*P z*)
shows $t\ (\lambda x.\ \sum y{\in}S.\ P\ y\ x) = (\lambda x.\ \sum y{\in}S.\ t\ (P\ y)\ x)$
proof(*rule finite-induct*, *simp-all add*:*assms*)
fix *z*::*'s* **and** *T*::*'s set*
assume *finT*: *finite T*
and *IH*: $t\ (\lambda x.\ \sum y{\in}T.\ P\ y\ x) = (\lambda x.\ \sum y{\in}T.\ t\ (P\ y)\ x)$

from *additive sPz*
have $t\ (\lambda x.\ P\ z\ x + (\sum y{\in}T.\ P\ y\ x)) =$
$(\lambda x.\ t\ (P\ z)\ x + t\ (\lambda x.\ \sum y{\in}T.\ P\ y\ x)\ x)$
by(*auto intro*!:*sum-sound additiveD*)
also from *IH*
have ... $= (\lambda x.\ t\ (P\ z)\ x + (\sum y{\in}T.\ t\ (P\ y)\ x))$
by(*simp*)
finally show $t\ (\lambda x.\ P\ z\ x + (\sum y{\in}T.\ P\ y\ x)) =$
$(\lambda x.\ t\ (P\ z)\ x + (\sum y{\in}T.\ t\ (P\ y)\ x))$ **.**
qed

An additive transformer (over a finite state space) is linear: it is simply the weighted sum of final expectation values, the weights being the probability of reaching a given final state. This is useful for reasoning using the forward, or "gambling game" interpretation.

lemma *additive-delta-split*:
fixes *t*::(*'s*::*finite* $\Rightarrow$ *real*) $\Rightarrow$ *'s* $\Rightarrow$ *real*
assumes *additive*: *additive t*
and *ht*: *healthy t*
and *sP*: *sound P*
shows $t\ P\ x = (\sum y{\in}UNIV.\ P\ y * t$ «$\lambda z.\ z = y$» $x)$
proof $-$
have $\bigwedge x.\ (\sum y{\in}UNIV.\ P\ y *$ «$\lambda z.\ z = y$» $x) =$
$(\sum y{\in}UNIV.$ *if* $y = x$ *then* $P\ y$ *else* $0)$
by (*rule sum.cong*) *auto*
also have $\bigwedge x.$ ... $x = P\ x$
by(*simp add*:*sum.delta*)
finally
have $t\ P\ x = t\ (\lambda x.\ \sum y{\in}UNIV.\ P\ y *$ «$\lambda z.\ z = y$» $x)\ x$
by(*simp*)

```
  also {
    from sP have ⋀z. sound (λa. P z ∗ « λza. za = z » a)
      by(auto intro!:mult-sound)
    hence t (λx. ∑ y∈UNIV. P y ∗ «λz. z = y» x) x =
          (∑ y∈UNIV. t (λx. P y ∗ «λz. z = y» x) x)
      by(subst additive-sum, simp-all add:assms)
  }
  also from sP
  have (∑ y∈UNIV. t (λx. P y ∗ «λz. z = y» x) x) =
       (∑ y∈UNIV. P y ∗ t «λz. z = y» x)
    by(subst scalingD[OF healthy-scalingD, OF ht], auto)
  finally show t P x = (∑ y∈UNIV. P y ∗ t « λz. z = y » x) .
qed
```

We can group the states in the linear form, to split on the value of a predicate (guard).

```
lemma additive-guard-split:
  fixes t::('s::finite ⇒ real) ⇒ 's ⇒ real
  assumes additive: additive t
    and ht: healthy t
    and sP: sound P
  shows t P x = (∑ y∈{s.   G s}. P y ∗ t «λz. z = y» x) +
               (∑ y∈{s. ¬ G s}. P y ∗ t «λz. z = y» x)
proof −
  from assms
  have t P x = (∑ y∈UNIV. P y ∗ t «λz. z = y» x)
    by(rule additive-delta-split)
  also {
    have UNIV = {s. G s} ∪ {s. ¬ G s}
      by(auto)
    hence (∑ y∈UNIV. P y ∗ t «λz. z = y» x) =
          (∑ y∈{s. G s} ∪ {s. ¬ G s}. P y ∗ t «λz. z = y» x)
      by(simp)
  }
  also
  have (∑ y∈{s. G s} ∪ {s. ¬ G s}. P y ∗ t «λz. z = y» x) =
       (∑ y∈{s.   G s}. P y ∗ t «λz. z = y» x) +
       (∑ y∈{s. ¬ G s}. P y ∗ t «λz. z = y» x)
    by(auto intro:sum.union-disjoint)
  finally show ?thesis .
qed
```

Maximality

```
definition
  maximal :: (('a ⇒ real) ⇒ 'a ⇒ real) ⇒ bool
where
  maximal t ≡ ∀ c. 0 ≤ c ⟶ t (λ-. c) = (λ-. c)
```

lemma *maximalI*[*intro*]:
$\llbracket \bigwedge c.\ 0 \le c \Longrightarrow t\ (\lambda\text{-}.\ c) = (\lambda\text{-}.\ c)\ \rrbracket \Longrightarrow maximal\ t$
by(*simp add*:*maximal-def*)

lemma *maximalD*[*dest*]:
$\llbracket\ maximal\ t;\ 0 \le c\ \rrbracket \Longrightarrow t\ (\lambda\text{-}.\ c) = (\lambda\text{-}.\ c)$
by(*simp add*:*maximal-def*)

A transformer that is both additive and maximal is deterministic:

definition *determ* :: $(('a \Rightarrow real) \Rightarrow 'a \Rightarrow real) \Rightarrow bool$
where
determ t $\equiv$ *additive t* $\wedge$ *maximal t*

lemma *determI*[*intro*]:
$\llbracket\ additive\ t;\ maximal\ t\ \rrbracket \Longrightarrow determ\ t$
by(*simp add*:*determ-def*)

lemma *determ-additiveD*[*intro*]:
determ t $\Longrightarrow$ *additive t*
by(*simp add*:*determ-def*)

lemma *determ-maximalD*[*intro*]:
determ t $\Longrightarrow$ *maximal t*
by(*simp add*:*determ-def*)

For a fully-deterministic transformer, a transformed standard expectation, and its transformed negation are complementary.

lemma *determ-negate*:
assumes *determ*: *determ t*
shows t «P» $s + t$ «$\mathcal{N}\ P$» $s = 1$
proof −
have t «P» $s + t$ «$\mathcal{N}\ P$» $s = t\ (\lambda s.$ «P» $s +$ «$\mathcal{N}\ P$» $s)\ s$
by(*simp add*:*additiveD determ determ-additiveD*)
also {
have $\bigwedge s.$ «P» $s +$ «$\mathcal{N}\ P$» $s = 1$
by(*case-tac P s*, *simp-all*)
hence $t\ (\lambda s.$ «P» $s +$ «$\mathcal{N}\ P$» $s) = t\ (\lambda s.\ 1)$
by(*simp*)
}
also have $t\ (\lambda s.\ 1) = (\lambda s.\ 1)$
by(*simp add*:*maximalD determ determ-maximalD*)
finally show *?thesis* **.**
qed

3.2.5 Modular Reasoning

The emphasis of a mechanised logic is on automation, and letting the computer tackle the large, uninteresting problems. However, as terms generally grow exponentially in the size of a program, it is still essential to break up a proof and reason in a modular fashion.

The following rules allow proof decomposition, and later will be incorporated into a verification condition generator.

lemma *entails-combine*:
  assumes *wp1*: $P \Vdash t\ R$
    and *wp2*: $Q \Vdash t\ S$
    and *sc*: *sub-conj t*
    and *sR*: *sound R*
    and *sS*: *sound S*
  shows $P\ \&\&\ Q \Vdash t\ (R\ \&\&\ S)$
proof −
  from *wp1* **and** *wp2* **have** $P\ \&\&\ Q \Vdash t\ R\ \&\&\ t\ S$
    by(*blast intro*:*entails-frame*)
  also from *sc* **and** *sR* **and** *sS* **have** $\ldots \leq t\ (R\ \&\&\ S)$
    by(*rule sub-conjD*)
  finally show *?thesis* **.**
qed

These allow mismatched results to be composed

lemma *entails-strengthen-post*:
  $[\![\ P \Vdash t\ Q;\ healthy\ t;\ sound\ R;\ Q \Vdash R;\ sound\ Q\ ]\!] \Longrightarrow P \Vdash t\ R$
  by(*blast intro*:*entails-trans*)

lemma *entails-weaken-pre*:
  $[\![\ Q \Vdash t\ R;\ P \vdash Q\ ]\!] \Longrightarrow P \Vdash t\ R$
  by(*blast intro*:*entails-trans*)

This rule is unique to pGCL. Use it to scale the post-expectation of a rule to 'fit under' the precondition you need to satisfy.

lemma *entails-scale*:
  assumes *wp*: $P \Vdash t\ Q$ **and** *h*: *healthy t*
    and *sQ*: *sound Q* **and** *pos*: $0 \leq c$
  shows $(\lambda s.\ c * P\ s) \Vdash t\ (\lambda s.\ c * Q\ s)$
proof(*rule le-funI*)
  fix *s*
  from *pos* **and** *wp* **have** $c * P\ s \leq c * t\ Q\ s$
    by(*auto intro*:*mult-left-mono*)
  with *sQ pos h* **show** $c * P\ s \leq t\ (\lambda s.\ c * Q\ s)\ s$
    by(*simp add*:*scalingD healthy-scalingD*)
qed

3.2.6 Transforming Standard Expectations

Reasoning with *standard* expectations, those obtained by embedding a predicate, is often easier, as the analogues of many familiar boolean rules hold in modified form.

One may use a standard pre-expectation as an assumption:

lemma *use-premise*:
 assumes *h*: *healthy t* **and** *wP*: $\bigwedge s.\ P\ s \Longrightarrow 1 \leq t$ «*Q*» *s*
 shows «*P*» ⊩ *t* «*Q*»
proof(*rule entailsI*)
 fix *s* **show** «*P*» *s* ≤ *t* «*Q*» *s*
 proof(*cases P s*)
  case *True* **with** *wP* **show** *?thesis* **by**(*auto*)
 next
  case *False* **with** *h* **show** *?thesis* **by**(*auto*)
 qed
qed

The other direction works too.

lemma *fold-premise*:
 assumes *ht*: *healthy t*
 and *wp*: «*P*» ⊩ *t* «*Q*»
 shows ∀ *s*. *P s* ⟶ *1* ≤ *t* «*Q*» *s*
proof(*clarify*)
 fix *s* **assume** *P s*
 hence *1* = «*P*» *s* **by**(*simp*)
 also from *wp* **have** ... ≤ *t* «*Q*» *s* **by**(*auto*)
 finally show *1* ≤ *t* «*Q*» *s* **.**
qed

Predicate conjunction behaves as expected:

lemma *conj-post*:
 ⟦ *P* ⊩ *t* «λ*s*. *Q s* ∧ *R s*»; *healthy t* ⟧ ⟹ *P* ⊩ *t* «*Q*»
 by(*blast intro*:*entails-strengthen-post implies-entails*)

Similar to ⟦*healthy ?t*; $\bigwedge$*s*. *?P s* ⟹ *1* ≤ *?t* « *?Q* » *s*⟧ ⟹ « *?P* » ⊩ *?t* « *?Q* », but more general.

lemma *entails-pconj-assumption*:
 assumes *f*: *feasible t* **and** *wP*: $\bigwedge$*s*. *P s* ⟹ *Q s* ≤ *t R s*
  and *uQ*: *unitary Q* **and** *uR*: *unitary R*
 shows «*P*» && *Q* ⊩ *t R*
 unfolding *exp-conj-def*
proof(*rule entailsI*)
 fix *s* **show** «*P*» *s* .& *Q s* ≤ *t R s*
 proof(*cases P s*)
  case *True*
  moreover from *uQ* **have** *0* ≤ *Q s* **by**(*auto*)
  ultimately show *?thesis* **by**(*simp add*:*pconj-lone wP*)
 next
  case *False*
  moreover from *uQ* **have** *Q s* ≤ *1* **by**(*auto*)
  ultimately show *?thesis* **using** *assms* **by** *auto*
 qed

qed

end

3.3 Induction

theory *Induction*
imports *Expectations Transformers*
begin

3.3.1 The Lattice of Expectations

Defining recursive (or iterative) programs requires us to reason about fixed points on the semantic objects, in this case expectations. The complication here, compared to the standard Knaster-Tarski theorem (for example, as shown in *HOL.Inductive*), is that we do not have a complete lattice.

Finding a lower bound is easy (it's λ-. *0*), but as we do not insist on any global bound on expectations (and work directly in HOL's real type, rather than extending it with a point at infinity), there is no top element. We solve the problem by defining the least (greatest) fixed point, restricted to an internally-bounded set, allowing us to substitute this bound for the top element. This works as long as the set contains at least one fixed point, which appears as an extra assumption in all the theorems.

This also works semantically, thanks to the definition of healthiness. Given a healthy transformer parameterised by some sound expectation: *t*. Imagine that we wish to find the least fixed point of *t P*. In practice, *t* is generally doubly healthy, that is $\forall P.\ sound\ P \longrightarrow healthy\ (t\ P)$ and $\forall Q.\ sound\ Q \longrightarrow healthy\ (\lambda P.\ t\ P\ Q)$. Thus by feasibility, *t P Q* must be bounded by *bound-of P*. Thus, as by definition $x \le t\ P\ x$ for any fixed point, all must lie in the set of sound expectations bounded above by λ-. *bound-of P*.

definition $\mathit{Inf\text{-}exp} :: {}'s\ expect\ set \Rightarrow {}'s\ expect$
where $\mathit{Inf\text{-}exp}\ S = (\lambda s.\ \mathit{Inf}\ \{f\ s\ |f.\ f \in S\})$

lemma *Inf-exp-lower*:
$[\![\, P \in S; \forall P{\in}S.\ nneg\ P\,]\!] \Longrightarrow \mathit{Inf\text{-}exp}\ S \le P$
unfolding *Inf-exp-def*
by(*intro le-funI cInf-lower bdd-belowI*[**where** *m=0*], *auto*)

lemma *Inf-exp-greatest*:
$[\![\, S \ne \{\}; \forall P{\in}S.\ Q \le P\,]\!] \Longrightarrow Q \le \mathit{Inf\text{-}exp}\ S$
unfolding *Inf-exp-def* **by**(*auto intro*!:*le-funI*[*OF cInf-greatest*])

definition $\mathit{Sup\text{-}exp} :: {}'s\ expect\ set \Rightarrow {}'s\ expect$
where $\mathit{Sup\text{-}exp}\ S = (\mathit{if}\ S = \{\}\ \mathit{then}\ \lambda s.\ 0\ \mathit{else}\ (\lambda s.\ \mathit{Sup}\ \{f\ s\ |f.\ f \in S\}))$

lemma *Sup-exp-upper*:
$[\![\, P \in S; \forall P{\in}S.\ bounded\text{-}by\ b\ P\,]\!] \Longrightarrow P \le \mathit{Sup\text{-}exp}\ S$

```
  unfolding Sup-exp-def
  by(cases S={}, simp-all, intro le-funI cSup-upper bdd-aboveI[where M=b], auto)

lemma Sup-exp-least:
  ⟦ ∀ P∈S. P ≤ Q; nneg Q ⟧ ⟹ Sup-exp S ≤ Q
  unfolding Sup-exp-def
  by(cases S={}, auto intro!:le-funI[OF cSup-least])

lemma Sup-exp-sound:
  assumes sS: ⋀P. P∈S ⟹ sound P
    and bS: ⋀P. P∈S ⟹ bounded-by b P
  shows sound (Sup-exp S)
proof(cases S={}, simp add:Sup-exp-def, blast,
    intro soundI2 bounded-byI2 nnegI2)
  assume neS: S ≠ {}
  then obtain P where Pin: P ∈ S by(auto)
  with sS bS have nP: nneg P bounded-by b P by(auto)
  hence nb: 0 ≤ b by(auto)

  from bS nb show Sup-exp S ⊩ λs. b
   by(auto intro:Sup-exp-least)

  from nP have λs. 0 ⊩ P by(auto)
  also from Pin bS have P ⊩ Sup-exp S
   by(auto intro:Sup-exp-upper)
  finally show λs. 0 ⊩ Sup-exp S .
qed

definition lfp-exp :: 's trans ⇒ 's expect
where lfp-exp t = Inf-exp {P. sound P ∧ t P ≤ P}

lemma lfp-exp-lowerbound:
  ⟦ t P ≤ P; sound P ⟧ ⟹ lfp-exp t ≤ P
  unfolding lfp-exp-def by(auto intro:Inf-exp-lower)

lemma lfp-exp-greatest:
  ⟦ ⋀P. ⟦ t P ≤ P; sound P ⟧ ⟹ Q ≤ P; sound Q; t R ⊩ R; sound R ⟧ ⟹ Q ≤ lfp-exp t
  unfolding lfp-exp-def by(auto intro:Inf-exp-greatest)

lemma feasible-lfp-exp-sound:
  feasible t ⟹ sound (lfp-exp t)
  by(intro soundI2 bounded-byI2 nnegI2, auto intro!:lfp-exp-lowerbound lfp-exp-greatest)

lemma lfp-exp-sound:
  assumes fR: t R ⊩ R and sR: sound R
  shows sound (lfp-exp t)
proof(intro soundI2)
  from fR sR have lfp-exp t ⊩ R
   by(auto intro:lfp-exp-lowerbound)
```

```
  also from sR have R ⊩ λs. bound-of R by(auto)
  finally show bounded-by (bound-of R) (lfp-exp t) by(auto)
  from fR sR show nneg (lfp-exp t) by(auto intro:lfp-exp-greatest)
qed

lemma lfp-exp-bound:
  (⋀P. unitary P ⟹ unitary (t P)) ⟹ bounded-by 1 (lfp-exp t)
  by(auto intro!:lfp-exp-lowerbound)

lemma lfp-exp-unitary:
  (⋀P. unitary P ⟹ unitary (t P)) ⟹ unitary (lfp-exp t)
proof(intro unitaryI[OF lfp-exp-sound lfp-exp-bound], simp-all)
  assume IH: ⋀P. unitary P ⟹ unitary (t P)
  have unitary (λs. 1) by(auto)
  with IH have unitary (t (λs. 1)) by(auto)
  thus t (λs. 1) ⊩ λs. 1 by(auto)
  show sound (λs. 1) by(auto)
qed

lemma lfp-exp-lemma2:
  fixes t::'s trans
  assumes st: ⋀P. sound P ⟹ sound (t P)
    and mt: mono-trans t
    and fR: t R ⊩ R and sR: sound R
  shows t (lfp-exp t) ≤ lfp-exp t
proof(rule lfp-exp-greatest[of t, OF - - fR sR])
  from fR sR show sound (t (lfp-exp t)) by(auto intro:lfp-exp-sound st)

  fix P::'s expect
  assume fP: t P ⊩ P and sP: sound P
  hence lfp-exp t ⊩ P by(rule lfp-exp-lowerbound)
  with fP sP have t (lfp-exp t) ⊩ t P by(auto intro:mono-transD[OF mt] lfp-exp-sound)
  also note fP
  finally show t (lfp-exp t) ⊩ P .
qed

lemma lfp-exp-lemma3:
  assumes st: ⋀P. sound P ⟹ sound (t P)
    and mt: mono-trans t
    and fR: t R ⊩ R and sR: sound R
  shows lfp-exp t ≤ t (lfp-exp t)
  by(iprover intro:lfp-exp-lowerbound lfp-exp-sound lfp-exp-lemma2 assms
              mono-transD[OF mt])

lemma lfp-exp-unfold:
  assumes nt: ⋀P. sound P ⟹ sound (t P)
    and mt: mono-trans t
    and fR: t R ⊩ R and sR: sound R
  shows lfp-exp t = t (lfp-exp t)
```

```
  by(iprover intro:antisym lfp-exp-lemma2 lfp-exp-lemma3 assms)

definition gfp-exp :: 's trans ⇒ 's expect
where gfp-exp t = Sup-exp {P. unitary P ∧ P ≤ t P}

lemma gfp-exp-upperbound:
  ⟦ P ≤ t P; unitary P ⟧ ⟹ P ≤ gfp-exp t
  by(auto simp:gfp-exp-def intro:Sup-exp-upper)

lemma gfp-exp-least:
  ⟦ ⋀P. ⟦ P ≤ t P; unitary P ⟧ ⟹ P ≤ Q; unitary Q ⟧ ⟹ gfp-exp t ≤ Q
  unfolding gfp-exp-def by(auto intro:Sup-exp-least)

lemma gfp-exp-bound:
  (⋀P. unitary P ⟹ unitary (t P)) ⟹ bounded-by 1 (gfp-exp t)
  unfolding gfp-exp-def
  by(rule bounded-byI2[OF Sup-exp-least], auto)

lemma gfp-exp-nneg[iff]:
  nneg (gfp-exp t)
proof(intro nnegI2, simp add:gfp-exp-def, cases)
  assume empty: {P. unitary P ∧ P ⊩ t P} = {}
  show λs. 0 ⊩ Sup-exp {P. unitary P ∧ P ⊩ t P}
    by(simp only:empty Sup-exp-def, auto)
next
  assume {P. unitary P ∧ P ⊩ t P} ≠ {}
  then obtain Q where Qin: Q ∈ {P. unitary P ∧ P ⊩ t P} by(auto)
  hence λs. 0 ⊩ Q by(auto)
  also from Qin have Q ⊩ Sup-exp {P. unitary P ∧ P ⊩ t P}
    by(auto intro:Sup-exp-upper)
  finally show λs. 0 ⊩ Sup-exp {P. unitary P ∧ P ⊩ t P} .
qed

lemma gfp-exp-unitary:
  (⋀P. unitary P ⟹ unitary (t P)) ⟹ unitary (gfp-exp t)
  by(iprover intro:gfp-exp-nneg gfp-exp-bound unitaryI2)

lemma gfp-exp-lemma2:
  assumes ft: ⋀P. unitary P ⟹ unitary (t P)
    and mt: ⋀P Q. ⟦ unitary P; unitary Q; P ⊩ Q ⟧ ⟹ t P ⊩ t Q
  shows gfp-exp t ≤ t (gfp-exp t)
proof(rule gfp-exp-least)
  show unitary (t (gfp-exp t)) by(auto intro:gfp-exp-unitary ft)
  fix P
  assume fp: P ≤ t P and uP: unitary P
  with ft have P ≤ gfp-exp t by(auto intro:gfp-exp-upperbound)
  with uP gfp-exp-unitary ft
  have t P ≤ t (gfp-exp t) by(blast intro: mt)
  with fp show P ≤ t (gfp-exp t) by(auto)
```

qed

lemma *gfp-exp-lemma3*:
  assumes *ft*: $\bigwedge P.\ unitary\ P \Longrightarrow unitary\ (t\ P)$
    and *mt*: $\bigwedge P\ Q.\ [\![\ unitary\ P;\ unitary\ Q;\ P \Vdash Q\ ]\!] \Longrightarrow t\ P \Vdash t\ Q$
  shows $t\ (gfp\text{-}exp\ t) \leq gfp\text{-}exp\ t$
  by(*iprover intro*:*gfp-exp-upperbound unitary-sound*
      gfp-exp-unitary gfp-exp-lemma2 assms)

lemma *gfp-exp-unfold*:
  $(\bigwedge P.\ unitary\ P \Longrightarrow unitary\ (t\ P)) \Longrightarrow (\bigwedge P\ Q.\ [\![\ unitary\ P;\ unitary\ Q;\ P \Vdash Q\ ]\!] \Longrightarrow t\ P \Vdash t\ Q) \Longrightarrow$
  $gfp\text{-}exp\ t = t\ (gfp\text{-}exp\ t)$
  by(*iprover intro*:*antisym gfp-exp-lemma2 gfp-exp-lemma3*)

3.3.2 The Lattice of Transformers

In addition to fixed points on expectations, we also need to reason about fixed points on expectation transformers. The interpretation of a recursive program in pGCL is as a fixed point of a function from transformers to transformers. In contrast to the case of expectations, *healthy* transformers do form a complete lattice, where the bottom element is λ- -. *0*, and the top element is the greatest allowed by feasibility: λP -. *bound-of P*.

definition *Inf-trans* :: *'s trans set* $\Rightarrow$ *'s trans*
where $Inf\text{-}trans\ S = (\lambda P.\ Inf\text{-}exp\ \{t\ P\ |t.\ t \in S\})$

lemma *Inf-trans-lower*:
  $[\![\ t \in S;\ \forall u{\in}S.\ \forall P.\ sound\ P \longrightarrow sound\ (u\ P)\ ]\!] \Longrightarrow le\text{-}trans\ (Inf\text{-}trans\ S)\ t$
  unfolding *Inf-trans-def*
  by(*rule le-transI*[*OF Inf-exp-lower*], *blast*+)

lemma *Inf-trans-greatest*:
  $[\![\ S \neq \{\};\ \forall t{\in}S.\ \forall P.\ le\text{-}trans\ u\ t\ ]\!] \Longrightarrow le\text{-}trans\ u\ (Inf\text{-}trans\ S)$
  unfolding *Inf-trans-def* **by**(*auto intro*!:*le-transI*[*OF Inf-exp-greatest*])

definition *Sup-trans* :: *'s trans set* $\Rightarrow$ *'s trans*
where $Sup\text{-}trans\ S = (\lambda P.\ Sup\text{-}exp\ \{t\ P\ |t.\ t \in S\})$

lemma *Sup-trans-upper*:
  $[\![\ t \in S;\ \forall u{\in}S.\ \forall P.\ unitary\ P \longrightarrow unitary\ (u\ P)\ ]\!] \Longrightarrow le\text{-}utrans\ t\ (Sup\text{-}trans\ S)$
  unfolding *Sup-trans-def*
  by(*intro le-utransI*[*OF Sup-exp-upper*], *auto intro*:*unitary-bound*)

lemma *Sup-trans-upper2*:
  $[\![\ t \in S;\ \forall u{\in}S.\ \forall P.\ (nneg\ P \wedge bounded\text{-}by\ b\ P) \longrightarrow (nneg\ (u\ P) \wedge bounded\text{-}by\ b\ (u\ P));$
    $nneg\ P;\ bounded\text{-}by\ b\ P\ ]\!] \Longrightarrow t\ P \Vdash Sup\text{-}trans\ S\ P$
  unfolding *Sup-trans-def* **by**(*blast intro*:*Sup-exp-upper*)

```
lemma Sup-trans-least:
  ⟦ ∀t∈S. le-utrans t u; ⋀P. unitary P ⟹ unitary (u P) ⟧ ⟹ le-utrans (Sup-trans S) u
  unfolding Sup-trans-def
  by(auto intro!:sound-nneg[OF unitary-sound] le-utransI[OF Sup-exp-least])

lemma Sup-trans-least2:
  ⟦ ∀t∈S. ∀P. nneg P ⟶ bounded-by b P ⟶ t P ⊩ u P;
    ∀u∈S. ∀P. (nneg P ∧ bounded-by b P) ⟶ (nneg (u P) ∧ bounded-by b (u P));
      nneg P; bounded-by b P; ⋀P. ⟦ nneg P; bounded-by b P ⟧ ⟹ nneg (u P) ⟧ ⟹
Sup-trans S P ⊩ u P
   unfolding Sup-trans-def by(blast intro!:Sup-exp-least)

lemma feasible-Sup-trans:
  fixes S::'s trans set
  assumes fS: ∀t∈S. feasible t
  shows feasible (Sup-trans S)
proof(cases S={}, simp add:Sup-trans-def Sup-exp-def, blast, intro feasibleI)
  fix b::real and P::'s expect
  assume bP: bounded-by b P and nP: nneg P
    and neS: S ≠ {}

  from neS obtain t where tin: t ∈ S by(auto)
  with fS have ft: feasible t by(auto)
  with bP nP have λs. 0 ⊩ t P by(auto)
  also {
    from bP nP have sound P by(auto)
    with tin fS have t P ⊩ Sup-trans S P
      by(auto intro!:Sup-trans-upper2)
  }
  finally show nneg (Sup-trans S P) by(auto)

  from fS bP nP
  show bounded-by b (Sup-trans S P)
    by(auto intro!:bounded-byI2[OF Sup-trans-least2])
qed

definition lfp-trans :: ('s trans ⇒ 's trans) ⇒ 's trans
where lfp-trans T = Inf-trans {t. (∀P. sound P ⟶ sound (t P)) ∧ le-trans (T t) t}

lemma lfp-trans-lowerbound:
  ⟦ le-trans (T t) t; ⋀P. sound P ⟹ sound (t P) ⟧ ⟹ le-trans (lfp-trans T) t
  unfolding lfp-trans-def
  by(auto intro:Inf-trans-lower)

lemma lfp-trans-greatest:
  ⟦ ⋀t P. ⟦ le-trans (T t) t; ⋀P. sound P ⟹ sound (t P) ⟧ ⟹ le-trans u t;
    ⋀P. sound P ⟹ sound (v P); le-trans (T v) v ⟧ ⟹
   le-trans u (lfp-trans T)
  unfolding lfp-trans-def by(rule Inf-trans-greatest, auto)
```

```
lemma lfp-trans-sound:
  fixes P Q::'s expect
  assumes sP: sound P
    and fv: le-trans (T v) v
    and sv: ⋀P. sound P ⟹ sound (v P)
  shows sound (lfp-trans T P)
proof(intro soundI2 bounded-byI2 nnegI2)
  from fv sv have le-trans (lfp-trans T) v
    by(iprover intro:lfp-trans-lowerbound)
  with sP have lfp-trans T P ⊩ v P by(auto)
  also {
    from sv sP have sound (v P) by(iprover)
    hence v P ⊩ λs. bound-of (v P) by(auto)
  }
  finally show lfp-trans T P ⊩ λs. bound-of (v P) .

  have le-trans (λP s. 0) (lfp-trans T)
  proof(intro lfp-trans-greatest)
    fix t::'s trans
    assume ⋀P. sound P ⟹ sound (t P)
    hence ⋀P. sound P ⟹ λs. 0 ⊩ t P by(auto)
    thus le-trans (λP s. 0) t by(auto)
  next
    fix P::'s expect
    assume sound P thus sound (v P) by(rule sv)
  next
    show le-trans (T v) v by(rule fv)
  qed
  with sP show λs. 0 ⊩ lfp-trans T P by(auto)
qed

lemma lfp-trans-unitary:
  fixes P Q::'s expect
  assumes uP: unitary P
    and fv: le-trans (T v) v
    and sv: ⋀P. sound P ⟹ sound (v P)
    and fT: le-trans (T (λP s. bound-of P)) (λP s. bound-of P)
  shows unitary (lfp-trans T P)
proof(rule unitaryI)
  from unitary-sound[OF uP] fv sv show sound (lfp-trans T P)
    by(rule lfp-trans-sound)

  show bounded-by 1 (lfp-trans T P)
  proof(rule bounded-byI2)
    from fT have le-trans (lfp-trans T) (λP s. bound-of P)
      by(auto intro: lfp-trans-lowerbound)
    with uP have lfp-trans T P ⊩ λs. bound-of P by(auto)
    also from uP have ... ⊩ λs. 1 by(auto)
```

finally show *lfp-trans T P* ⊩ *λs. 1* **.**
qed
qed

lemma *lfp-trans-lemma2*:
fixes *v*::*'s trans*
assumes *mono*: ⋀*t u*. ⟦ *le-trans t u*; ⋀*P*. *sound P* ⟹ *sound* (*t P*);
⋀*P*. *sound P* ⟹ *sound* (*u P*) ⟧ ⟹ *le-trans* (*T t*) (*T u*)
and *nT*: ⋀*t P*. ⟦ ⋀*Q*. *sound Q* ⟹ *sound* (*t Q*); *sound P* ⟧ ⟹ *sound* (*T t P*)
and *fv*: *le-trans* (*T v*) *v*
and *sv*: ⋀*P*. *sound P* ⟹ *sound* (*v P*)
shows *le-trans* (*T* (*lfp-trans T*)) (*lfp-trans T*)
proof(*rule lfp-trans-greatest*[**where** *T*=*T* **and** *v*=*v*], *simp-all add*:*assms*)
fix *t*::*'s trans* **and** *P*::*'s expect*
assume *ft*: *le-trans* (*T t*) *t* **and** *st*: ⋀*P*. *sound P* ⟹ *sound* (*t P*)
hence *le-trans* (*lfp-trans T*) *t* **by**(*auto intro*!:*lfp-trans-lowerbound*)
with *ft st* **have** *le-trans* (*T* (*lfp-trans T*)) (*T t*)
by(*iprover intro*:*mono lfp-trans-sound fv sv*)
also note *ft*
finally show *le-trans* (*T* (*lfp-trans T*)) *t* **.**
qed

lemma *lfp-trans-lemma3*:
fixes *v*::*'s trans*
assumes *mono*: ⋀*t u*. ⟦ *le-trans t u*; ⋀*P*. *sound P* ⟹ *sound* (*t P*);
⋀*P*. *sound P* ⟹ *sound* (*u P*) ⟧ ⟹ *le-trans* (*T t*) (*T u*)
and *sT*: ⋀*t P*. ⟦ ⋀*Q*. *sound Q* ⟹ *sound* (*t Q*); *sound P* ⟧ ⟹ *sound* (*T t P*)
and *fv*: *le-trans* (*T v*) *v*
and *sv*: ⋀*P*. *sound P* ⟹ *sound* (*v P*)
shows *le-trans* (*lfp-trans T*) (*T* (*lfp-trans T*))
proof(*rule lfp-trans-lowerbound*)
fix *P*::*'s expect*
assume *sP*: *sound P*
have *n1*: ⋀*P*. *sound P* ⟹ *sound* (*lfp-trans T P*)
by(*iprover intro*:*lfp-trans-sound fv sv*)
with *sP* **have** *n2*: *sound* (*lfp-trans T P*)
by(*iprover intro*:*lfp-trans-sound fv sv sT*)
with *n1 sP* **show** *n3*: *sound* (*T* (*lfp-trans T*) *P*)
by(*iprover intro*: *sT*)
next
show *le-trans* (*T* (*T* (*lfp-trans T*))) (*T* (*lfp-trans T*))
by(*rule mono*[*OF lfp-trans-lemma2*, *OF mono*],
(*iprover intro*:*assms lfp-trans-sound*)+)
qed

lemma *lfp-trans-unfold*:
fixes *P*::*'s expect*
assumes *mono*: ⋀*t u*. ⟦ *le-trans t u*; ⋀*P*. *sound P* ⟹ *sound* (*t P*);
⋀*P*. *sound P* ⟹ *sound* (*u P*) ⟧ ⟹ *le-trans* (*T t*) (*T u*)

and *sT*: $\bigwedge t\ P.\ [\![\ \bigwedge Q.\ sound\ Q \Longrightarrow sound\ (t\ Q);\ sound\ P\ ]\!] \Longrightarrow sound\ (T\ t\ P)$
and *fv*: *le-trans* (*T v*) *v*
and *sv*: $\bigwedge P.\ sound\ P \Longrightarrow sound\ (v\ P)$
shows *equiv-trans* (*lfp-trans T*) (*T* (*lfp-trans T*))
by(*rule le-trans-antisym*,
rule lfp-trans-lemma3[*OF mono*], (*iprover intro*:*assms*)+,
rule lfp-trans-lemma2[*OF mono*], (*iprover intro*:*assms*)+)

definition *gfp-trans* :: ($'s\ trans \Rightarrow {}'s\ trans) \Rightarrow {}'s\ trans$
where *gfp-trans T* = *Sup-trans* $\{t.\ (\forall P.\ unitary\ P \longrightarrow unitary\ (t\ P)) \wedge le\text{-}utrans\ t\ (T\ t)\}$

lemma *gfp-trans-upperbound*:
$[\![\ le\text{-}utrans\ t\ (T\ t);\ \bigwedge P.\ unitary\ P \Longrightarrow unitary\ (t\ P)\ ]\!] \Longrightarrow le\text{-}utrans\ t\ (gfp\text{-}trans\ T)$
unfolding *gfp-trans-def* **by**(*auto intro*:*Sup-trans-upper*)

lemma *gfp-trans-least*:
$[\![\ \bigwedge t.\ [\![\ le\text{-}utrans\ t\ (T\ t);\ \bigwedge P.\ unitary\ P \Longrightarrow unitary\ (t\ P)\ ]\!] \Longrightarrow le\text{-}utrans\ t\ u;$
$\bigwedge P.\ unitary\ P \Longrightarrow unitary\ (u\ P)\ ]\!] \Longrightarrow$
le-utrans (*gfp-trans T*) *u*
unfolding *gfp-trans-def* **by**(*auto intro*:*Sup-trans-least*)

lemma *gfp-trans-unitary*:
fixes *P*::*'s expect*
assumes *uP*: *unitary P*
shows *unitary* (*gfp-trans T P*)
proof(*intro unitaryI2 nnegI2 bounded-byI2*)
show *gfp-trans T P* $\Vdash \lambda s.\ 1$
unfolding *gfp-trans-def Sup-trans-def*
proof(*rule Sup-exp-least*, *clarify*)
fix *t*::*'s trans*
assume $\forall P.\ unitary\ P \longrightarrow unitary\ (t\ P)$
with *uP* **have** *unitary* (*t P*) **by**(*auto*)
thus *t P* $\Vdash \lambda s.\ 1$ **by**(*auto*)
next
show *nneg* ($\lambda s.\ 1$::*real*) **by**(*auto*)
qed
let $?S = \{t\ P\ |t.\ t \in \{t.\ (\forall P.\ unitary\ P \longrightarrow unitary\ (t\ P)) \wedge le\text{-}utrans\ t\ (T\ t)\}\}$
show $\lambda s.\ 0 \Vdash$ *gfp-trans T P*
unfolding *gfp-trans-def Sup-trans-def*
proof(*cases*)
assume *empty*: $?S = \{\}$
show $\lambda s.\ 0 \Vdash$ *Sup-exp ?S*
by(*simp only*:*empty Sup-exp-def*, *auto*)
next
assume $?S \neq \{\}$
then obtain *Q* **where** *Qin*: $Q \in ?S$ **by**(*auto*)
with *uP* **have** *unitary Q* **by**(*auto*)
hence $\lambda s.\ 0 \Vdash Q$ **by**(*auto*)
also with *uP Qin* **have** $Q \Vdash$ *Sup-exp ?S*

proof(*intro Sup-exp-upper*, *blast*, *clarify*)
fix *t*::*'s trans*
assume ∀ *Q*. *unitary Q* ⟶ *unitary* (*t Q*)
with *uP* **show** *bounded-by 1* (*t P*) **by**(*auto*)
qed
finally show λ*s*. *0* ⊩ *Sup-exp ?S* **.**
qed
qed

lemma *gfp-trans-lemma2*:
assumes *mono*: ⋀*t u*. ⟦ *le-utrans t u*; ⋀*P*. *unitary P* ⟹ *unitary* (*t P*);
⋀*P*. *unitary P* ⟹ *unitary* (*u P*) ⟧ ⟹ *le-utrans* (*T t*) (*T u*)
and *hT*: ⋀*t P*. ⟦ ⋀*Q*. *unitary Q* ⟹ *unitary* (*t Q*); *unitary P* ⟧ ⟹ *unitary* (*T t P*)
shows *le-utrans* (*gfp-trans T*) (*T* (*gfp-trans T*))
proof(*rule gfp-trans-least*, *simp-all add*:*hT gfp-trans-unitary*)
fix *t*
assume *fp*: *le-utrans t* (*T t*) **and** *ht*: ⋀*P*. *unitary P* ⟹ *unitary* (*t P*)

note *fp*
also {
from *fp ht* **have** *le-utrans t* (*gfp-trans T*)**by**(*rule gfp-trans-upperbound*)
moreover note *ht gfp-trans-unitary*
ultimately have *le-utrans* (*T t*) (*T* (*gfp-trans T*)) **by**(*rule mono*)
}
finally show *le-utrans t* (*T* (*gfp-trans T*)) **.**
qed

lemma *gfp-trans-lemma3*:
assumes *mono*: ⋀*t u*. ⟦ *le-utrans t u*; ⋀*P*. *unitary P* ⟹ *unitary* (*t P*);
⋀*P*. *unitary P* ⟹ *unitary* (*u P*) ⟧ ⟹ *le-utrans* (*T t*) (*T u*)
and *hT*: ⋀*t P*. ⟦ ⋀*Q*. *unitary Q* ⟹ *unitary* (*t Q*); *unitary P* ⟧ ⟹ *unitary* (*T t P*)
shows *le-utrans* (*T* (*gfp-trans T*)) (*gfp-trans T*)
by(*blast intro*!:*mono gfp-trans-unitary gfp-trans-upperbound gfp-trans-lemma2 mono hT*)

lemma *gfp-trans-unfold*:
assumes *mono*: ⋀*t u*. ⟦ *le-utrans t u*; ⋀*P*. *unitary P* ⟹ *unitary* (*t P*);
⋀*P*. *unitary P* ⟹ *unitary* (*u P*) ⟧ ⟹ *le-utrans* (*T t*) (*T u*)
and *hT*: ⋀*t P*. ⟦ ⋀*Q*. *unitary Q* ⟹ *unitary* (*t Q*); *unitary P* ⟧ ⟹ *unitary* (*T t P*)
shows *equiv-utrans* (*gfp-trans T*) (*T* (*gfp-trans T*))
using *assms* **by**(*auto intro*!: *le-utrans-antisym gfp-trans-lemma2 gfp-trans-lemma3*)

3.3.3 Tail Recursion

The least (greatest) fixed point of a tail-recursive expression on transformers is equivalent (given appropriate side conditions) to the least (greatest) fixed point on expectations.

lemma *gfp-pulldown*:

```
  fixes P::'s expect
  assumes tailcall:  ⋀u P. unitary P ⟹ T u P = t P (u P)
    and fT:      ⋀t P. ⟦ ⋀Q. unitary Q ⟹ unitary (t Q); unitary P ⟧ ⟹ unitary (T t P)
    and ft:      ⋀P Q. unitary P ⟹ unitary Q ⟹ unitary (t P Q)
    and mt:      ⋀P Q R. ⟦ unitary P; unitary Q; unitary R; Q ⊩ R ⟧ ⟹ t P Q ⊩ t P R
    and uP:      unitary P
    and monoT:   ⋀t u. ⟦ le-utrans t u; ⋀P. unitary P ⟹ unitary (t P);
                 ⋀P. unitary P ⟹ unitary (u P) ⟧ ⟹ le-utrans (T t) (T u)
  shows gfp-trans T P = gfp-exp (t P) (is ?X P = ?Y P)
proof(rule antisym)
  show ?X P ≤ ?Y P
  proof(rule gfp-exp-upperbound)
    from monoT fT uP have (gfp-trans T) P ≤ (T (gfp-trans T)) P
      by(auto intro!: le-utransD[OF gfp-trans-lemma2])
    also from uP have (T (gfp-trans T)) P = t P (gfp-trans T P) by(rule tailcall)
    finally show gfp-trans T P ⊩ t P (gfp-trans T P) .
    from uP gfp-trans-unitary show unitary (gfp-trans T P) by(auto)
  qed
  show ?Y P ≤ ?X P
  proof(rule le-utransD[OF gfp-trans-upperbound], simp-all add:assms)
    show le-utrans (λa. gfp-exp (t a)) (T (λa. gfp-exp (t a)))
    proof(rule le-utransI)
      fix Q::'s expect assume uQ: unitary Q
      with ft have ⋀R. unitary R ⟹ unitary (t Q R) by(auto)
      with mt[OF uQ] have gfp-exp (t Q) = t Q (gfp-exp (t Q)) by(blast intro:gfp-exp-unfold)
      also from uQ have ... = T (λa. gfp-exp (t a)) Q by(rule tailcall[symmetric])
      finally show gfp-exp (t Q) ≤ T (λa. gfp-exp (t a)) Q by(simp)
    qed
    fix Q::'s expect assume unitary Q
    with ft have ⋀R. unitary R ⟹ unitary (t Q R) by(auto)
    thus unitary (gfp-exp (t Q)) by(rule gfp-exp-unitary)
  qed
qed

lemma lfp-pulldown:
  fixes P::'s expect and t::'s expect ⇒ 's trans
    and T::'s trans ⇒ 's trans
  assumes tailcall:  ⋀u P. sound P ⟹ T u P = t P (u P)
    and st:      ⋀P Q. sound P ⟹ sound Q ⟹ sound (t P Q)
    and mt:      ⋀P. sound P ⟹ mono-trans (t P)
    and monoT: ⋀t u. ⟦ le-trans t u; ⋀P. sound P ⟹ sound (t P);
               ⋀P. sound P ⟹ sound (u P) ⟧ ⟹ le-trans (T t) (T u)
    and nT:   ⋀t P. ⟦ ⋀Q. sound Q ⟹ sound (t Q); sound P ⟧ ⟹ sound (T t P)
    and fv:   le-trans (T v) v
    and sv:   ⋀P. sound P ⟹ sound (v P)
    and sP:   sound P
  shows lfp-trans T P = lfp-exp (t P) (is ?X P = ?Y P)
proof(rule antisym)
  show ?Y P ≤ ?X P
```

```
  proof(rule lfp-exp-lowerbound)
    from sP have t P (lfp-trans T P) = (T (lfp-trans T)) P by(rule tailcall[symmetric])
    also have (T (lfp-trans T)) P ≤ (lfp-trans T) P
      by(rule le-transD[OF lfp-trans-lemma2[OF monoT]], (iprover intro:assms)+)
    finally show t P (lfp-trans T P) ≤ lfp-trans T P .
    from sP show sound (lfp-trans T P)
      by(iprover intro:lfp-trans-sound assms)
  qed

  have ⋀P. sound P ⟹ t P (v P) = T v P by(simp add:tailcall)
  also have ⋀P. sound P ⟹ ... P ⊩ v P by(auto intro:le-transD[OF fv])
  finally have fvP: ⋀P. sound P ⟹ t P (v P) ⊩ v P .
  have svP: ⋀P. sound P ⟹ sound (v P) by(rule sv)

  show ?X P ≤ ?Y P
  proof(rule le-transD[OF lfp-trans-lowerbound, OF - - sP])
    show le-trans (T (λa. lfp-exp (t a))) (λa. lfp-exp (t a))
    proof(rule le-transI)
      fix P::'s expect
      assume sP: sound P

      from sP have T (λa. lfp-exp (t a)) P = t P (lfp-exp (t P)) by(rule tailcall)
      also have t P (lfp-exp (t P)) = lfp-exp (t P)
        by(iprover intro: lfp-exp-unfold[symmetric] sP st mt fvP svP)
      finally show T (λa. lfp-exp (t a)) P ⊩ lfp-exp (t P) by(simp)
    qed
    fix P::'s expect
    assume sound P
    with fvP svP show sound (lfp-exp (t P))
      by(blast intro:lfp-exp-sound)
  qed
qed

definition Inf-utrans :: 's trans set ⇒ 's trans
where Inf-utrans S = (if S = {} then λP s. 1 else Inf-trans S)

lemma Inf-utrans-lower:
  ⟦ t ∈ S; ∀ t∈S. ∀ P. unitary P ⟶ unitary (t P) ⟧ ⟹ le-utrans (Inf-utrans S) t
  unfolding Inf-utrans-def
  by(cases S={},
     auto intro!:le-utransI Inf-exp-lower sound-nneg unitary-sound
          simp:Inf-trans-def)

lemma Inf-utrans-greatest:
  ⟦ ⋀P. unitary P ⟹ unitary (t P); ∀ u∈S. le-utrans t u ⟧ ⟹ le-utrans t (Inf-utrans S)
  unfolding Inf-utrans-def Inf-trans-def
  by(cases S={}, simp-all, (blast intro!:le-utransI Inf-exp-greatest)+)

end
```

Chapter 4

The pGCL Language

4.1 A Shallow Embedding of pGCL in HOL

theory *Embedding* **imports** *Misc Induction* **begin**

4.1.1 Core Primitives and Syntax

A pGCL program is embedded directly as its strict or liberal transformer. This is achieved with an additional parameter, specifying which semantics should be obeyed.

type-synonym $'s\ prog = bool \Rightarrow ('s \Rightarrow real) \Rightarrow ('s \Rightarrow real)$

Abort either always fails, $\lambda P\ s.\ 0$, or always succeeds, $\lambda P\ s.\ 1$.

definition $Abort :: 's\ prog$
where $Abort \equiv \lambda ab\ P\ s.\ if\ ab\ then\ 0\ else\ 1$

Skip does nothing at all.

definition $Skip :: 's\ prog$
where $Skip \equiv \lambda ab\ P.\ P$

Apply lifts a state transformer into the space of programs.

definition $Apply :: ('s \Rightarrow 's) \Rightarrow 's\ prog$
where $Apply\ f \equiv \lambda ab\ P\ s.\ P\ (f\ s)$

Seq is sequential composition.

definition $Seq :: 's\ prog \Rightarrow 's\ prog \Rightarrow 's\ prog$
(**infixl** ‹;;› *59*)
where $Seq\ a\ b \equiv (\lambda ab.\ a\ ab\ o\ b\ ab)$

PC is *probabilistic* choice between programs.

definition $PC :: 's\ prog \Rightarrow ('s \Rightarrow real) \Rightarrow 's\ prog \Rightarrow 's\ prog$
(‹- _⊕ -› [*58*,*57*,*57*] *57*)

where *PC a P b* ≡ $\lambda ab\ Q\ s.\ P\ s * a\ ab\ Q\ s + (1 - P\ s) * b\ ab\ Q\ s$

DC is *demonic* choice between programs.

definition *DC* :: *'s prog* ⇒ *'s prog* ⇒ *'s prog* (‹- ⊓ -› [58,57] 57)
where *DC a b* ≡ $\lambda ab\ Q\ s.$ *min* (*a ab Q s*) (*b ab Q s*)

AC is *angelic* choice between programs.

definition *AC* :: *'s prog* ⇒ *'s prog* ⇒ *'s prog* (‹- ⊔ -› [58,57] 57)
where *AC a b* ≡ $\lambda ab\ Q\ s.$ *max* (*a ab Q s*) (*b ab Q s*)

Embed allows any expectation transformer to be treated syntactically as a program, by ignoring the failure flag.

definition *Embed* :: *'s trans* ⇒ *'s prog*
where *Embed t* = ($\lambda ab.\ t$)

Mu is the recursive primitive, and is either then least or greatest fixed point.

definition *Mu* :: (*'s prog* ⇒ *'s prog*) ⇒ *'s prog* (**binder** ‹μ› *50*)
where *Mu*(*T*) ≡ ($\lambda ab.$ *if ab then lfp-trans* ($\lambda t.$ *T* (*Embed t*) *ab*)
else gfp-trans ($\lambda t.$ *T* (*Embed t*) *ab*))

repeat expresses finite repetition

primrec
repeat :: *nat* ⇒ *'a prog* ⇒ *'a prog*
where
repeat 0 p = *Skip* |
repeat (*Suc n*) *p* = *p* ;; *repeat n p*

SetDC is demonic choice between a set of alternatives, which may depend on the state.

definition *SetDC* :: (*'a* ⇒ *'s prog*) ⇒ (*'s* ⇒ *'a set*) ⇒ *'s prog*
where *SetDC f S* ≡ $\lambda ab\ P\ s.$ *Inf* (($\lambda a.$ *f a ab P s*) ‘ *S s*)

syntax *-SetDC* :: *pttrn* => (*'s* => *'a set*) => *'s prog* => *'s prog*
(‹⊓-∈-./ -› *100*)
syntax-consts *-SetDC* == *SetDC*
translations ⊓*x*∈*S*. *p* == *CONST SetDC* (%*x*. *p*) *S*

The above syntax allows us to write ⊓*x*∈*S*. *Apply f*

SetPC is *probabilistic* choice from a set. Note that this is only meaningful for distributions of finite support.

definition
SetPC :: (*'a* ⇒ *'s prog*) ⇒ (*'s* ⇒ *'a* ⇒ *real*) ⇒ *'s prog*
where
SetPC f p ≡ $\lambda ab\ P\ s.\ \sum a \in supp\ (p\ s).\ p\ s\ a * f\ a\ ab\ P\ s$

Bind allows us to name an expression in the current state, and re-use it later.

definition
Bind :: ('s ⇒ 'a) ⇒ ('a ⇒ 's prog) ⇒ 's prog
where
Bind g f ab ≡ λP s. let a = g s in f a ab P s

This gives us something like let syntax

syntax *-Bind* :: *pttrn* => ('s => 'a) => 's prog => 's prog
(‹- is - in -› [55,55,55]55)
syntax-consts *-Bind* == *Bind*
translations *x is f in a* => *CONST Bind f* (%x. a)

definition *flip* :: ('a ⇒ 'b ⇒ 'c) ⇒ 'b ⇒ 'a ⇒ 'c
where [*simp*]: *flip f* = (λb a. f a b)

The following pair of translations introduce let-style syntax for *SetPC* and *SetDC*, respectively.

syntax *-PBind* :: *pttrn* => ('s => real) => 's prog => 's prog
(‹bind - at - in -› [55,55,55]55)
syntax-consts *-PBind* == *SetPC*
translations *bind x at p in a* => *CONST SetPC* (%x. a) (*CONST flip* (%x. p))

syntax *-DBind* :: *pttrn* => ('s => 'a set) ⇒ 's prog => 's prog
(‹bind - from - in -› [55,55,55]55)
syntax-consts *-DBind* == *SetDC*
translations *bind x from S in a* => *CONST SetDC* (%x. a) *S*

The following syntax translations are for convenience when using a record as the state type.

syntax
-assign :: *ident* => 'a => 's prog (‹- := -› [1000,900]900)
ML ‹
fun assign-tr - [Const (name,-), arg] =
Const (Embedding.Apply, dummyT) $
Abs (s, dummyT,
Syntax.const (suffix Record.updateN name) $
Abs (Name.uu-, dummyT, arg $ Bound 1) $ Bound 0)
| assign-tr - ts = raise TERM (assign-tr, ts)
›
parse-translation ‹[(@{*syntax-const -assign*}, *assign-tr*)]›

syntax
-SetPC :: *ident* => ('s => 'a => real) => 's prog
(‹choose - at -› [66,66]66)
syntax-consts
-SetPC ⇌ *SetPC*
ML ‹
fun set-pc-tr - [Const (f,-), P] =
Const (SetPC, dummyT) $

```
    Abs (v, dummyT,
        (Const (Embedding.Apply, dummyT) $
         Abs (s, dummyT,
             Syntax.const (suffix Record.updateN f) $
             Abs (Name.uu-, dummyT, Bound 2) $ Bound 0))) $
    P
  | set-pc-tr - ts = raise TERM (set-pc-tr, ts)
›
parse-translation ‹[(@{syntax-const -SetPC}, set-pc-tr)]›
```

```
syntax
  -set-dc :: ident => ('s => 'a set) => 's prog (‹- :∈ -› [66,66]66)
syntax-consts
  -set-dc ⇌ SetDC
ML ‹
 fun set-dc-tr - [Const (f,-), S] =
    Const (SetDC, dummyT) $
    Abs (v, dummyT,
        (Const (Embedding.Apply, dummyT) $
         Abs (s, dummyT,
             Syntax.const (suffix Record.updateN f) $
             Abs (Name.uu-, dummyT, Bound 2) $ Bound 0))) $
    S
  | set-dc-tr - ts = raise TERM (set-dc-tr, ts)
›
parse-translation ‹[(@{syntax-const -set-dc}, set-dc-tr)]›
```

These definitions instantiate the embedding as either weakest precondition (True) or weakest liberal precondition (False).

```
syntax
  -set-dc-UNIV :: ident => 's prog (‹any -› [66]66)
syntax-consts
  -set-dc-UNIV == SetDC
translations
  -set-dc-UNIV x => -set-dc x (%-. CONST UNIV)
```

```
definition
  wp :: 's prog ⇒ 's trans
where
  wp pr ≡ pr True
```

```
definition
  wlp :: 's prog ⇒ 's trans
where
  wlp pr ≡ pr False
```

If-Then-Else as a degenerate probabilistic choice.

```
abbreviation(input)
  if-then-else :: ['s ⇒ bool, 's prog, 's prog] ⇒ 's prog
```

(‹*If* - *Then* - *Else* -› *58*)
where
If P Then a Else b == $a\ _{«P»}\oplus\ b$

Syntax for loops

abbreviation
do-while :: $['s \Rightarrow bool, 's\ prog] \Rightarrow 's\ prog$
(‹*do* - ⟶// (*4* -) //*od*›)
where
do-while P a ≡ μ *x*. *If P Then a* ;; *x Else Skip*

4.1.2 Unfolding rules for non-recursive primitives

lemma *eval-wp-Abort*:
wp Abort P = (λ*s*. *0*)
unfolding *wp-def Abort-def* **by**(*simp*)

lemma *eval-wlp-Abort*:
wlp Abort P = (λ*s*. *1*)
unfolding *wlp-def Abort-def* **by**(*simp*)

lemma *eval-wp-Skip*:
wp Skip P = *P*
unfolding *wp-def Skip-def* **by**(*simp*)

lemma *eval-wlp-Skip*:
wlp Skip P = *P*
unfolding *wlp-def Skip-def* **by**(*simp*)

lemma *eval-wp-Apply*:
wp (*Apply f*) *P* = *P o f*
unfolding *wp-def Apply-def* **by**(*simp add*:*o-def*)

lemma *eval-wlp-Apply*:
wlp (*Apply f*) *P* = *P o f*
unfolding *wlp-def Apply-def* **by**(*simp add*:*o-def*)

lemma *eval-wp-Seq*:
wp (*a* ;; *b*) *P* = (*wp a o wp b*) *P*
unfolding *wp-def Seq-def* **by**(*simp*)

lemma *eval-wlp-Seq*:
wlp (*a* ;; *b*) *P* = (*wlp a o wlp b*) *P*
unfolding *wlp-def Seq-def* **by**(*simp*)

lemma *eval-wp-PC*:
wp ($a\ _{Q}\oplus\ b$) *P* = (λ*s*. *Q s* ∗ *wp a P s* + (*1* − *Q s*) ∗ *wp b P s*)
unfolding *wp-def PC-def* **by**(*simp*)

lemma *eval-wlp-PC*:
$wlp\ (a\ {}_{Q}\!\oplus\ b)\ P = (\lambda s.\ Q\ s * wlp\ a\ P\ s + (1 - Q\ s) * wlp\ b\ P\ s)$
unfolding *wlp-def PC-def* **by**(*simp*)

lemma *eval-wp-DC*:
$wp\ (a \sqcap b)\ P = (\lambda s.\ min\ (wp\ a\ P\ s)\ (wp\ b\ P\ s))$
unfolding *wp-def DC-def* **by**(*simp*)

lemma *eval-wlp-DC*:
$wlp\ (a \sqcap b)\ P = (\lambda s.\ min\ (wlp\ a\ P\ s)\ (wlp\ b\ P\ s))$
unfolding *wlp-def DC-def* **by**(*simp*)

lemma *eval-wp-AC*:
$wp\ (a \sqcup b)\ P = (\lambda s.\ max\ (wp\ a\ P\ s)\ (wp\ b\ P\ s))$
unfolding *wp-def AC-def* **by**(*simp*)

lemma *eval-wlp-AC*:
$wlp\ (a \sqcup b)\ P = (\lambda s.\ max\ (wlp\ a\ P\ s)\ (wlp\ b\ P\ s))$
unfolding *wlp-def AC-def* **by**(*simp*)

lemma *eval-wp-Embed*:
$wp\ (Embed\ t) = t$
unfolding *wp-def Embed-def* **by**(*simp*)

lemma *eval-wlp-Embed*:
$wlp\ (Embed\ t) = t$
unfolding *wlp-def Embed-def* **by**(*simp*)

lemma *eval-wp-SetDC*:
$wp\ (SetDC\ p\ S)\ R\ s = Inf\ ((\lambda a.\ wp\ (p\ a)\ R\ s)\ `\ S\ s)$
unfolding *wp-def SetDC-def* **by**(*simp*)

lemma *eval-wlp-SetDC*:
$wlp\ (SetDC\ p\ S)\ R\ s = Inf\ ((\lambda a.\ wlp\ (p\ a)\ R\ s)\ `\ S\ s)$
unfolding *wlp-def SetDC-def* **by**(*simp*)

lemma *eval-wp-SetPC*:
$wp\ (SetPC\ f\ p)\ P = (\lambda s.\ \sum a{\in}supp\ (p\ s).\ p\ s\ a * wp\ (f\ a)\ P\ s)$
unfolding *wp-def SetPC-def* **by**(*simp*)

lemma *eval-wlp-SetPC*:
$wlp\ (SetPC\ f\ p)\ P = (\lambda s.\ \sum a{\in}supp\ (p\ s).\ p\ s\ a * wlp\ (f\ a)\ P\ s)$
unfolding *wlp-def SetPC-def* **by**(*simp*)

lemma *eval-wp-Mu*:
$wp\ (\mu\ t.\ T\ t) = lfp\text{-}trans\ (\lambda t.\ wp\ (T\ (Embed\ t)))$
unfolding *wp-def Mu-def* **by**(*simp*)

lemma *eval-wlp-Mu*:

wlp (μ *t*. *T t*) = *gfp-trans* (λ*t*. *wlp* (*T* (*Embed t*)))
unfolding *wlp-def Mu-def* **by**(*simp*)

lemma *eval-wp-Bind*:
wp (*Bind g f*) = (λ*P s*. *wp* (*f* (*g s*)) *P s*)
unfolding *Bind-def wp-def Let-def* **by**(*simp*)

lemma *eval-wlp-Bind*:
wlp (*Bind g f*) = (λ*P s*. *wlp* (*f* (*g s*)) *P s*)
unfolding *Bind-def wlp-def Let-def* **by**(*simp*)

Use simp add:wp_eval to fully unfold a program fragment

lemmas *wp-eval* = *eval-wp-Abort eval-wlp-Abort eval-wp-Skip eval-wlp-Skip*
eval-wp-Apply eval-wlp-Apply eval-wp-Seq eval-wlp-Seq
eval-wp-PC eval-wlp-PC eval-wp-DC eval-wlp-DC
eval-wp-AC eval-wlp-AC
eval-wp-Embed eval-wlp-Embed eval-wp-SetDC eval-wlp-SetDC
eval-wp-SetPC eval-wlp-SetPC eval-wp-Mu eval-wlp-Mu
eval-wp-Bind eval-wlp-Bind

lemma *Skip-Seq*:
Skip ;; *A* = *A*
unfolding *Skip-def Seq-def o-def* **by**(*rule refl*)

lemma *Seq-Skip*:
A ;; *Skip* = *A*
unfolding *Skip-def Seq-def o-def* **by**(*rule refl*)

Use these as simp rules to clear out Skips

lemmas *skip-simps* = *Skip-Seq Seq-Skip*

end

4.2 Healthiness

theory *Healthiness* **imports** *Embedding* **begin**

4.2.1 The Healthiness of the Embedding

Healthiness is mostly derived by structural induction using the simplifier. *Abort*, *Skip* and *Apply* form base cases.

lemma *healthy-wp-Abort*:
healthy (*wp Abort*)
proof(*rule healthy-parts*)
fix *b* **and** *P*::'*a* ⇒ *real*
assume *nP*: *nneg P* **and** *bP*: *bounded-by b P*
thus *bounded-by b* (*wp Abort P*)

```
    unfolding wp-eval by(blast)
  show nneg (wp Abort P)
    unfolding wp-eval by(blast)
next
  fix P Q::'a expect
  show wp Abort P ⊩ wp Abort Q
    unfolding wp-eval by(blast)
next
  fix P and c and s::'a
  show c * wp Abort P s = wp Abort (λs. c * P s) s
    unfolding wp-eval by(auto)
qed

lemma nearly-healthy-wlp-Abort:
  nearly-healthy (wlp Abort)
proof(rule nearly-healthyI)
  fix P::'s ⇒ real
  show unitary (wlp Abort P)
    by(simp add:wp-eval)
next
  fix P Q :: 's expect
  assume P ⊩ Q and unitary P and unitary Q
  thus wlp Abort P ⊩ wlp Abort Q
    unfolding wp-eval by(blast)
qed

lemma healthy-wp-Skip:
  healthy (wp Skip)
  by(force intro!:healthy-parts simp:wp-eval)

lemma nearly-healthy-wlp-Skip:
  nearly-healthy (wlp Skip)
  by(auto simp:wp-eval)

lemma healthy-wp-Seq:
  fixes t::'s prog and u
  assumes ht: healthy (wp t) and hu: healthy (wp u)
  shows healthy (wp (t ;; u))
proof(rule healthy-parts, simp-all add:wp-eval)
  fix b and P::'s ⇒ real
  assume bounded-by b P and nneg P
  with hu have bounded-by b (wp u P) and nneg (wp u P) by(auto)
  with ht show bounded-by b (wp t (wp u P))
         and nneg (wp t (wp u P)) by(auto)
next
  fix P::'s ⇒ real and Q
  assume sound P and sound Q and P ⊩ Q
  with hu have sound (wp u P) and sound (wp u Q)
    and wp u P ⊩ wp u Q by(auto)
```

with *ht* **show** $wp\ t\ (wp\ u\ P) \Vdash wp\ t\ (wp\ u\ Q)$ **by**(*auto*)
next
fix $P::'s \Rightarrow real$ **and** $c::real$ **and** s
assume *pos*: $0 \leq c$ **and** *sP*: *sound P*
with *ht* **and** *hu* **have** $c * wp\ t\ (wp\ u\ P)\ s = wp\ t\ (\lambda s.\ c * wp\ u\ P\ s)\ s$
by(*auto intro*!:*scalingD*)
also with *hu* **and** *pos* **and** *sP* **have** $\ldots = wp\ t\ (wp\ u\ (\lambda s.\ c * P\ s))\ s$
by(*simp add*:*scalingD*[*OF healthy-scalingD*])
finally show $c * wp\ t\ (wp\ u\ P)\ s = wp\ t\ (wp\ u\ (\lambda s.\ c * P\ s))\ s$ **.**
qed

lemma *nearly-healthy-wlp-Seq*:
fixes $t::'s\ prog$ **and** u
assumes *ht*: *nearly-healthy* (*wlp t*) **and** *hu*: *nearly-healthy* (*wlp u*)
shows *nearly-healthy* (*wlp* (*t* ;; *u*))
proof(*rule nearly-healthyI*, *simp-all add*:*wp-eval*)
fix *b* **and** $P::'s \Rightarrow real$
assume *unitary P*
with *hu* **have** *unitary* (*wlp u P*) **by**(*auto*)
with *ht* **show** *unitary* (*wlp t* (*wlp u P*)) **by**(*auto*)
next
fix $P\ Q::'s \Rightarrow real$
assume *unitary P* **and** *unitary Q* **and** $P \Vdash Q$
with *hu* **have** *unitary* (*wlp u P*) **and** *unitary* (*wlp u Q*)
and $wlp\ u\ P \Vdash wlp\ u\ Q$ **by**(*auto*)
with *ht* **show** $wlp\ t\ (wlp\ u\ P) \Vdash wlp\ t\ (wlp\ u\ Q)$ **by**(*auto*)
qed

lemma *healthy-wp-PC*:
fixes $f::'s\ prog$
assumes *hf*: *healthy* (*wp f*) **and** *hg*: *healthy* (*wp g*)
and *uP*: *unitary P*
shows *healthy* ($wp\ (f\ {}_P\!\oplus\ g)$)
proof(*intro healthy-parts bounded-byI nnegI le-funI*, *simp-all add*:*wp-eval*)
fix *b* **and** $Q::'s \Rightarrow real$ **and** $s::'s$
assume *nQ*: *nneg Q* **and** *bQ*: *bounded-by b Q*

Non-negative:

from *nQ* **and** *bQ* **and** *hf* **have** $0 \leq wp\ f\ Q\ s$ **by**(*auto*)
with *uP* **have** $0 \leq P\ s * \ldots$ **by**(*auto intro*:*mult-nonneg-nonneg*)
moreover {
from *uP* **have** $0 \leq 1 - P\ s$
by *auto*
with *nQ* **and** *bQ* **and** *hg* **have** $0 \leq \ldots * wp\ g\ Q\ s$
by (*metis healthy-nnegD2 mult-nonneg-nonneg nneg-def*)
}
ultimately show $0 \leq P\ s * wp\ f\ Q\ s + (1 - P\ s) * wp\ g\ Q\ s$
by(*auto intro*:*mult-nonneg-nonneg*)

Bounded:

from *nQ bQ hf* **have** $wp\ f\ Q\ s \leq b$ **by**(*auto*)
with *uP nQ bQ hf* **have** $P\ s * wp\ f\ Q\ s \leq P\ s * b$
by(*blast intro*!:*mult-mono*)
moreover {
from *nQ bQ hg uP*
have $wp\ g\ Q\ s \leq b$ **and** $0 \leq 1 - P\ s$
by *auto*
with *nQ bQ hg* **have** $(1 - P\ s) * wp\ g\ Q\ s \leq (1 - P\ s) * b$
by(*blast intro*!:*mult-mono*)
}
ultimately have $P\ s * wp\ f\ Q\ s + (1 - P\ s) * wp\ g\ Q\ s \leq$
$P\ s * b + (1 - P\ s) * b$
by(*blast intro*:*add-mono*)
also have $\ldots = b$ **by**(*auto simp*:*algebra-simps*)
finally show $P\ s * wp\ f\ Q\ s + (1 - P\ s) * wp\ g\ Q\ s \leq b$ **.**
next

Monotonic:

fix $Q\ R::'s \Rightarrow real$ **and** s
assume *sQ*: *sound Q* **and** *sR*: *sound R* **and** *le*: $Q \Vdash R$

with *hf* **have** $wp\ f\ Q\ s \leq wp\ f\ R\ s$ **by**(*blast dest*:*mono-transD*)
with *uP* **have** $P\ s * wp\ f\ Q\ s \leq P\ s * wp\ f\ R\ s$
by(*auto intro*:*mult-left-mono*)
moreover {
from *sQ sR le hg*
have $wp\ g\ Q\ s \leq wp\ g\ R\ s$ **by**(*blast dest*:*mono-transD*)
moreover from *uP* **have** $0 \leq 1 - P\ s$
by *auto*
ultimately have $(1 - P\ s) * wp\ g\ Q\ s \leq (1 - P\ s) * wp\ g\ R\ s$
by(*auto intro*:*mult-left-mono*)
}
ultimately show $P\ s * wp\ f\ Q\ s + (1 - P\ s) * wp\ g\ Q\ s \leq$
$P\ s * wp\ f\ R\ s + (1 - P\ s) * wp\ g\ R\ s$ **by**(*auto*)
next

Scaling:

fix $Q::'s \Rightarrow real$ **and** $c::real$ **and** $s::'s$
assume *sQ*: *sound Q* **and** *pos*: $0 \leq c$
have $c * (P\ s * wp\ f\ Q\ s + (1 - P\ s) * wp\ g\ Q\ s) =$
$P\ s * (c * wp\ f\ Q\ s) + (1 - P\ s) * (c * wp\ g\ Q\ s)$
by(*simp add*:*distrib-left*)
also have $\ldots = P\ s * wp\ f\ (\lambda s.\ c * Q\ s)\ s +$
$(1 - P\ s) * wp\ g\ (\lambda s.\ c * Q\ s)\ s$
using *hf hg sQ pos*
by(*simp add*:*scalingD*[*OF healthy-scalingD*])
finally show $c * (P\ s * wp\ f\ Q\ s + (1 - P\ s) * wp\ g\ Q\ s) =$
$P\ s * wp\ f\ (\lambda s.\ c * Q\ s)\ s + (1 - P\ s) * wp\ g\ (\lambda s.\ c * Q\ s)\ s$ **.**
qed

```
lemma nearly-healthy-wlp-PC:
  fixes f::'s prog
  assumes hf: nearly-healthy (wlp f)
     and hg: nearly-healthy (wlp g)
     and uP: unitary P
  shows nearly-healthy (wlp (f P⊕ g))
proof(intro nearly-healthyI unitaryI2 nnegI bounded-byI le-funI,
     simp-all add:wp-eval)
  fix Q::'s expect and s::'s
  assume uQ: unitary Q
  from uQ hf hg have utQ: unitary (wlp f Q) unitary (wlp g Q) by(auto)
  from uP have nnP: 0 ≤ P s 0 ≤ 1 − P s
   by auto
  moreover from utQ have 0 ≤ wlp f Q s 0 ≤ wlp g Q s by(auto)
  ultimately show 0 ≤ P s ∗ wlp f Q s + (1 − P s) ∗ wlp g Q s
   by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)

  from utQ have wlp f Q s ≤ 1 wlp g Q s ≤ 1 by(auto)
  with nnP have P s ∗ wlp f Q s + (1 − P s) ∗ wlp g Q s ≤ P s ∗ 1 + (1 − P s) ∗ 1
   by(blast intro:add-mono mult-left-mono)
  thus P s ∗ wlp f Q s + (1 − P s) ∗ wlp g Q s ≤ 1 by(simp)

  fix R::'s expect
  assume uR: unitary R and le: Q ⊩ R
  with uQ have wlp f Q s ≤ wlp f R s
   by(auto intro:le-funD[OF nearly-healthy-monoD, OF hf])
  with nnP have P s ∗ wlp f Q s ≤ P s ∗ wlp f R s
   by(auto intro:mult-left-mono)
  moreover {
   from uQ uR le have wlp g Q s ≤ wlp g R s
     by(auto intro:le-funD[OF nearly-healthy-monoD, OF hg])
   with nnP have (1 − P s) ∗ wlp g Q s ≤ (1 − P s) ∗ wlp g R s
     by(auto intro:mult-left-mono)
  }
  ultimately show P s ∗ wlp f Q s + (1 − P s) ∗ wlp g Q s ≤
              P s ∗ wlp f R s + (1 − P s) ∗ wlp g R s
   by(auto)
qed

lemma healthy-wp-DC:
  fixes f::'s prog
  assumes hf: healthy (wp f) and hg: healthy (wp g)
  shows healthy (wp (f ⨅ g))
proof(intro healthy-parts bounded-byI nnegI le-funI, simp-all only:wp-eval)
  fix b and P::'s ⇒ real and s::'s
  assume nP: nneg P and bP: bounded-by b P

  with hf have bounded-by b (wp f P) by(auto)
```

```
  hence wp f P s ≤ b by(blast)
  thus min (wp f P s) (wp g P s) ≤ b by(auto)

  from nP bP assms show 0 ≤ min (wp f P s) (wp g P s) by(auto)
next
  fix P::'s ⇒ real and Q and s::'s
  from assms have mf: mono-trans (wp f) and mg: mono-trans (wp g) by(auto)
  assume sP: sound P and sQ: sound Q and le: P ⊩ Q
  hence wp f P s ≤ wp f Q s and wp g P s ≤ wp g Q s
    by(auto intro:le-funD[OF mono-transD[OF mf]] le-funD[OF mono-transD[OF mg]])
  thus min (wp f P s) (wp g P s) ≤ min (wp f Q s) (wp g Q s) by(auto)
next
  fix P::'s ⇒ real and c::real and s::'s
  assume sP: sound P and pos: 0 ≤ c
  from assms have sf: scaling (wp f) and sg: scaling (wp g) by(auto)
  from pos have c ∗ min (wp f P s) (wp g P s) =
              min (c ∗ wp f P s) (c ∗ wp g P s)
    by(simp add:min-distrib)
  also from sP and pos
  have ... = min (wp f (λs. c ∗ P s) s) (wp g (λs. c ∗ P s) s)
    by(simp add:scalingD[OF sf] scalingD[OF sg])
  finally show c ∗ min (wp f P s) (wp g P s) =
             min (wp f (λs. c ∗ P s) s) (wp g (λs. c ∗ P s) s) .
qed

lemma nearly-healthy-wlp-DC:
  fixes f::'s prog
  assumes hf: nearly-healthy (wlp f)
      and hg: nearly-healthy (wlp g)
  shows nearly-healthy (wlp (f ⨅ g))
proof(intro nearly-healthyI bounded-byI nnegI le-funI unitaryI2,
      simp-all add:wp-eval, safe)
  fix P::'s ⇒ real and s::'s
  assume uP: unitary P
  with hf hg have utP: unitary (wlp f P) unitary (wlp g P) by(auto)
  thus 0 ≤ wlp f P s 0 ≤ wlp g P s by(auto)

  have min (wlp f P s) (wlp g P s) ≤ wlp f P s by(auto)
  also from utP have ... ≤ 1 by(auto)
  finally show min (wlp f P s) (wlp g P s) ≤ 1 .

  fix Q::'s ⇒ real
  assume uQ: unitary Q and le: P ⊩ Q
  have min (wlp f P s) (wlp g P s) ≤ wlp f P s by(auto)
  also from uP uQ le have ... ≤ wlp f Q s
    by(auto intro:le-funD[OF nearly-healthy-monoD, OF hf])
  finally show min (wlp f P s) (wlp g P s) ≤ wlp f Q s .

  have min (wlp f P s) (wlp g P s) ≤ wlp g P s by(auto)
```

```
  also from uP uQ le have ... ≤ wlp g Q s
    by(auto intro:le-funD[OF nearly-healthy-monoD, OF hg])
  finally show min (wlp f P s) (wlp g P s) ≤ wlp g Q s .
qed

lemma healthy-wp-AC:
  fixes f::'s prog
  assumes hf: healthy (wp f) and hg: healthy (wp g)
  shows healthy (wp (f ⨆ g))
proof(intro healthy-parts bounded-byI nnegI le-funI, simp-all only:wp-eval)
  fix b and P::'s ⇒ real and s::'s
  assume nP: nneg P and bP: bounded-by b P

  with hf have bounded-by b (wp f P) by(auto)
  hence wp f P s ≤ b by(blast)
  moreover {
    from bP nP hg have bounded-by b (wp g P) by(auto)
    hence wp g P s ≤ b by(blast)
  }
  ultimately show max (wp f P s) (wp g P s) ≤ b by(auto)

  from nP bP assms have 0 ≤ wp f P s by(auto)
  thus 0 ≤ max (wp f P s) (wp g P s) by(auto)
next
  fix P::'s ⇒ real and Q and s::'s
  from assms have mf: mono-trans (wp f) and mg: mono-trans (wp g) by(auto)
  assume sP: sound P and sQ: sound Q and le: P ⊩ Q
  hence wp f P s ≤ wp f Q s and wp g P s ≤ wp g Q s
    by(auto intro:le-funD[OF mono-transD, OF mf] le-funD[OF mono-transD, OF mg])
  thus max (wp f P s) (wp g P s) ≤ max (wp f Q s) (wp g Q s) by(auto)
next
  fix P::'s ⇒ real and c::real and s::'s
  assume sP: sound P and pos: 0 ≤ c
  from assms have sf: scaling (wp f) and sg: scaling (wp g) by(auto)
  from pos have c ∗ max (wp f P s) (wp g P s) =
               max (c ∗ wp f P s) (c ∗ wp g P s)
    by(simp add:max-distrib)
  also from sP and pos
  have ... = max (wp f (λs. c ∗ P s) s) (wp g (λs. c ∗ P s) s)
    by(simp add:scalingD[OF sf] scalingD[OF sg])
  finally show c ∗ max (wp f P s) (wp g P s) =
               max (wp f (λs. c ∗ P s) s) (wp g (λs. c ∗ P s) s) .
qed

lemma nearly-healthy-wlp-AC:
  fixes f::'s prog
  assumes hf: nearly-healthy (wlp f)
      and hg: nearly-healthy (wlp g)
  shows nearly-healthy (wlp (f ⨆ g))
```

```
proof(intro nearly-healthyI bounded-byI nnegI unitaryI2 le-funI, simp-all only:wp-eval)
  fix b and P::'s ⇒ real and s::'s
  assume uP: unitary P

  with hf have wlp f P s ≤ 1 by(auto)
  moreover from uP hg have unitary (wlp g P) by(auto)
  hence wlp g P s ≤ 1 by(auto)
  ultimately show max (wlp f P s) (wlp g P s) ≤ 1 by(auto)

  from uP hf have unitary (wlp f P) by(auto)
  hence 0 ≤ wlp f P s by(auto)
  thus 0 ≤ max (wlp f P s) (wlp g P s) by(auto)
next
  fix P::'s ⇒ real and Q and s::'s
  assume uP: unitary P and uQ: unitary Q and le: P ⊩ Q
  hence wlp f P s ≤ wlp f Q s and wlp g P s ≤ wlp g Q s
    by(auto intro:le-funD[OF nearly-healthy-monoD, OF hf]
              le-funD[OF nearly-healthy-monoD, OF hg])
  thus max (wlp f P s) (wlp g P s) ≤ max (wlp f Q s) (wlp g Q s) by(auto)
qed

lemma healthy-wp-Embed:
  healthy t ⟹ healthy (wp (Embed t))
  unfolding wp-def Embed-def by(simp)

lemma nearly-healthy-wlp-Embed:
  nearly-healthy t ⟹ nearly-healthy (wlp (Embed t))
  unfolding wlp-def Embed-def by(simp)

lemma healthy-wp-repeat:
  assumes h-a: healthy (wp a)
  shows healthy (wp (repeat n a)) (is ?X n)
proof(induct n)
  show ?X 0 by(auto simp:wp-eval)
next
  fix n assume IH: ?X n
  thus ?X (Suc n) by(simp add:healthy-wp-Seq h-a)
qed

lemma nearly-healthy-wlp-repeat:
  assumes h-a: nearly-healthy (wlp a)
  shows nearly-healthy (wlp (repeat n a)) (is ?X n)
proof(induct n)
  show ?X 0 by(simp add:wp-eval)
next
  fix n assume IH: ?X n
  thus ?X (Suc n) by(simp add:nearly-healthy-wlp-Seq h-a)
qed
```

```
lemma healthy-wp-SetDC:
  fixes prog::'b ⇒ 'a prog and S::'a ⇒ 'b set
  assumes healthy: ⋀x s. x ∈ S s ⟹ healthy (wp (prog x))
    and nonempty: ⋀s. ∃x. x ∈ S s
  shows healthy (wp (SetDC prog S)) (is healthy ?T)
proof(intro healthy-parts bounded-byI nnegI le-funI, simp-all only:wp-eval)
  fix b and P::'a ⇒ real and s::'a
  assume bP: bounded-by b P and nP: nneg P
  hence sP: sound P by(auto)

  from nonempty obtain x where xin: x ∈ (λa. wp (prog a) P s) ` S s by(blast)
  moreover from sP and healthy
  have ∀x∈(λa. wp (prog a) P s) ` S s. 0 ≤ x by(auto)
  ultimately have Inf ((λa. wp (prog a) P s) ` S s) ≤ x
    by(intro cInf-lower bdd-belowI, auto)
  also from xin and healthy and sP and bP have x ≤ b by(blast)
  finally show Inf ((λa. wp (prog a) P s) ` S s) ≤ b .

  from xin and sP and healthy
  show 0 ≤ Inf ((λa. wp (prog a) P s) ` S s) by(blast intro:cInf-greatest)
next
  fix P::'a ⇒ real and Q and s::'a
  assume sP: sound P and sQ: sound Q and le: P ⊩ Q

  from nonempty obtain x where xin: x ∈ (λa. wp (prog a) P s) ` S s by(blast)
  moreover from sP and healthy
  have ∀x∈(λa. wp (prog a) P s) ` S s. 0 ≤ x by(auto)
  moreover
  have ∀x∈(λa. wp (prog a) Q s) ` S s. ∃y∈(λa. wp (prog a) P s) ` S s. y ≤ x
  proof(rule ballI, clarify, rule bexI)
    fix x and a assume ain: a ∈ S s
    with healthy and sP and sQ and le show wp (prog a) P s ≤ wp (prog a) Q s
      by(auto dest:mono-transD[OF healthy-monoD])
    from ain show wp (prog a) P s ∈ (λa. wp (prog a) P s) ` S s by(simp)
  qed
  ultimately
  show Inf ((λa. wp (prog a) P s) ` S s) ≤ Inf ((λa. wp (prog a) Q s) ` S s)
    by(intro cInf-mono, blast+)
next
  fix P::'a ⇒ real and c::real and s::'a
  assume sP: sound P and pos: 0 ≤ c
  from nonempty obtain x where xin: x ∈ (λa. wp (prog a) P s) ` S s by(blast)
  have c ∗ Inf ((λa. wp (prog a) P s) ` S s) =
      Inf ((∗) c ` ((λa. wp (prog a) P s) ` S s)) (is ?U = ?V)
  proof(rule antisym)
    show ?U ≤ ?V
    proof(rule cInf-greatest)
      from nonempty show (∗) c ` (λa. wp (prog a) P s) ` S s ≠ {} by(auto)
      fix x assume x ∈ (∗) c ` (λa. wp (prog a) P s) ` S s
```

```
    then obtain y where yin: y ∈ (λa. wp (prog a) P s) ‘ S s and rwx: x = c ∗ y by(auto)
    have Inf ((λa. wp (prog a) P s) ‘ S s) ≤ y
    proof(intro cInf-lower[OF yin] bdd-belowI)
      fix z assume zin: z ∈ (λa. wp (prog a) P s) ‘ S s
      then obtain a where a ∈ S s and z = wp (prog a) P s by(auto)
      with sP show 0 ≤ z by(auto dest:healthy)
    qed
   with pos rwx show c ∗ Inf ((λa. wp (prog a) P s) ‘ S s) ≤ x by(auto intro:mult-left-mono)
  qed
  show ?V ≤ ?U
  proof(cases)
    assume cz: c = 0
    moreover {
      from nonempty obtain c where c ∈ S s by(auto)
      hence ∃x. ∃xa∈S s. x = wp (prog xa) P s by(auto)
    }
    ultimately show ?thesis by(simp add:image-def)
  next
    assume c ≠ 0
    from nonempty have S s ≠ {} by blast
    then have inverse c ∗ (INF x∈S s. c ∗ wp (prog x) P s) ≤ (INF a∈S s. wp (prog a) P s)
    proof (rule cINF-greatest)
      fix x
      assume x ∈ S s
      have bdd-below ((λx. c ∗ wp (prog x) P s) ‘ S s)
      proof (rule bdd-belowI [of - 0])
        fix z
        assume z ∈ (λx. c ∗ wp (prog x) P s) ‘ S s
        then obtain b where b ∈ S s and rwz: z = c ∗ wp (prog b) P s by auto
        with sP have 0 ≤ wp (prog b) P s by (auto dest: healthy)
        with pos show 0 ≤ z by (auto simp: rwz intro: mult-nonneg-nonneg)
      qed
      then have (INF x∈S s. c ∗ wp (prog x) P s) ≤ c ∗ wp (prog x) P s
      using ‹x ∈ S s› by (rule cINF-lower)
      with ‹c ≠ 0› show inverse c ∗ (INF x∈S s. c ∗ wp (prog x) P s) ≤ wp (prog x) P s
        by (simp add: mult-div-mono-left pos)
    qed
    with ‹c ≠ 0› have inverse c ∗ ?V ≤ inverse c ∗ ?U
      by (simp add: mult.assoc [symmetric] image-comp)
    with pos have c ∗ (inverse c ∗ ?V) ≤ c ∗ (inverse c ∗ ?U)
      by(auto intro:mult-left-mono)
    with ‹c ≠ 0› show ?thesis by (simp add:mult.assoc [symmetric])
  qed
qed
also have ... = Inf ((λa. c ∗ wp (prog a) P s) ‘ S s)
  by (simp add: image-comp)
also from sP and pos have ... = Inf ((λa. wp (prog a) (λs. c ∗ P s) s) ‘ S s)
  by(simp add:scalingD[OF healthy-scalingD, OF healthy] cong:image-cong)
finally show c ∗ Inf ((λa. wp (prog a) P s) ‘ S s) =
```

```
        Inf ((λa. wp (prog a) (λs. c ∗ P s) s) ‘ S s) .
qed

lemma nearly-healthy-wlp-SetDC:
  fixes prog::'b ⇒ 'a prog and S::'a ⇒ 'b set
  assumes healthy: ⋀x s. x ∈ S s ⟹ nearly-healthy (wlp (prog x))
    and nonempty: ⋀s. ∃x. x ∈ S s
  shows nearly-healthy (wlp (SetDC prog S)) (is nearly-healthy ?T)
proof(intro nearly-healthyI unitaryI2 bounded-byI nnegI le-funI, simp-all only:wp-eval)
  fix b and P::'a ⇒ real and s::'a
  assume uP: unitary P

  from nonempty obtain x where xin: x ∈ (λa. wlp (prog a) P s) ‘ S s by(blast)
  moreover {
    from uP healthy
    have ∀x∈(λa. wlp (prog a) P) ‘ S s. unitary x by(auto)
    hence ∀x∈(λa. wlp (prog a) P) ‘ S s. 0 ≤ x s by(auto)
    hence ∀y∈(λa. wlp (prog a) P s) ‘ S s. 0 ≤ y by(auto)
  }
  ultimately have Inf ((λa. wlp (prog a) P s) ‘ S s) ≤ x by(intro cInf-lower bdd-belowI,
auto)
  also from xin healthy uP have x ≤ 1 by(blast)
  finally show Inf ((λa. wlp (prog a) P s) ‘ S s) ≤ 1 .

  from xin uP healthy
  show 0 ≤ Inf ((λa. wlp (prog a) P s) ‘ S s)
    by(blast dest!:unitary-sound[OF nearly-healthy-unitaryD[OF - uP]]
          intro:cInf-greatest)
next
  fix P::'a ⇒ real and Q and s::'a
  assume uP: unitary P and uQ: unitary Q and le: P ⊩ Q

  from nonempty obtain x where xin: x ∈ (λa. wlp (prog a) P s) ‘ S s by(blast)
  moreover {
    from uP healthy
    have ∀x∈(λa. wlp (prog a) P) ‘ S s. unitary x by(auto)
    hence ∀x∈(λa. wlp (prog a) P) ‘ S s. 0 ≤ x s by(auto)
    hence ∀y∈(λa. wlp (prog a) P s) ‘ S s. 0 ≤ y by(auto)
  }
  moreover
  have ∀x∈(λa. wlp (prog a) Q s) ‘ S s. ∃y∈(λa. wlp (prog a) P s) ‘ S s. y ≤ x
  proof(rule ballI, clarify, rule bexI)
    fix x and a assume ain: a ∈ S s
    from uP uQ le show wlp (prog a) P s ≤ wlp (prog a) Q s
      by(auto intro:le-funD[OF nearly-healthy-monoD[OF healthy, OF ain]])
    from ain show wlp (prog a) P s ∈ (λa. wlp (prog a) P s) ‘ S s by(simp)
  qed
  ultimately
  show Inf ((λa. wlp (prog a) P s) ‘ S s) ≤ Inf ((λa. wlp (prog a) Q s) ‘ S s)
```

```
  by(intro cInf-mono, blast+)
qed

lemma healthy-wp-SetPC:
  fixes p::'s ⇒ 'a ⇒ real
  and  f::'a ⇒ 's prog
  assumes healthy: ⋀a s. a ∈ supp (p s) ⟹ healthy (wp (f a))
    and sound: ⋀s. sound (p s)
    and sub-dist: ⋀s. (∑a∈supp (p s). p s a) ≤ 1
  shows healthy (wp (SetPC f p)) (is healthy ?X)
proof(intro healthy-parts bounded-byI nnegI le-funI, simp-all add:wp-eval)
  fix b and P::'s ⇒ real and s::'s
  assume bP: bounded-by b P and nP: nneg P
  hence sP: sound P by(auto)

  from sP and bP and healthy have ⋀a s. a ∈ supp (p s) ⟹ wp (f a) P s ≤ b
    by(blast dest:healthy-bounded-byD)
  with sound have (∑a∈supp (p s). p s a ∗ wp (f a) P s) ≤ (∑a∈supp (p s). p s a ∗ b)
    by(blast intro:sum-mono mult-left-mono)
  also have ... = (∑a∈supp (p s). p s a) ∗ b
    by(simp add:sum-distrib-right)
  also {
    from bP and nP have 0 ≤ b by(blast)
    with sub-dist have (∑a∈supp (p s). p s a) ∗ b ≤ 1 ∗ b
      by(rule mult-right-mono)
  }
  also have 1 ∗ b = b by(simp)
  finally show (∑a∈supp (p s). p s a ∗ wp (f a) P s) ≤ b .

  show 0 ≤ (∑a∈supp (p s). p s a ∗ wp (f a) P s)
  proof(rule sum-nonneg [OF mult-nonneg-nonneg])
    fix x
    from sound show 0 ≤ p s x by(blast)
    assume x ∈ supp (p s) with sP and healthy
    show 0 ≤ wp (f x) P s by(blast)
  qed
next
  fix P::'s ⇒ real and Q::'s ⇒ real and s
  assume s-P: sound P and s-Q: sound Q and ent: P ⊩ Q
  with healthy have ⋀a. a ∈ supp (p s) ⟹ wp (f a) P s ≤ wp (f a) Q s
    by(blast)
  with sound show (∑a∈supp (p s). p s a ∗ wp (f a) P s) ≤
              (∑a∈supp (p s). p s a ∗ wp (f a) Q s)
    by(blast intro:sum-mono mult-left-mono)
next
  fix P::'s ⇒ real and c::real and s::'s
  assume sound: sound P and pos: 0 ≤ c
  have c ∗ (∑a∈supp (p s). p s a ∗ wp (f a) P s) =
      (∑a∈supp (p s). p s a ∗ (c ∗ wp (f a) P s))
```

```
    (is ?A = ?B)
   by(simp add:sum-distrib-left ac-simps)
  also from sound and pos and healthy
  have ... = (∑ a∈supp (p s). p s a ∗ wp (f a) (λs. c ∗ P s) s)
   by(auto simp:scalingD[OF healthy-scalingD])
  finally show ?A = ... .
qed

lemma nearly-healthy-wlp-SetPC:
  fixes p::'s ⇒ 'a ⇒ real
  and   f::'a ⇒ 's prog
  assumes healthy: ⋀a s. a ∈ supp (p s) ⟹ nearly-healthy (wlp (f a))
     and sound: ⋀s. sound (p s)
     and sub-dist: ⋀s. (∑ a∈supp (p s). p s a) ≤ 1
  shows nearly-healthy (wlp (SetPC f p)) (is nearly-healthy ?X)
proof(intro nearly-healthyI unitaryI2 bounded-byI nnegI le-funI, simp-all only:wp-eval)
  fix b and P::'s ⇒ real and s::'s
  assume uP: unitary P

  from uP healthy have ⋀a. a ∈ supp (p s) ⟹ unitary (wlp (f a) P) by(auto)
  hence ⋀a. a ∈ supp (p s) ⟹ wlp (f a) P s ≤ 1 by(auto)
  with sound have (∑ a∈supp (p s). p s a ∗ wlp (f a) P s) ≤ (∑ a∈supp (p s). p s a ∗ 1)
   by(blast intro:sum-mono mult-left-mono)
  also have ... = (∑ a∈supp (p s). p s a)
   by(simp add:sum-distrib-right)
  also note sub-dist
  finally show (∑ a∈supp (p s). p s a ∗ wlp (f a) P s) ≤ 1 .
  show 0 ≤ (∑ a∈supp (p s). p s a ∗ wlp (f a) P s)
  proof(rule sum-nonneg [OF mult-nonneg-nonneg])
   fix x
   from sound show 0 ≤ p s x by(blast)
   assume x ∈ supp (p s) with uP healthy
   show 0 ≤ wlp (f x) P s by(blast)
  qed
next
  fix P::'s expect and Q::'s expect and s
  assume uP: unitary P and uQ: unitary Q and le: P ⊩ Q
  hence ⋀a. a ∈ supp (p s) ⟹ wlp (f a) P s ≤ wlp (f a) Q s
   by(blast intro:le-funD[OF nearly-healthy-monoD, OF healthy])
  with sound show (∑ a∈supp (p s). p s a ∗ wlp (f a) P s) ≤
              (∑ a∈supp (p s). p s a ∗ wlp (f a) Q s)
   by(blast intro:sum-mono mult-left-mono)
qed

lemma healthy-wp-Apply:
  healthy (wp (Apply f))
  unfolding Apply-def wp-def by(blast)

lemma nearly-healthy-wlp-Apply:
```

nearly-healthy (*wlp* (*Apply f*))
by(*intro nearly-healthyI unitaryI2 nnegI bounded-byI*, *auto simp*:*o-def wp-eval*)

lemma *healthy-wp-Bind*:
fixes f::$'s \Rightarrow 'a$
assumes *hsub*: $\bigwedge s.$ *healthy* (*wp* (*p* (*f s*)))
shows *healthy* (*wp* (*Bind f p*))
proof(*intro healthy-parts nnegI bounded-byI le-funI*, *simp-all only*:*wp-eval*)
fix *b* **and** *P*::$'s$ *expect* **and** *s*::$'s$
assume *bP*: *bounded-by b P* **and** *nP*: *nneg P*
with *hsub* **have** *bounded-by b* (*wp* (*p* (*f s*)) *P*) **by**(*auto*)
thus *wp* (*p* (*f s*)) *P s* $\leq$ *b* **by**(*auto*)
from *bP nP hsub* **have** *nneg* (*wp* (*p* (*f s*)) *P*) **by**(*auto*)
thus $0 \leq$ *wp* (*p* (*f s*)) *P s* **by**(*auto*)
next
fix *P Q*::$'s$ *expect* **and** *s*::$'s$
assume *sound P sound Q P* $\Vdash$ *Q*
thus *wp* (*p* (*f s*)) *P s* $\leq$ *wp* (*p* (*f s*)) *Q s*
by(*rule le-funD*[*OF mono-transD*, *OF healthy-monoD*, *OF hsub*])
next
fix *P*::$'s$ *expect* **and** *c*::*real* **and** *s*::$'s$
assume *sound P* **and** $0 \leq c$
thus $c *$ *wp* (*p* (*f s*)) *P s* = *wp* (*p* (*f s*)) ($\lambda s.\ c * P\ s$) *s*
by(*simp add*:*scalingD*[*OF healthy-scalingD*, *OF hsub*])
qed

lemma *nearly-healthy-wlp-Bind*:
fixes f::$'s \Rightarrow 'a$
assumes *hsub*: $\bigwedge s.$ *nearly-healthy* (*wlp* (*p* (*f s*)))
shows *nearly-healthy* (*wlp* (*Bind f p*))
proof(*intro nearly-healthyI unitaryI2 nnegI bounded-byI le-funI*, *simp-all only*:*wp-eval*)
fix *P*::$'s$ *expect* **and** *s*::$'s$ **assume** *uP*: *unitary P*
with *hsub* **have** *unitary* (*wlp* (*p* (*f s*)) *P*) **by**(*auto*)
thus $0 \leq$ *wlp* (*p* (*f s*)) *P s wlp* (*p* (*f s*)) *P s* ≤ 1 **by**(*auto*)

fix *Q*::$'s$ *expect*
assume *unitary Q P* $\Vdash$ *Q*
with *uP* **show** *wlp* (*p* (*f s*)) *P s* $\leq$ *wlp* (*p* (*f s*)) *Q s*
by(*blast intro*:*le-funD*[*OF nearly-healthy-monoD*, *OF hsub*])
qed

4.2.2 Healthiness for Loops

lemma *wp-loop-step-mono*:
fixes *t u*::$'s$ *trans*
assumes *hb*: *healthy* (*wp body*)
and *le*: *le-trans t u*
and *ht*: $\bigwedge P.$ *sound P* $\Longrightarrow$ *sound* (*t P*)
and *hu*: $\bigwedge P.$ *sound P* $\Longrightarrow$ *sound* (*u P*)

```
  shows le-trans (wp (body ;; Embed t « G »⊕ Skip))
         (wp (body ;; Embed u « G »⊕ Skip))
proof(intro le-transI le-funI, simp add:wp-eval)
  fix P::'s expect and s::'s
  assume sP: sound P
  with le have t P ⊩ u P by(auto)
  moreover from sP ht hu have sound (t P) sound (u P) by(auto)
  ultimately have wp body (t P) s ≤ wp body (u P) s
    by(auto intro:le-funD[OF mono-transD, OF healthy-monoD, OF hb])
  thus «G» s ∗ wp body (t P) s ≤ «G» s ∗ wp body (u P) s
    by(auto intro:mult-left-mono)
qed

lemma wlp-loop-step-mono:
  fixes t u::'s trans
  assumes mb: nearly-healthy (wlp body)
      and le: le-utrans t u
      and ht: ⋀P. unitary P ⟹ unitary (t P)
      and hu: ⋀P. unitary P ⟹ unitary (u P)
  shows le-utrans (wlp (body ;; Embed t « G »⊕ Skip))
          (wlp (body ;; Embed u « G »⊕ Skip))
proof(intro le-utransI le-funI, simp add:wp-eval)
  fix P::'s expect and s::'s
  assume uP: unitary P
  with le have t P ⊩ u P by(auto)
  moreover from uP ht hu have unitary (t P) unitary (u P) by(auto)
  ultimately have wlp body (t P) s ≤ wlp body (u P) s
    by(rule le-funD[OF nearly-healthy-monoD[OF mb]])
  thus «G» s ∗ wlp body (t P) s ≤ «G» s ∗ wlp body (u P) s
    by(auto intro:mult-left-mono)
qed
```

For each sound expectation, we have a pre fixed point of the loop body. This lets us use the relevant fixed-point lemmas.

```
lemma lfp-loop-fp:
  assumes hb: healthy (wp body)
      and sP: sound P
  shows λs. «G» s ∗ wp body (λs. bound-of P) s + «𝒩 G» s ∗ P s ⊩ λs. bound-of P
proof(rule le-funI)
  fix s
  from sP have sound (λs. bound-of P) by(auto)
  moreover hence bounded-by (bound-of P) (λs. bound-of P) by(auto)
  ultimately have bounded-by (bound-of P) (wp body (λs. bound-of P))
    using hb by(auto)
  hence wp body (λs. bound-of P) s ≤ bound-of P by(auto)
  moreover from sP have P s ≤ bound-of P by(auto)
  ultimately have «G» s ∗ wp body (λa. bound-of P) s + (1 − «G» s) ∗ P s ≤
          «G» s ∗ bound-of P + (1 − «G» s) ∗ bound-of P
    by(blast intro:add-mono mult-left-mono)
```

thus *«G» s ∗ wp body (λa. bound-of P) s + «𝒩 G» s ∗ P s ≤ bound-of P*
by(*simp add*:*algebra-simps negate-embed*)
qed

lemma *lfp-loop-greatest*:
fixes *P*::′*s expect*
assumes *lb*: ⋀*R*. *λs*. *«G» s ∗ wp body R s + «𝒩 G» s ∗ P s* ⊩ *R* ⟹ *sound R* ⟹ *Q* ⊩ *R*
and *hb*: *healthy* (*wp body*)
and *sP*: *sound P*
and *sQ*: *sound Q*
shows *Q* ⊩ *lfp-exp* (*λQ s*. *«G» s ∗ wp body Q s + «𝒩 G» s ∗ P s*)
using *sP* **by**(*auto intro*!:*lfp-exp-greatest*[*OF lb sQ*] *sP lfp-loop-fp hb*)

lemma *lfp-loop-sound*:
fixes *P*::′*s expect*
assumes *hb*: *healthy* (*wp body*)
and *sP*: *sound P*
shows *sound* (*lfp-exp* (*λQ s*. *«G» s ∗ wp body Q s + «𝒩 G» s ∗ P s*))
using *assms* **by**(*auto intro*!:*lfp-exp-sound lfp-loop-fp*)

lemma *wlp-loop-step-unitary*:
fixes *t u*::′*s trans*
assumes *hb*: *nearly-healthy* (*wlp body*)
and *ht*: ⋀*P*. *unitary P* ⟹ *unitary* (*t P*)
and *uP*: *unitary P*
shows *unitary* (*wlp* (*body* ;; *Embed t* $_{« G »}$⊕ *Skip*) *P*)
proof(*intro unitaryI2 nnegI bounded-byI*, *simp-all add*:*wp-eval*)
fix *s*::′*s*
from *ht uP* **have** *utP*: *unitary* (*t P*) **by**(*auto*)
with *hb* **have** *unitary* (*wlp body* (*t P*)) **by**(*auto*)
hence *0 ≤ wlp body* (*t P*) *s* **by**(*auto*)
with *uP* **show** *0 ≤ « G » s ∗ wlp body* (*t P*) *s* + (*1* − *« G » s*) ∗ *P s*
by(*auto intro*!:*add-nonneg-nonneg mult-nonneg-nonneg*)
from *ht uP* **have** *bounded-by 1* (*t P*) **by**(*auto*)
with *utP hb* **have** *bounded-by 1* (*wlp body* (*t P*)) **by**(*auto*)
hence *wlp body* (*t P*) *s ≤ 1* **by**(*auto*)
with *uP* **have** *«G» s ∗ wlp body* (*t P*) *s* + (*1* − *«G» s*) ∗ *P s ≤ «G» s ∗ 1* + (*1* − *«G» s*) ∗ *1*
by(*blast intro*:*add-mono mult-left-mono*)
also have ... = *1* **by**(*simp*)
finally show *«G» s ∗ wlp body* (*t P*) *s* + (*1* − *«G» s*) ∗ *P s ≤ 1* **.**
qed

lemma *wp-loop-step-sound*:
fixes *t u*::′*s trans*
assumes *hb*: *healthy* (*wp body*)
and *ht*: ⋀*P*. *sound P* ⟹ *sound* (*t P*)
and *sP*: *sound P*
shows *sound* (*wp* (*body* ;; *Embed t* $_{« G »}$⊕ *Skip*) *P*)

proof(*intro soundI2 nnegI bounded-byI*, *simp-all add*:*wp-eval*)
 fix *s*::*'s*
 from *ht sP* **have** *stP*: *sound* (*t P*) **by**(*auto*)
 with *hb* **have** $0 \leq$ *wp body* (*t P*) *s* **by**(*auto*)
 with *sP* **show** $0 \leq$ « *G* » $s *$ *wp body* (*t P*) $s + (1 -$ « *G* » $s) * P\ s$
  by(*auto intro*!:*add-nonneg-nonneg mult-nonneg-nonneg*)

 from *ht sP* **have** *sound* (*t P*) **by**(*auto*)
 moreover hence *bounded-by* (*bound-of* (*t P*)) (*t P*) **by**(*auto*)
 ultimately have *wp body* (*t P*) $s \leq$ *bound-of* (*t P*) **using** *hb* **by**(*auto*)
 hence *wp body* (*t P*) $s \leq$ *max* (*bound-of P*) (*bound-of* (*t P*)) **by**(*auto*)
 moreover {
  from *sP* **have** $P\ s \leq$ *bound-of P* **by**(*auto*)
  hence $P\ s \leq$ *max* (*bound-of P*) (*bound-of* (*t P*)) **by**(*auto*)
 }
 ultimately
 have «*G*» $s *$ *wp body* (*t P*) $s + (1 -$ «*G*» $s) * P\ s \leq$
   «*G*» $s *$ *max* (*bound-of P*) (*bound-of* (*t P*)) +
   $(1 -$ «*G*» $s) *$ *max* (*bound-of P*) (*bound-of* (*t P*))
  by(*blast intro*:*add-mono mult-left-mono*)
 also have ... = *max* (*bound-of P*) (*bound-of* (*t P*)) **by**(*simp add*:*algebra-simps*)
 finally show «*G*» $s *$ *wp body* (*t P*) $s + (1 -$ «*G*» $s) * P\ s \leq$
    max (*bound-of P*) (*bound-of* (*t P*)) **.**
qed

This gives the equivalence with the alternative definition for loops[McIver and Morgan, 2004, §7, p. 198, footnote 23].

lemma *wlp-Loop1*:
 fixes *body* :: *'s prog*
 assumes *unitary*: *unitary P*
  and *healthy*: *nearly-healthy* (*wlp body*)
 shows *wlp* (*do G* $\longrightarrow$ *body od*) *P* =
 gfp-exp ($\lambda Q\ s.$ «*G*» $s *$ *wlp body Q s* + «$\mathcal{N}$ *G*» $s * P\ s$)
 (**is** *?X* = *gfp-exp* (*?Y P*))
proof –
 let *?Z u* = (*body* ;; *Embed u* $_{«\ G\ »}\oplus$ *Skip*)
 show *?thesis*
 proof(*simp only*: *wp-eval*, *intro gfp-pulldown assms le-funI*)
  fix *u P*
  show *wlp* (*?Z u*) *P* = *?Y P* (*u P*) **by**(*simp add*:*wp-eval negate-embed*)
 next
  fix *t*::*'s trans* **and** *P*::*'s expect*
  assume *ut*: $\bigwedge Q.$ *unitary Q* $\Longrightarrow$ *unitary* (*t Q*) **and** *uP*: *unitary P*
  thus *unitary* (*wlp* (*?Z t*) *P*)
   by(*rule wlp-loop-step-unitary*[*OF healthy*])
 next
  fix *P Q*::*'s expect*
  assume *uP*: *unitary P* **and** *uQ*: *unitary Q*
  show *unitary* ($\lambda a.$ « *G* » $a *$ *wlp body Q a* + « $\mathcal{N}$ *G* » $a * P\ a$)

```
    proof(intro unitaryI2 nnegI bounded-byI)
      fix s::'s
      from healthy uQ
      have unitary (wlp body Q) by(auto)
      hence 0 ≤ wlp body Q s by(auto)
      with uP show 0 ≤ «G» s ∗ wlp body Q s + «𝒩 G» s ∗ P s
        by(auto intro!:add-nonneg-nonneg mult-nonneg-nonneg)

      from healthy uQ have bounded-by 1 (wlp body Q) by(auto)
      with uP have «G» s ∗ wlp body Q s + (1 − «G» s) ∗ P s ≤ «G» s ∗ 1 + (1 − «G» s)
∗ 1
        by(blast intro:add-mono mult-left-mono)
      also have ... = 1 by(simp)
      finally show «G» s ∗ wlp body Q s + «𝒩 G» s ∗ P s ≤ 1
        by(simp add:negate-embed)
    qed
  next
    fix P Q R::'s expect and s::'s
    assume uP: unitary P and uQ: unitary Q and uR: unitary R
       and le: Q ⊩ R
    hence wlp body Q s ≤ wlp body R s
      by(blast intro:le-funD[OF nearly-healthy-monoD, OF healthy])
    thus «G» s ∗ wlp body Q s + «𝒩 G» s ∗ P s ≤
           «G» s ∗ wlp body R s + «𝒩 G» s ∗ P s
      by(auto intro:mult-left-mono)
  next
    fix t u::'s trans
    assume le-utrans t u
           ⋀P. unitary P ⟹ unitary (t P)
           ⋀P. unitary P ⟹ unitary (u P)
    thus le-utrans (wlp (?Z t)) (wlp (?Z u))
      by(blast intro!:wlp-loop-step-mono[OF healthy])
  qed
qed

lemma wp-loop-sound:
  assumes sP: sound P
      and hb: healthy (wp body)
  shows sound (wp do G ⟶ body od P)
proof(simp only: wp-eval, intro lfp-trans-sound sP)
  let ?v = λP s. bound-of P
  show le-trans (wp (body ;; Embed ?v « G »⊕ Skip)) ?v
    by(intro le-transI, simp add:wp-eval lfp-loop-fp[unfolded negate-embed] hb)
  show ⋀P. sound P ⟹ sound (?v P) by(auto)
qed
```

Likewise, we can rewrite strict loops.

```
lemma wp-Loop1:
  fixes body :: 's prog
```

```
  assumes sP: sound P
    and healthy: healthy (wp body)
  shows wp (do G ⟶ body od) P =
   lfp-exp (λQ s. «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s)
  (is ?X = lfp-exp (?Y P))
proof −
  let ?Z u = (body ;; Embed u « G »⊕ Skip)
  show ?thesis
  proof(simp only: wp-eval, intro lfp-pulldown assms le-funI sP mono-transI)
    fix u P
    show wp (?Z u) P = ?Y P (u P) by(simp add:wp-eval negate-embed)
  next
    fix t::'s trans and P::'s expect
    assume ut: ⋀Q. sound Q ⟹ sound (t Q) and uP: sound P
    with healthy show sound (wp (?Z t) P) by(rule wp-loop-step-sound)
  next
    fix P Q::'s expect
    assume sP: sound P and sQ: sound Q
    show sound (λa. « G » a ∗ wp body Q a + « 𝒩 G » a ∗ P a)
    proof(intro soundI2 nnegI bounded-byI)
      fix s::'s
      from sQ have nneg Q bounded-by (bound-of Q) Q by(auto)
      with healthy have bounded-by (bound-of Q) (wp body Q) by(auto)
      hence wp body Q s ≤ bound-of Q by(auto)
      hence wp body Q s ≤ max (bound-of P) (bound-of Q) by(auto)
      moreover {
        from sP have P s ≤ bound-of P by(auto)
        hence P s ≤ max (bound-of P) (bound-of Q) by(auto)
      }
      ultimately have «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s ≤
                  «G» s ∗ max (bound-of P) (bound-of Q) +
                  «𝒩 G» s ∗ max (bound-of P) (bound-of Q)
        by(auto intro!:add-mono mult-left-mono)
      also have ... = max (bound-of P) (bound-of Q) by(simp add:algebra-simps negate-embed)
      finally show «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s ≤ max (bound-of P) (bound-of Q)
.

      from sP have 0 ≤ P s by(auto)
      moreover from sQ healthy have 0 ≤ wp body Q s by(auto)
      ultimately show 0 ≤ «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s
        by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)
    qed
  next
    fix P Q R::'s expect and s::'s
    assume sQ: sound Q and sR: sound R
      and le: Q ⊩ R
    hence wp body Q s ≤ wp body R s
      by(blast intro:le-funD[OF mono-transD, OF healthy-monoD, OF healthy])
    thus «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s ≤
```

```
      «G» s ∗ wp body R s + «𝒩 G» s ∗ P s
    by(auto intro:mult-left-mono)
  next
    fix t u::'s trans
    assume le: le-trans t u
       and st: ⋀P. sound P ⟹ sound (t P)
       and su: ⋀P. sound P ⟹ sound (u P)
    with healthy show le-trans (wp (?Z t)) (wp (?Z u))
      by(rule wp-loop-step-mono)
  next
    from healthy show le-trans (wp (?Z (λP s. bound-of P))) (λP s. bound-of P)
      by(intro le-transI, simp add:wp-eval lfp-loop-fp[unfolded negate-embed])
  next
    fix P::'s expect and s::'s
    assume sound P
    thus sound (λs. bound-of P) by(auto)
  qed
qed

lemma nearly-healthy-wlp-loop:
  fixes body::'s prog
  assumes hb: nearly-healthy (wlp body)
  shows nearly-healthy (wlp (do G ⟶ body od))
proof(intro nearly-healthyI unitaryI2 nnegI2 bounded-byI2, simp-all add:wlp-Loop1 hb)
  fix P::'s expect
  assume uP: unitary P
  let ?X R = λQ s. « G » s ∗ wlp body Q s + « 𝒩 G » s ∗ R s

  show λs. 0 ⊩ gfp-exp (?X P)
  proof(rule gfp-exp-upperbound)
    show unitary (λs. 0::real) by(auto)
    with hb have unitary (wlp body (λs. 0)) by(auto)
    with uP show λs. 0 ⊩ (?X P (λs. 0))
      by(blast intro!:le-funI add-nonneg-nonneg mult-nonneg-nonneg)
  qed

  show gfp-exp (?X P) ⊩ λs. 1
  proof(rule gfp-exp-least)
    show unitary (λs. 1::real) by(auto)
    fix Q::'s expect
    assume unitary Q
    thus Q ⊩ λs. 1 by(auto)
  qed

  fix Q::'s expect
  assume uQ: unitary Q and le: P ⊩ Q
  show gfp-exp (?X P) ⊩ gfp-exp (?X Q)
  proof(rule gfp-exp-least)
    fix R::'s expect assume uR: unitary R
```

```
    assume fp: R ⊩ ?X P R
    also from le have ... ⊩ ?X Q R
      by(blast intro:add-mono mult-left-mono le-funI)
    finally show R ⊩ gfp-exp (?X Q)
      using uR by(auto intro:gfp-exp-upperbound)
  next
    show unitary (gfp-exp (?X Q))
    proof(rule gfp-exp-unitary, intro unitaryI2 nnegI bounded-byI)
      fix R::'s expect and s::'s assume uR: unitary R
      with hb have ubP: unitary (wlp body R) by(auto)
      with uQ show 0 ≤ « G » s ∗ wlp body R s + « 𝒩 G » s ∗ Q s
        by(blast intro:add-nonneg-nonneg mult-nonneg-nonneg)

      from ubP uQ have wlp body R s ≤ 1 Q s ≤ 1 by(auto)
      hence « G » s ∗ wlp body R s + « 𝒩 G » s ∗ Q s ≤ «G» s ∗ 1 + «𝒩 G» s ∗ 1
        by(blast intro:add-mono mult-left-mono)
      thus « G » s ∗ wlp body R s + « 𝒩 G » s ∗ Q s ≤ 1
        by(simp add:negate-embed)
    qed
  qed
qed
```

We show healthiness by appealing to the properties of expectation fixed points, applied to the alternative loop definition.

```
lemma healthy-wp-loop:
  fixes body::'s prog
  assumes hb: healthy (wp body)
  shows healthy (wp (do G ⟶ body od))
proof −
  let ?X P = (λQ s. «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s)
  show ?thesis
  proof(intro healthy-parts bounded-byI2 nnegI2, simp-all add:wp-Loop1 hb soundI2 sound-intros)
    fix P::'s expect and c::real and s::'s
    assume sP: sound P and nnc: 0 ≤ c
    show c ∗ (lfp-exp (?X P)) s = lfp-exp (?X (λs. c ∗ P s)) s
    proof(cases)
      assume c = 0 thus ?thesis
      proof(simp, intro antisym)
        from hb have fp: λs. «G» s ∗ wp body (λ-. 0) s ⊩ λs. 0 by(simp)
        hence lfp-exp (λP s. «G» s ∗ wp body P s) ⊩ λs. 0
          by(auto intro:lfp-exp-lowerbound)
        thus lfp-exp (λP s. «G» s ∗ wp body P s) s ≤ 0 by(auto)
        have λs. 0 ⊩ lfp-exp (λP s. «G» s ∗ wp body P s)
          by(auto intro:lfp-exp-greatest fp)
        thus 0 ≤ lfp-exp (λP s. «G» s ∗ wp body P s) s by(auto)
      qed
    next
      have onesided: ⋀P c. c ≠ 0 ⟹ 0 ≤ c ⟹ sound P ⟹
          λa. c ∗ lfp-exp (λa b. «G» b ∗ wp body a b + «𝒩 G» b ∗ P b) a ⊩
```

```
          lfp-exp (λa b. «G» b ∗ wp body a b + «𝒩 G» b ∗ (c ∗ P b))
proof −
  fix P::'s expect and c::real
  assume cnz: c ≠ 0 and nnc: 0 ≤ c and sP: sound P
  with nnc have cpos: 0 < c by(auto)
  hence nnic: 0 ≤ inverse c by(auto)
  show λa. c ∗ lfp-exp (λa b. «G» b ∗ wp body a b + «𝒩 G» b ∗ P b) a ⊩
      lfp-exp (λa b. «G» b ∗ wp body a b + «𝒩 G» b ∗ (c ∗ P b))
  proof(rule lfp-exp-greatest)
    fix Q::'s expect
    assume sQ: sound Q
      and fp: λb. «G» b ∗ wp body Q b + «𝒩 G» b ∗ (c ∗ P b) ⊩ Q
    hence ⋀s. «G» s ∗ wp body Q s + «𝒩 G» s ∗ (c ∗ P s) ≤ Q s by(auto)
    with nnic
    have ⋀s. inverse c ∗ («G» s ∗ wp body Q s + «𝒩 G» s ∗ (c ∗ P s)) ≤
            inverse c ∗ Q s
      by(auto intro:mult-left-mono)
    hence ⋀s. «G» s ∗ (inverse c ∗ wp body Q s) + (inverse c ∗ c) ∗ «𝒩 G» s ∗ P s ≤
            inverse c ∗ Q s
      by(simp add:algebra-simps)
    hence ⋀s. «G» s ∗ wp body (λs. inverse c ∗ Q s) s + «𝒩 G» s ∗ P s ≤
            inverse c ∗ Q s
      by(simp add:cnz scalingD[OF healthy-scalingD, OF hb sQ nnic])
    hence λs. «G» s ∗ wp body (λs. inverse c ∗ Q s) s + «𝒩 G» s ∗ P s ⊩
        λs. inverse c ∗ Q s by(rule le-funI)
    moreover from nnic sQ have sound (λs. inverse c ∗ Q s)
      by(iprover intro:sound-intros)
    ultimately have lfp-exp (λa b. «G» b ∗ wp body a b + «𝒩 G» b ∗ P b) ⊩
                λs. inverse c ∗ Q s
      by(rule lfp-exp-lowerbound)
    hence ⋀s. lfp-exp (λa b. «G» b ∗ wp body a b + «𝒩 G» b ∗ P b) s ≤ inverse c ∗ Q s
      by(rule le-funD)
    with nnc
    have ⋀s. c ∗ lfp-exp (λa b. «G» b ∗ wp body a b + «𝒩 G» b ∗ P b) s ≤
            c ∗ (inverse c ∗ Q s)
      by(auto intro:mult-left-mono)
    also from cnz have ⋀s. ... s = Q s by(simp)
    finally show λa. c ∗ lfp-exp (λa b. «G» b ∗ wp body a b + «𝒩 G» b ∗ P b) a ⊩ Q
      by(rule le-funI)
  next
    from sP have sound (λs. bound-of P) by(auto)
    with hb sP have sound (lfp-exp (?X P))
      by(blast intro:lfp-exp-sound lfp-loop-fp)
    with nnc show sound (λs. c ∗ lfp-exp (?X P) s)
      by(auto intro!:sound-intros)

    from hb sP nnc
    show λs. «G» s ∗ wp body (λs. bound-of (λs. c ∗ P s)) s +
          «𝒩 G» s ∗ (c ∗ P s) ⊩ λs. bound-of (λs. c ∗ P s)
```

```
        by(iprover intro:lfp-loop-fp sound-intros)

      from sP nnc show sound (λs. bound-of (λs. c ∗ P s))
        by(auto intro!:sound-intros)
    qed
  qed

  assume nzc: c ≠ 0
  show ?thesis (is ?X P c s = ?Y P c s)
  proof(rule fun-cong[where x=s], rule antisym)
    from nzc nnc sP show ?X P c ⊩ ?Y P c by(rule onesided)

    from nzc have nzic: inverse c ≠ 0 by(auto)
    moreover with nnc have nnic: 0 ≤ inverse c by(auto)
    moreover from nnc sP have scP: sound (λs. c ∗ P s) by(auto intro!:sound-intros)
    ultimately have ?X (λs. c ∗ P s) (inverse c) ⊩ ?Y (λs. c ∗ P s) (inverse c)
      by(rule onesided)
    with nnc have λs. c ∗ ?X (λs. c ∗ P s) (inverse c) s ⊩
            λs. c ∗ ?Y (λs. c ∗ P s) (inverse c) s
      by(blast intro:mult-left-mono)
    with nzc show ?Y P c ⊩ ?X P c by(simp add:mult.assoc[symmetric])
  qed
 qed
next
 fix P::'s expect and b::real
 assume bP: bounded-by b P and nP: nneg P
 show lfp-exp (λQ s. «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s) ⊩ λs. b
 proof(intro lfp-exp-lowerbound le-funI)
  fix s::'s
  from bP nP hb have bounded-by b (wp body (λs. b)) by(auto)
  hence wp body (λs. b) s ≤ b by(auto)
  moreover from bP have P s ≤ b by(auto)
  ultimately have «G» s ∗ wp body (λs. b) s + «𝒩 G» s ∗ P s ≤ «G» s ∗ b + «𝒩 G» s
∗ b
    by(auto intro!:add-mono mult-left-mono)
  also have ... = b by(simp add:negate-embed field-simps)
  finally show «G» s ∗ wp body (λs. b) s + «𝒩 G» s ∗ P s ≤ b .
  from bP nP have 0 ≤ b by(auto)
  thus sound (λs. b) by(auto)
 qed
 from hb bP nP show λs. 0 ⊩ lfp-exp (λQ s. «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s)
  by(auto dest!:sound-nneg intro!:lfp-loop-greatest)
next
 fix P Q::'s expect
 assume sP: sound P and sQ: sound Q and le: P ⊩ Q
 show lfp-exp (?X P) ⊩ lfp-exp (?X Q)
 proof(rule lfp-exp-greatest)
  fix R::'s expect
  assume sR: sound R
```

and *fp*: $\lambda s.$ «*G*» *s* ∗ *wp body R s* + «$\mathcal{N}$ *G*» *s* ∗ *Q s* ⊩ *R*
from *le* **have** $\lambda s.$ «*G*» *s* ∗ *wp body R s* + «$\mathcal{N}$ *G*» *s* ∗ *P s* ⊩
$\lambda s.$ «*G*» *s* ∗ *wp body R s* + «$\mathcal{N}$ *G*» *s* ∗ *Q s*
by(*auto intro*:*le-funI add-left-mono mult-left-mono*)
also note *fp*
finally show *lfp-exp* (λR *s*. «*G*» *s* ∗ *wp body R s* + «$\mathcal{N}$ *G*» *s* ∗ *P s*) ⊩ *R*
using *sR* **by**(*auto intro*:*lfp-exp-lowerbound*)
next
from *hb sP* **show** *sound* (*lfp-exp* (λR *s*. «*G*» *s* ∗ *wp body R s* + «$\mathcal{N}$ *G*» *s* ∗ *P s*))
by(*rule lfp-loop-sound*)
from *hb sQ* **show** $\lambda s.$ «*G*» *s* ∗ *wp body* ($\lambda s.$ *bound-of Q*) *s* + «$\mathcal{N}$ *G*» *s* ∗ *Q s* ⊩ $\lambda s.$ *bound-of Q*
by(*rule lfp-loop-fp*)
from *sQ* **show** *sound* ($\lambda s.$ *bound-of Q*) **by**(*auto*)
qed
qed
qed

Use 'simp add:healthy_intros' or 'blast intro:healthy_intros' as appropriate to discharge healthiness side-contitions for primitive programs automatically.

lemmas *healthy-intros* =
healthy-wp-Abort nearly-healthy-wlp-Abort healthy-wp-Skip nearly-healthy-wlp-Skip
healthy-wp-Seq nearly-healthy-wlp-Seq healthy-wp-PC nearly-healthy-wlp-PC
healthy-wp-DC nearly-healthy-wlp-DC healthy-wp-AC nearly-healthy-wlp-AC
healthy-wp-Embed nearly-healthy-wlp-Embed healthy-wp-Apply nearly-healthy-wlp-Apply
healthy-wp-SetDC nearly-healthy-wlp-SetDC healthy-wp-SetPC nearly-healthy-wlp-SetPC
healthy-wp-Bind nearly-healthy-wlp-Bind healthy-wp-repeat nearly-healthy-wlp-repeat
healthy-wp-loop nearly-healthy-wlp-loop

end

4.3 Continuity

theory *Continuity* **imports** *Healthiness* **begin**

We rely on one additional healthiness property, continuity, which is shown here seperately, as its proof relies, in general, on healthiness. It is only relevant when a program appears in an inductive context i.e. inside a loop.

A continuous transformer preserves limits (or the suprema of ascending chains).

definition *bd-cts* :: *'s trans* ⇒ *bool*
where *bd-cts t* = (∀ *M*. (∀ *i*. (*M i* ⊩ *M* (*Suc i*)) ∧ *sound* (*M i*)) ⟶
(∃ *b*. ∀ *i*. *bounded-by b* (*M i*)) ⟶
t (*Sup-exp* (*range M*)) = *Sup-exp* (*range* (*t o M*)))

lemma *bd-ctsD*:
⟦ *bd-cts t*; ⋀*i*. *M i* ⊩ *M* (*Suc i*); ⋀*i*. *sound* (*M i*); ⋀*i*. *bounded-by b* (*M i*) ⟧ ⟹
t (*Sup-exp* (*range M*)) = *Sup-exp* (*range* (*t o M*))

unfolding *bd-cts-def* **by**(*auto*)

lemma *bd-ctsI*:
$(\bigwedge b\, M.\ (\bigwedge i.\ M\ i \Vdash M\ (Suc\ i)) \Longrightarrow (\bigwedge i.\ sound\ (M\ i)) \Longrightarrow (\bigwedge i.\ bounded\text{-}by\ b\ (M\ i)) \Longrightarrow$
$t\ (Sup\text{-}exp\ (range\ M)) = Sup\text{-}exp\ (range\ (t\ o\ M))) \Longrightarrow bd\text{-}cts\ t$
unfolding *bd-cts-def* **by**(*auto*)

A generalised property for transformers of transformers.

definition *bd-cts-tr* :: $('s\ trans \Rightarrow 's\ trans) \Rightarrow bool$
where $bd\text{-}cts\text{-}tr\ T = (\forall M.\ (\forall i.\ le\text{-}trans\ (M\ i)\ (M\ (Suc\ i)) \wedge feasible\ (M\ i)) \longrightarrow$
$equiv\text{-}trans\ (T\ (Sup\text{-}trans\ (M\ `\ UNIV)))\ (Sup\text{-}trans\ ((T\ o\ M)\ `\ UNIV)))$

lemma *bd-cts-trD*:
$[\![\ bd\text{-}cts\text{-}tr\ T;\ \bigwedge i.\ le\text{-}trans\ (M\ i)\ (M\ (Suc\ i));\ \bigwedge i.\ feasible\ (M\ i)\ ]\!] \Longrightarrow$
$equiv\text{-}trans\ (T\ (Sup\text{-}trans\ (M\ `\ UNIV)))\ (Sup\text{-}trans\ ((T\ o\ M)\ `\ UNIV))$
by(*simp add*:*bd-cts-tr-def*)

lemma *bd-cts-trI*:
$(\bigwedge M.\ (\bigwedge i.\ le\text{-}trans\ (M\ i)\ (M\ (Suc\ i))) \Longrightarrow (\bigwedge i.\ feasible\ (M\ i)) \Longrightarrow$
$equiv\text{-}trans\ (T\ (Sup\text{-}trans\ (M\ `\ UNIV)))\ (Sup\text{-}trans\ ((T\ o\ M)\ `\ UNIV))) \Longrightarrow bd\text{-}cts\text{-}tr$
T
by(*simp add*:*bd-cts-tr-def*)

4.3.1 Continuity of Primitives

lemma *cts-wp-Abort*:
$bd\text{-}cts\ (wp\ (Abort::'s\ prog))$
proof −
have X: $range\ (\lambda(i::nat)\ (s::'s).\ 0) = \{\lambda s.\ 0\}$ **by**(*auto*)
show *?thesis* **by**(*intro bd-ctsI*, *simp add*:*wp-eval o-def Sup-exp-def X*)
qed

lemma *cts-wp-Skip*:
$bd\text{-}cts\ (wp\ Skip)$
by(*rule bd-ctsI*, *simp add*:*wp-def Skip-def o-def*)

lemma *cts-wp-Apply*:
$bd\text{-}cts\ (wp\ (Apply\ f))$
proof −
have X: $\bigwedge M\ s.\ \{P\ (f\ s)\ |P.\ P \in range\ M\} = \{P\ s\ |P.\ P \in range\ (\lambda i\ s.\ M\ i\ (f\ s))\}$ **by**(*auto*)
show *?thesis* **by**(*intro bd-ctsI ext*, *simp add*:*wp-eval o-def Sup-exp-def X*)
qed

lemma *cts-wp-Bind*:
fixes $a::'a \Rightarrow 's\ prog$
assumes ca: $\bigwedge s.\ bd\text{-}cts\ (wp\ (a\ (f\ s)))$
shows $bd\text{-}cts\ (wp\ (Bind\ f\ a))$
proof(*rule bd-ctsI*)
fix $M::nat \Rightarrow 's\ expect$ **and** $c::real$

assume *chain*: $\bigwedge i.\ M\ i \Vdash M\ (Suc\ i)$ **and** *sM*: $\bigwedge i.\ sound\ (M\ i)$
and *bM*: $\bigwedge i.\ bounded\text{-}by\ c\ (M\ i)$
with *bd-ctsD*[*OF ca*]
have $\bigwedge s.\ wp\ (a\ (f\ s))\ (Sup\text{-}exp\ (range\ M)) =$
$Sup\text{-}exp\ (range\ (wp\ (a\ (f\ s))\ o\ M))$
by(*auto*)
moreover have $\bigwedge s.\ \{fa\ s\ |fa.\ fa \in range\ (\lambda x.\ wp\ (a\ (f\ s))\ (M\ x))\} =$
$\{fa\ s\ |fa.\ fa \in range\ (\lambda x\ s.\ wp\ (a\ (f\ s))\ (M\ x)\ s)\}$
by(*auto*)
ultimately show $wp\ (Bind\ f\ a)\ (Sup\text{-}exp\ (range\ M)) =$
$Sup\text{-}exp\ (range\ (wp\ (Bind\ f\ a) \circ M))$
by(*simp add*:*wp-eval o-def Sup-exp-def*)
qed

The first nontrivial proof. We transform the suprema into limits, and appeal to the continuity of the underlying operation (here infimum). This is typical of the remainder of the nonrecursive elements.

lemma *cts-wp-DC*:
fixes *a b*::*'s prog*
assumes *ca*: *bd-cts* (*wp a*)
and *cb*: *bd-cts* (*wp b*)
and *ha*: *healthy* (*wp a*)
and *hb*: *healthy* (*wp b*)
shows *bd-cts* $(wp\ (a \sqcap b))$
proof(*rule bd-ctsI*, *rule antisym*)
fix M::$nat \Rightarrow$ *'s expect* **and** *c*::*real*
assume *chain*: $\bigwedge i.\ M\ i \Vdash M\ (Suc\ i)$ **and** *sM*: $\bigwedge i.\ sound\ (M\ i)$
and *bM*: $\bigwedge i.\ bounded\text{-}by\ c\ (M\ i)$

from *ha hb* **have** *hab*: $healthy\ (wp\ (a \sqcap b))$ **by**(*rule healthy-intros*)
from *bM* **have** *leSup*: $\bigwedge i.\ M\ i \Vdash Sup\text{-}exp\ (range\ M)$ **by**(*auto intro*:*Sup-exp-upper*)
from *sM bM* **have** *sSup*: $sound\ (Sup\text{-}exp\ (range\ M))$ **by**(*auto intro*:*Sup-exp-sound*)

show $Sup\text{-}exp\ (range\ (wp\ (a \sqcap b) \circ M)) \Vdash wp\ (a \sqcap b)\ (Sup\text{-}exp\ (range\ M))$
proof(*rule Sup-exp-least*, *clarsimp*, *rule le-funI*)
fix *i s*
from *mono-transD*[*OF healthy-monoD*[*OF hab*]] *leSup sM sSup*
have $wp\ (a \sqcap b)\ (M\ i) \Vdash wp\ (a \sqcap b)\ (Sup\text{-}exp\ (range\ M))$ **by**(*auto*)
thus $wp\ (a \sqcap b)\ (M\ i)\ s \le wp\ (a \sqcap b)\ (Sup\text{-}exp\ (range\ M))\ s$ **by**(*auto*)

from *hab sSup* **have** $sound\ (wp\ (a \sqcap b)\ (Sup\text{-}exp\ (range\ M)))$ **by**(*auto*)
thus $nneg\ (wp\ (a \sqcap b)\ (Sup\text{-}exp\ (range\ M)))$ **by**(*auto*)
qed

from *sM bM ha* **have** $\bigwedge i.\ bounded\text{-}by\ c\ (wp\ a\ (M\ i))$ **by**(*auto*)
hence *baM*: $\bigwedge i\ s.\ wp\ a\ (M\ i)\ s \le c$ **by**(*auto*)
from *sM bM hb* **have** $\bigwedge i.\ bounded\text{-}by\ c\ (wp\ b\ (M\ i))$ **by**(*auto*)
hence *bbM*: $\bigwedge i\ s.\ wp\ b\ (M\ i)\ s \le c$ **by**(*auto*)

show *wp* $(a \sqcap b)$ *(Sup-exp (range M))* $\Vdash$ *Sup-exp (range (wp* $(a \sqcap b) \circ M$*))*
proof(*simp add*:*wp-eval o-def*, *rule le-funI*)
  fix *s*::′*s*
  from *bd-ctsD*[*OF ca*, *of M*, *OF chain sM bM*] *bd-ctsD*[*OF cb*, *of M*, *OF chain sM bM*]
  have *min (wp a (Sup-exp (range M)) s) (wp b (Sup-exp (range M)) s)* =
    min (Sup-exp (range (wp a o M)) s) (Sup-exp (range (wp b o M)) s) **by**(*simp*)
  also {
   have $\{f\,s \mid f.\, f \in$ *range* $(\lambda x.$ *wp a (M x))*$\}$ = *range* $(\lambda i.$ *wp a (M i) s)*
     $\{f\,s \mid f.\, f \in$ *range* $(\lambda x.$ *wp b (M x))*$\}$ = *range* $(\lambda i.$ *wp b (M i) s)*
    by(*auto*)
   hence *min (Sup-exp (range (wp a o M)) s) (Sup-exp (range (wp b o M)) s)* =
     min (Sup (range $(\lambda i.$ *wp a (M i) s))) (Sup (range* $(\lambda i.$ *wp b (M i) s)))*
    by(*simp add*:*Sup-exp-def o-def*)
  }
  also {
   have $(\lambda i.$ *wp a (M i) s)* $\longrightarrow$ *Sup (range* $(\lambda i.$ *wp a (M i) s))*
   proof(*rule increasing-LIMSEQ*)
    fix *n*
    from *mono-transD*[*OF healthy-monoD*, *OF ha*] *sM chain*
    show *wp a (M n) s* $\leq$ *wp a (M (Suc n)) s* **by**(*auto intro*:*le-funD*)
    from *baM* **show** *wp a (M n) s* $\leq$ *Sup (range* $(\lambda i.$ *wp a (M i) s))*
     by(*intro cSup-upper bdd-aboveI*, *auto*)

    fix *e*::*real* **assume** *pe*: $0 < e$
    from *baM* **have** *cSup*: *Sup (range* $(\lambda i.$ *wp a (M i) s))* $\in$ *closure (range* $(\lambda i.$ *wp a (M
i) s))*
     by(*blast intro*:*closure-contains-Sup*)
    with *pe* **obtain** *y* **where** *yin*: $y \in$ *(range* $(\lambda i.$ *wp a (M i) s))*
         and *dy*: *dist y (Sup (range* $(\lambda i.$ *wp a (M i) s)))* $< e$
     by(*blast dest*:*iffD1*[*OF closure-approachable*])
    from *yin* **obtain** *i* **where** $y =$ *wp a (M i) s* **by**(*auto*)
    with *dy* **have** *dist (wp a (M i) s) (Sup (range* $(\lambda i.$ *wp a (M i) s)))* $< e$
     by(*simp*)
    moreover from *baM* **have** *wp a (M i) s* $\leq$ *Sup (range* $(\lambda i.$ *wp a (M i) s))*
     by(*intro cSup-upper bdd-aboveI*, *auto*)
    ultimately have *Sup (range* $(\lambda i.$ *wp a (M i) s))* $\leq$ *wp a (M i) s* $+ e$
     by(*simp add*:*dist-real-def*)
    thus $\exists\, i.$ *Sup (range* $(\lambda i.$ *wp a (M i) s))* $\leq$ *wp a (M i) s* $+ e$ **by**(*auto*)
   qed
   moreover
   have $(\lambda i.$ *wp b (M i) s)* $\longrightarrow$ *Sup (range* $(\lambda i.$ *wp b (M i) s))*
   proof(*rule increasing-LIMSEQ*)
    fix *n*
    from *mono-transD*[*OF healthy-monoD*, *OF hb*] *sM chain*
    show *wp b (M n) s* $\leq$ *wp b (M (Suc n)) s* **by**(*auto intro*:*le-funD*)
    from *bbM* **show** *wp b (M n) s* $\leq$ *Sup (range* $(\lambda i.$ *wp b (M i) s))*
     by(*intro cSup-upper bdd-aboveI*, *auto*)

    fix *e*::*real* **assume** *pe*: $0 < e$

```
      from bbM have cSup: Sup (range (λi. wp b (M i) s)) ∈ closure (range (λi. wp b (M
i) s))
        by(blast intro:closure-contains-Sup)
      with pe obtain y where yin: y ∈ (range (λi. wp b (M i) s))
                and dy: dist y (Sup (range (λi. wp b (M i) s))) < e
        by(blast dest:iffD1[OF closure-approachable])
      from yin obtain i where y = wp b (M i) s by(auto)
      with dy have dist (wp b (M i) s) (Sup (range (λi. wp b (M i) s))) < e
        by(simp)
      moreover from bbM have wp b (M i) s ≤ Sup (range (λi. wp b (M i) s))
        by(intro cSup-upper bdd-aboveI, auto)
      ultimately have Sup (range (λi. wp b (M i) s)) ≤ wp b (M i) s + e
        by(simp add:dist-real-def)
      thus ∃ i. Sup (range (λi. wp b (M i) s)) ≤ wp b (M i) s + e by(auto)
    qed
    ultimately have (λi. min (wp a (M i) s) (wp b (M i) s)) ⟶
              min (Sup (range (λi. wp a (M i) s))) (Sup (range (λi. wp b (M i) s)))
      by(rule tendsto-min)
    moreover have bdd-above (range (λi. min (wp a (M i) s) (wp b (M i) s)))
    proof(intro bdd-aboveI, clarsimp)
      fix i
      have min (wp a (M i) s) (wp b (M i) s) ≤ wp a (M i) s by(auto)
      also {
        from ha sM bM have bounded-by c (wp a (M i)) by(auto)
        hence wp a (M i) s ≤ c by(auto)
      }
      finally show min (wp a (M i) s) (wp b (M i) s) ≤ c .
    qed
    ultimately
    have min (Sup (range (λi. wp a (M i) s))) (Sup (range (λi. wp b (M i) s))) ≤
        Sup (range (λi. min (wp a (M i) s) (wp b (M i) s)))
      by(blast intro:LIMSEQ-le-const2 cSup-upper min.mono[OF baM bbM])
  }
  also {
    have range (λi. min (wp a (M i) s) (wp b (M i) s)) =
        {f s |f. f ∈ range (λi s. min (wp a (M i) s) (wp b (M i) s))}
      by(auto)
    hence Sup (range (λi. min (wp a (M i) s) (wp b (M i) s))) =
        Sup-exp (range (λi s. min (wp a (M i) s) (wp b (M i) s))) s
      by (simp add: Sup-exp-def cong del: SUP-cong-simp)
  }
  finally show min (wp a (Sup-exp (range M)) s) (wp b (Sup-exp (range M)) s) ≤
            Sup-exp (range (λi s. min (wp a (M i) s) (wp b (M i) s))) s .
 qed
qed

lemma cts-wp-Seq:
 fixes a b::'s prog
 assumes ca: bd-cts (wp a)
```

```
    and cb: bd-cts (wp b)
    and hb: healthy (wp b)
  shows bd-cts (wp (a ;; b))
proof(rule bd-ctsI, simp add:o-def wp-eval)
  fix M::nat ⇒ 's expect and c::real
  assume chain: ⋀i. M i ⊩ M (Suc i) and sM: ⋀i. sound (M i)
    and bM: ⋀i. bounded-by c (M i)
  hence wp a (wp b (Sup-exp (range M))) = wp a (Sup-exp (range (wp b o M)))
    by(subst bd-ctsD[OF cb], auto)
  also {
    from sM hb have ⋀i. sound ((wp b o M) i) by(auto)
    moreover from chain sM
    have ⋀i. (wp b o M) i ⊩ (wp b o M) (Suc i)
      by(auto intro:mono-transD[OF healthy-monoD, OF hb])
    moreover from sM bM hb have ⋀i. bounded-by c ((wp b o M) i) by(auto)
    ultimately have wp a (Sup-exp (range (wp b o M))) =
                 Sup-exp (range (wp a o (wp b o M)))
      by(subst bd-ctsD[OF ca], auto)
  }
  also have Sup-exp (range (wp a o (wp b o M))) =
           Sup-exp (range (λi. wp a (wp b (M i))))
    by(simp add:o-def)
  finally show wp a (wp b (Sup-exp (range M))) =
             Sup-exp (range (λi. wp a (wp b (M i)))) .
qed

lemma cts-wp-PC:
  fixes a b::'s prog
  assumes ca: bd-cts (wp a)
    and cb: bd-cts (wp b)
    and ha: healthy (wp a)
    and hb: healthy (wp b)
    and up: unitary p
  shows bd-cts (wp (PC a p b))
proof(rule bd-ctsI, rule ext, simp add:o-def wp-eval)
  fix M::nat ⇒ 's expect and c::real and s::'s
  assume chain: ⋀i. M i ⊩ M (Suc i) and sM: ⋀i. sound (M i)
    and bM: ⋀i. bounded-by c (M i)

  from sM have ⋀i. nneg (M i) by(auto)
  with bM have nc: 0 ≤ c by(auto)

  from chain sM bM have wp a (Sup-exp (range M)) = Sup-exp (range (wp a o M))
    by(rule bd-ctsD[OF ca])
  hence wp a (Sup-exp (range M)) s = Sup-exp (range (wp a o M)) s
    by(simp)
  also {
    have {f s |f. f ∈ range (λx. wp a (M x))} = range (λi. wp a (M i) s)
      by(auto)
```

```
  hence Sup-exp (range (wp a o M)) s = Sup (range (λi. wp a (M i) s))
    by(simp add:Sup-exp-def o-def)
}
finally have p s ∗ wp a (Sup-exp (range M)) s =
          p s ∗ Sup (range (λi. wp a (M i) s)) by(simp)
also have ... = Sup {p s ∗ x |x. x ∈ range (λi. wp a (M i) s)}
proof(rule cSup-mult, blast, clarsimp)
  from up show 0 ≤ p s by(auto)
  fix i
  from sM bM ha have bounded-by c (wp a (M i)) by(auto)
  thus wp a (M i) s ≤ c by(auto)
qed
also {
  have {p s ∗ x |x. x ∈ range (λi. wp a (M i) s)} = range (λi. p s ∗ wp a (M i) s)
    by(auto)
  hence Sup {p s ∗ x |x. x ∈ range (λi. wp a (M i) s)} =
        Sup (range (λi. p s ∗ wp a (M i) s)) by(simp)
}
finally have p s ∗ wp a (Sup-exp (range M)) s = Sup (range (λi. p s ∗ wp a (M i) s)) .
moreover {
  from chain sM bM have wp b (Sup-exp (range M)) = Sup-exp (range (wp b o M))
    by(rule bd-ctsD[OF cb])
  hence wp b (Sup-exp (range M)) s = Sup-exp (range (wp b o M)) s
    by(simp)
  also {
    have {f s |f. f ∈ range (λx. wp b (M x))} = range (λi. wp b (M i) s)
      by(auto)
    hence Sup-exp (range (wp b o M)) s = Sup (range (λi. wp b (M i) s))
      by(simp add:Sup-exp-def o-def)
  }
  finally have (1 − p s) ∗ wp b (Sup-exp (range M)) s =
            (1 − p s) ∗ Sup (range (λi. wp b (M i) s)) by(simp)
  also have ... = Sup {(1 − p s) ∗ x |x. x ∈ range (λi. wp b (M i) s)}
  proof(rule cSup-mult, blast, clarsimp)
    from up show 0 ≤ 1 − p s
      by auto
    fix i
    from sM bM hb have bounded-by c (wp b (M i)) by(auto)
    thus wp b (M i) s ≤ c by(auto)
  qed
  also {
    have {(1 − p s) ∗ x |x. x ∈ range (λi. wp b (M i) s)} =
        range (λi. (1 − p s) ∗ wp b (M i) s)
      by(auto)
    hence Sup {(1 − p s) ∗ x |x. x ∈ range (λi. wp b (M i) s)} =
          Sup (range (λi. (1 − p s) ∗ wp b (M i) s)) by(simp)
  }
  finally have (1 − p s) ∗ wp b (Sup-exp (range M)) s =
            Sup (range (λi. (1 − p s) ∗ wp b (M i) s)) .
```

```
}
ultimately
have p s ∗ wp a (Sup-exp (range M)) s + (1 − p s) ∗ wp b (Sup-exp (range M)) s =
   Sup (range (λi. p s ∗ wp a (M i) s)) + Sup (range (λi. (1 − p s) ∗ wp b (M i) s))
 by(simp)
also {
 from bM sM ha have ⋀i. bounded-by c (wp a (M i)) by(auto)
 hence ⋀i. wp a (M i) s ≤ c by(auto)
 moreover from up have 0 ≤ p s by(auto)
 ultimately have ⋀i. p s ∗ wp a (M i) s ≤ p s ∗ c by(auto intro:mult-left-mono)
 also from up nc have p s ∗ c ≤ 1 ∗ c by(blast intro:mult-right-mono)
 also have ... = c by(simp)
 finally have baM: ⋀i. p s ∗ wp a (M i) s ≤ c .

 have lima: (λi. p s ∗ wp a (M i) s) ⟶ Sup (range (λi. p s ∗ wp a (M i) s))
 proof(rule increasing-LIMSEQ)
  fix n
  from sM chain healthy-monoD[OF ha] have wp a (M n) ⊩ wp a (M (Suc n))
   by(auto)
  with up show p s ∗ wp a (M n) s ≤ p s ∗ wp a (M (Suc n)) s
   by(blast intro:mult-left-mono)
  from baM show p s ∗ wp a (M n) s ≤ Sup (range (λi. p s ∗ wp a (M i) s))
   by(intro cSup-upper bdd-aboveI, auto)
 next
  fix e::real
  assume pe: 0 < e
  from baM have Sup (range (λi. p s ∗ wp a (M i) s)) ∈
          closure (range (λi. p s ∗ wp a (M i) s))
   by(blast intro:closure-contains-Sup)
  thm closure-approachable
  with pe obtain y where yin: y ∈ range (λi. p s ∗ wp a (M i) s)
            and dy: dist y (Sup (range (λi. p s ∗ wp a (M i) s))) < e
   by(blast dest:iffD1[OF closure-approachable])
  from yin obtain i where y = p s ∗ wp a (M i) s by(auto)
  with dy have dist (p s ∗ wp a (M i) s) (Sup (range (λi. p s ∗ wp a (M i) s))) < e
   by(simp)
  moreover from baM have p s ∗ wp a (M i) s ≤ Sup (range (λi. p s ∗ wp a (M i) s))
   by(intro cSup-upper bdd-aboveI, auto)
  ultimately have Sup (range (λi. p s ∗ wp a (M i) s)) ≤ p s ∗ wp a (M i) s + e
   by(simp add:dist-real-def)
  thus ∃ i. Sup (range (λi. p s ∗ wp a (M i) s)) ≤ p s ∗ wp a (M i) s + e by(auto)
 qed

 from bM sM hb have ⋀i. bounded-by c (wp b (M i)) by(auto)
 hence ⋀i. wp b (M i) s ≤ c by(auto)
 moreover from up have 0 ≤ (1 − p s)
  by auto
 ultimately have ⋀i. (1 − p s) ∗ wp b (M i) s ≤ (1 − p s) ∗ c by(auto intro:mult-left-mono)
 also {
```

```
    from up have 1 − p s ≤ 1 by(auto)
    with nc have (1 − p s) ∗ c ≤ 1 ∗ c by(blast intro:mult-right-mono)
  }
  also have 1 ∗ c = c by(simp)
  finally have bbM: ⋀i. (1 − p s) ∗ wp b (M i) s ≤ c .

  have limb: (λi. (1 − p s) ∗ wp b (M i) s) ⟶ Sup (range (λi. (1 − p s) ∗ wp b (M
i) s))
  proof(rule increasing-LIMSEQ)
    fix n
    from sM chain healthy-monoD[OF hb] have wp b (M n) ⊩ wp b (M (Suc n))
      by(auto)
    moreover from up have 0 ≤ 1 − p s
      by auto
    ultimately show (1 − p s) ∗ wp b (M n) s ≤ (1 − p s) ∗ wp b (M (Suc n)) s
      by(blast intro:mult-left-mono)
    from bbM show (1 − p s) ∗ wp b (M n) s ≤ Sup (range (λi. (1 − p s) ∗ wp b (M i) s))
      by(intro cSup-upper bdd-aboveI, auto)
  next
    fix e::real
    assume pe: 0 < e
    from bbM have Sup (range (λi. (1 − p s) ∗ wp b (M i) s)) ∈
              closure (range (λi. (1 − p s) ∗ wp b (M i) s))
      by(blast intro:closure-contains-Sup)
    with pe obtain y where yin: y ∈ range (λi. (1 − p s) ∗ wp b (M i) s)
                    and dy: dist y (Sup (range (λi. (1 − p s) ∗ wp b (M i) s))) < e
      by(blast dest:iffD1[OF closure-approachable])
    from yin obtain i where y = (1 − p s) ∗ wp b (M i) s by(auto)
    with dy have dist ((1 − p s) ∗ wp b (M i) s)
                    (Sup (range (λi. (1 − p s) ∗ wp b (M i) s))) < e
      by(simp)
    moreover from bbM
    have (1 − p s) ∗ wp b (M i) s ≤ Sup (range (λi. (1 − p s) ∗ wp b (M i) s))
      by(intro cSup-upper bdd-aboveI, auto)
    ultimately have Sup (range (λi. (1 − p s) ∗ wp b (M i) s)) ≤ (1 − p s) ∗ wp b (M i) s
+ e
      by(simp add:dist-real-def)
     thus ∃ i. Sup (range (λi. (1 − p s) ∗ wp b (M i) s)) ≤ (1 − p s) ∗ wp b (M i) s + e
by(auto)
  qed

  from lima limb have (λi. p s ∗ wp a (M i) s + (1 − p s) ∗ wp b (M i) s) ⟶
    Sup (range (λi. p s ∗ wp a (M i) s)) + Sup (range (λi. (1 − p s) ∗ wp b (M i) s))
    by(rule tendsto-add)
  moreover from add-mono[OF baM bbM]
  have ⋀i. p s ∗ wp a (M i) s + (1 − p s) ∗ wp b (M i) s ≤
                Sup (range (λi. p s ∗ wp a (M i) s + (1 − p s) ∗ wp b (M i) s))
    by(intro cSup-upper bdd-aboveI, auto)
  ultimately have Sup (range (λi. p s ∗ wp a (M i) s)) +
```

```
        Sup (range (λi. (1 − p s) ∗ wp b (M i) s)) ≤
        Sup (range (λi. p s ∗ wp a (M i) s + (1 − p s) ∗ wp b (M i) s))
  by(blast intro: LIMSEQ-le-const2)
}
also {
 have range (λi. p s ∗ wp a (M i) s + (1 − p s) ∗ wp b (M i) s) =
     {f s |f. f ∈ range (λx s. p s ∗ wp a (M x) s + (1 − p s) ∗ wp b (M x) s)}
  by(auto)
 hence Sup (range (λi. p s ∗ wp a (M i) s + (1 − p s) ∗ wp b (M i) s)) =
      Sup-exp (range (λx s. p s ∗ wp a (M x) s + (1 − p s) ∗ wp b (M x) s)) s
  by (simp add: Sup-exp-def cong del: SUP-cong-simp)
}
finally
have p s ∗ wp a (Sup-exp (range M)) s + (1 − p s) ∗ wp b (Sup-exp (range M)) s ≤
    Sup-exp (range (λi s. p s ∗ wp a (M i) s + (1 − p s) ∗ wp b (M i) s)) s .
moreover
have Sup-exp (range (λi s. p s ∗ wp a (M i) s + (1 − p s) ∗ wp b (M i) s)) s ≤
    p s ∗ wp a (Sup-exp (range M)) s + (1 − p s) ∗ wp b (Sup-exp (range M)) s
proof(rule le-funD[OF Sup-exp-least], clarsimp, rule le-funI)
 fix i::nat and s::'s
 from bM have leSup: M i ⊩ Sup-exp (range M)
  by(blast intro: Sup-exp-upper)
 moreover from sM bM have sSup: sound (Sup-exp (range M))
  by(auto intro:Sup-exp-sound)
 moreover note healthy-monoD[OF ha] sM
 ultimately have wp a (M i) ⊩ wp a (Sup-exp (range M)) by(auto)
 hence wp a (M i) s ≤ wp a (Sup-exp (range M)) s by(auto)
 moreover {
  from leSup sSup healthy-monoD[OF hb] sM
  have wp b (M i) ⊩ wp b (Sup-exp (range M)) by(auto)
  hence wp b (M i) s ≤ wp b (Sup-exp (range M)) s by(auto)
 }
 moreover from up have 0 ≤ p s 0 ≤ 1 − p s
  by auto
 ultimately
 show p s ∗ wp a (M i) s + (1 − p s) ∗ wp b (M i) s ≤
     p s ∗ wp a (Sup-exp (range M)) s + (1 − p s) ∗ wp b (Sup-exp (range M)) s
  by(blast intro:add-mono mult-left-mono)

 from sSup ha hb have sound (wp a (Sup-exp (range M)))
            sound (wp b (Sup-exp (range M)))
  by(auto)
 hence ⋀s. 0 ≤ wp a (Sup-exp (range M)) s ⋀s. 0 ≤ wp b (Sup-exp (range M)) s
  by(auto)
 moreover from up have ⋀s. 0 ≤ p s ⋀s. 0 ≤ 1 − p s
  by auto
 ultimately show nneg (λc. p c ∗ wp a (Sup-exp (range M)) c +
            (1 − p c) ∗ wp b (Sup-exp (range M)) c)
  by(blast intro:add-nonneg-nonneg mult-nonneg-nonneg)
```

qed
ultimately show *p s* $*$ *wp a* (*Sup-exp* (*range M*)) *s* + (*1* − *p s*) $*$ *wp b* (*Sup-exp* (*range M*)) *s* =
Sup-exp (*range* (λ*x s*. *p s* $*$ *wp a* (*M x*) *s* + (*1* − *p s*) $*$ *wp b* (*M x*) *s*)) *s*
by(*auto*)
qed

Both set-based choice operators are only continuous for finite sets (probabilistic choice *can* be extended infinitely, but we have not done so). The proofs for both are inductive, and rely on the above results on binary operators.

lemma *SetPC-Bind*:
SetPC a p = *Bind p* (λ*p*. *SetPC a* (λ-. *p*))
by(*intro ext*, *simp add*:*SetPC-def Bind-def Let-def*)

lemma *SetPC-remove*:
assumes *nz*: *p x* ≠ *0* **and** *n1*: *p x* ≠ *1*
and *fsupp*: *finite* (*supp p*)
shows *SetPC a* (λ-. *p*) = *PC* (*a x*) (λ-. *p x*) (*SetPC a* (λ-. *dist-remove p x*))
proof(*intro ext*, *simp add*:*SetPC-def PC-def*)
fix *ab P s*
from *nz* **have** *x* ∈ *supp p* **by**(*simp add*:*supp-def*)
hence *supp p* = *insert x* (*supp p* − {*x*}) **by**(*auto*)
hence (∑*x*∈*supp p*. *p x* $*$ *a x ab P s*) =
(∑*x*∈*insert x* (*supp p* − {*x*}). *p x* $*$ *a x ab P s*)
by(*simp*)
also from *fsupp*
have ... = *p x* $*$ *a x ab P s* + (∑*x*∈*supp p* − {*x*}. *p x* $*$ *a x ab P s*)
by(*blast intro*:*sum.insert*)
also from *n1*
have ... = *p x* $*$ *a x ab P s* + (*1* − *p x*) $*$ ((∑*x*∈*supp p* − {*x*}. *p x* $*$ *a x ab P s*) / (*1* − *p x*))
by(*simp add*:*field-simps*)
also have ... = *p x* $*$ *a x ab P s* +
(*1* − *p x*) $*$ ((∑*y*∈*supp p* − {*x*}. (*p y* / (*1* − *p x*)) $*$ *a y ab P s*))
by(*simp add*:*sum-divide-distrib*)
also have ... = *p x* $*$ *a x ab P s* +
(*1* − *p x*) $*$ ((∑*y*∈*supp p* − {*x*}. *dist-remove p x y* $*$ *a y ab P s*))
by(*simp add*:*dist-remove-def*)
also from *nz n1*
have ... = *p x* $*$ *a x ab P s* +
(*1* − *p x*) $*$ ((∑*y*∈*supp* (*dist-remove p x*). *dist-remove p x y* $*$ *a y ab P s*))
by(*simp add*:*supp-dist-remove*)
finally show (∑*x*∈*supp p*. *p x* $*$ *a x ab P s*) =
p x $*$ *a x ab P s* +
(*1* − *p x*) $*$ (∑*y*∈*supp* (*dist-remove p x*). *dist-remove p x y* $*$ *a y ab P s*) **.**
qed

lemma *cts-bot*:
bd-cts (λ(*P*::′*s expect*) (*s*::′*s*). *0*::*real*)

```
proof −
  have X: ⋀s::'s. {(P::'s expect) s |P. P ∈ range (λP s. 0)} = {0} by(auto)
  show ?thesis by(intro bd-ctsI, simp add:Sup-exp-def o-def X)
qed

lemma wp-SetPC-nil:
  wp (SetPC a (λs a. 0)) = (λP s. 0)
  by(intro ext, simp add:wp-eval)

lemma SetPC-sgl:
  supp p = {x} ⟹ SetPC a (λ-. p) = (λab P s. p x ∗ a x ab P s)
  by(simp add:SetPC-def)

lemma bd-cts-scale:
  fixes a::'s trans
  assumes ca: bd-cts a
     and ha: healthy a
     and nnc: 0 ≤ c
  shows bd-cts (λP s. c ∗ a P s)
proof(intro bd-ctsI ext, simp add:o-def)
  fix M::nat ⇒ 's expect and d::real and s::'s
  assume chain: ⋀i. M i ⊩ M (Suc i) and sM: ⋀i. sound (M i)
    and bM: ⋀i. bounded-by d (M i)

  from sM have ⋀i. nneg (M i) by(auto)
  with bM have nnd: 0 ≤ d by(auto)

  from sM bM have sSup: sound (Sup-exp (range M)) by(auto intro:Sup-exp-sound)
  with healthy-scalingD[OF ha] nnc
  have c ∗ a (Sup-exp (range M)) s = a (λs. c ∗ Sup-exp (range M) s) s
   by(auto intro:scalingD)
  also {
   have ⋀s. {f s |f. f ∈ range M} = range (λi. M i s) by(auto)
   hence a (λs. c ∗ Sup-exp (range M) s) s =
        a (λs. c ∗ Sup (range (λi. M i s))) s
     by(simp add:Sup-exp-def)
  }
  also {
   from bM have ⋀x s. x ∈ range (λi. M i s) ⟹ x ≤ d by(auto)
   with nnc have a (λs. c ∗ Sup (range (λi. M i s))) s =
              a (λs. Sup {c∗x |x. x ∈ range (λi. M i s)}) s
     by(subst cSup-mult, blast+)
  }
  also {
   have X: ⋀s. {c ∗ x |x. x ∈ range (λi. M i s)} = range (λi. c ∗ M i s) by(auto)
   have a (λs. Sup {c ∗ x |x. x ∈ range (λi. M i s)}) s =
       a (λs. Sup (range (λi. c ∗ M i s))) s by(simp add:X)
  }
  also {
```

  have ⋀*s*. *range* (λ*i*. *c* ∗ *M i s*) = {*f s* |*f*. *f* ∈ *range* (λ*i s*. *c* ∗ *M i s*)}
   by(*auto*)
  hence (λ*s*. *Sup* (*range* (λ*i*. *c* ∗ *M i s*))) = *Sup-exp* (*range* (λ*i s*. *c* ∗ *M i s*))
   by (*simp add*: *Sup-exp-def cong del*: *SUP-cong-simp*)
  hence *a* (λ*s*. *Sup* (*range* (λ*i*. *c* ∗ *M i s*))) *s* =
     a (*Sup-exp* (*range* (λ*i s*. *c* ∗ *M i s*))) *s* **by**(*simp*)
 }
 also {
  from *le-funD*[*OF chain*] *nnc*
  have ⋀*i*. (λ*s*. *c* ∗ *M i s*) ⊩ (λ*s*. *c* ∗ *M* (*Suc i*) *s*)
   by(*auto intro*:*le-funI*[*OF mult-left-mono*])
  moreover from *sM nnc*
  have ⋀*i*. *sound* (λ*s*. *c* ∗ *M i s*)
   by(*auto intro*:*sound-intros*)
  moreover from *bM nnc*
  have ⋀*i*. *bounded-by* (*c* ∗ *d*) (λ*s*. *c* ∗ *M i s*)
   by(*auto intro*:*mult-left-mono*)
  ultimately
  have *a* (*Sup-exp* (*range* (λ*i s*. *c* ∗ *M i s*))) =
     Sup-exp (*range* (*a o* (λ*i s*. *c* ∗ *M i s*)))
   by(*rule bd-ctsD*[*OF ca*])
  hence *a* (*Sup-exp* (*range* (λ*i s*. *c* ∗ *M i s*))) *s* =
     Sup-exp (*range* (*a o* (λ*i s*. *c* ∗ *M i s*))) *s*
   by(*auto*)
 }
 also have *Sup-exp* (*range* (*a o* (λ*i s*. *c* ∗ *M i s*))) *s* =
      Sup-exp (*range* (λ*x*. *a* (λ*s*. *c* ∗ *M x s*))) *s*
  by(*simp add*:*o-def*)
 also {
  from *nnc sM*
  have ⋀*x*. *a* (λ*s*. *c* ∗ *M x s*) = (λ*s*. *c* ∗ *a* (*M x*) *s*)
   by(*auto intro*:*scalingD*[*OF healthy-scalingD*, *OF ha*, *symmetric*])
  hence *Sup-exp* (*range* (λ*x*. *a* (λ*s*. *c* ∗ *M x s*))) *s* =
     Sup-exp (*range* (λ*x s*. *c* ∗ *a* (*M x*) *s*)) *s*
   by(*simp*)
 }
 finally show *c* ∗ *a* (*Sup-exp* (*range M*)) *s* = *Sup-exp* (*range* (λ*x s*. *c* ∗ *a* (*M x*) *s*)) *s* **.**
qed

lemma *cts-wp-SetPC-const*:
 fixes *a*::′*a* ⇒ ′*s prog*
 assumes *ca*: ⋀*x*. *x* ∈ (*supp p*) ⟹ *bd-cts* (*wp* (*a x*))
   and *ha*: ⋀*x*. *x* ∈ (*supp p*) ⟹ *healthy* (*wp* (*a x*))
   and *up*: *unitary p*
   and *sump*: *sum p* (*supp p*) ≤ *1*
   and *fsupp*: *finite* (*supp p*)
 shows *bd-cts* (*wp* (*SetPC a* (λ-. *p*)))
proof(*cases supp p* = {}, *simp add*:*supp-empty SetPC-def wp-def cts-bot*)
 assume *nesupp*: *supp p* ≠ {}

```
from fsupp have unitary p ⟶ sum p (supp p) ≤ 1 ⟶
         (∀x∈supp p. bd-cts (wp (a x))) ⟶
         (∀x∈supp p. healthy (wp (a x))) ⟶
         bd-cts (wp (SetPC a (λ-. p)))
proof(induct supp p arbitrary:p, simp add:supp-empty wp-SetPC-nil cts-bot, clarify)
 fix x::'a and F::'a set and p::'a ⇒ real
 assume fF: finite F
 assume insert x F = supp p
 hence pstep: supp p = insert x F by(simp)
 hence xin: x ∈ supp p by(auto)
 assume up: unitary p and ca: ∀x∈supp p. bd-cts (wp (a x))
   and ha: ∀x∈supp p. healthy (wp (a x))
   and sump: sum p (supp p) ≤ 1
   and xni: x ∉ F
 assume IH: ⋀p. F = supp p ⟹
          unitary p ⟶ sum p (supp p) ≤ 1 ⟶
          (∀x∈supp p. bd-cts (wp (a x))) ⟶
          (∀x∈supp p. healthy (wp (a x))) ⟶
          bd-cts (wp (SetPC a (λ-. p)))

 from fF pstep have fsupp: finite (supp p) by(auto)

 from xin have nzp: p x ≠ 0 by(simp add:supp-def)

 have xy-le-sum:
   ⋀y. y ∈ supp p ⟹ y ≠ x ⟹ p x + p y ≤ sum p (supp p)
 proof −
   fix y assume yin: y ∈ supp p and yne: y ≠ x
   from up have 0 ≤ sum p (supp p − {x,y})
     by(auto intro:sum-nonneg)
   hence p x + p y ≤ p x + p y + sum p (supp p − {x,y})
     by(auto)
 also {
   from yin yne fsupp
   have p y + sum p (supp p − {x,y}) = sum p (supp p − {x})
     by(subst sum.insert[symmetric], (blast intro!:sum.cong)+)
   moreover
   from xin fsupp
   have p x + sum p (supp p − {x}) = sum p (supp p)
     by(subst sum.insert[symmetric], (blast intro!:sum.cong)+)
   ultimately
   have p x + p y + sum p (supp p − {x, y}) = sum p (supp p) by(simp)
 }
 finally show p x + p y ≤ sum p (supp p) .
 qed

 have n1p: ⋀y. y ∈ supp p ⟹ y ≠ x ⟹ p x ≠ 1
 proof(rule ccontr, simp)
   assume px1: p x = 1
```

```
  fix y assume yin: y ∈ supp p and yne: y ≠ x
  from up have 0 ≤ p y by(auto)
  with yin have 0 < p y by(auto simp:supp-def)
  hence 0 + p x < p y + p x by(rule add-strict-right-mono)
  with px1 have 1 < p x + p y by(simp)
  also from yin yne have p x + p y ≤ sum p (supp p)
    by(rule xy-le-sum)
  finally show False using sump by(simp)
qed

show bd-cts (wp (SetPC a (λ-. p)))
proof(cases F = {})
  case True with pstep have supp p = {x} by(simp)
  hence wp (SetPC a (λ-. p)) = (λP s. p x ∗ wp (a x) P s)
    by(simp add:SetPC-sgl wp-def)
  moreover {
    from up ca ha xin have bd-cts (wp (a x)) healthy (wp (a x)) 0 ≤ p x
      by(auto)
    hence bd-cts (λP s. p x ∗ wp (a x) P s)
      by(rule bd-cts-scale)
  }
  ultimately show ?thesis by(simp)
next
  assume neF: F ≠ {}
  then obtain y where yinF: y ∈ F by(auto)
  with xni have yne: y ≠ x by(auto)
  from yinF pstep have yin: y ∈ supp p by(auto)

  from supp-dist-remove[of p x, OF nzp n1p, OF yin yne]
  have supp-sub: supp (dist-remove p x) ⊆ supp p by(auto)

  from xin ca have cax: bd-cts (wp (a x)) by(auto)
  from xin ha have hax: healthy (wp (a x)) by(auto)

  from supp-sub ha have hra: ∀ x∈supp (dist-remove p x). healthy (wp (a x))
    by(auto)
  from supp-sub ca have cra: ∀ x∈supp (dist-remove p x). bd-cts (wp (a x))
    by(auto)

  from supp-dist-remove[of p x, OF nzp n1p, OF yin yne] pstep xni
  have Fsupp: F = supp (dist-remove p x)
    by(simp)

  have udp: unitary (dist-remove p x)
  proof(intro unitaryI2 nnegI bounded-byI)
    fix y
    show 0 ≤ dist-remove p x y
    proof(cases y=x, simp-all add:dist-remove-def)
      from up have 0 ≤ p y 0 ≤ 1 − p x
```

```
      by auto
    thus 0 ≤ p y / (1 − p x)
      by(rule divide-nonneg-nonneg)
  qed
  show dist-remove p x y ≤ 1
  proof(cases y=x, simp-all add:dist-remove-def,
      cases y∈supp p, simp-all add:nsupp-zero)
    assume yne: y ≠ x and yin: y ∈ supp p
    hence p x + p y ≤ sum p (supp p)
      by(auto intro:xy-le-sum)
    also note sump
    finally have p y ≤ 1 − p x by(auto)
    moreover from up have p x ≤ 1 by(auto)
    moreover from yin yne have p x ≠ 1 by(rule n1p)
    ultimately show p y / (1 − p x) ≤ 1 by(auto)
  qed
qed

from xin have pxn0: p x ≠ 0 by(auto simp:supp-def)
from yin yne have pxn1: p x ≠ 1 by(rule n1p)

from pxn0 pxn1 have sum (dist-remove p x) (supp (dist-remove p x)) =
          sum (dist-remove p x) (supp p − {x})
  by(simp add:supp-dist-remove)
also have ... = (∑ y∈supp p − {x}. p y / (1 − p x))
  by(simp add:dist-remove-def)
also have ... = (∑ y∈supp p − {x}. p y) / (1 − p x)
  by(simp add:sum-divide-distrib)
also {
  from xin have insert x (supp p) = supp p by(auto)
  with fsupp have p x + (∑ y∈supp p − {x}. p y) = sum p (supp p)
    by(simp add:sum.insert[symmetric])
  also note sump
  finally have sum p (supp p − {x}) ≤ 1 − p x by(auto)
  moreover {
    from up have p x ≤ 1 by(auto)
    with pxn1 have p x < 1 by(auto)
    hence 0 < 1 − p x by(auto)
  }
  ultimately have sum p (supp p − {x}) / (1 − p x) ≤ 1
    by(auto)
}
finally have sdp: sum (dist-remove p x) (supp (dist-remove p x)) ≤ 1 .

from Fsupp udp sdp hra cra IH
have cts-dr: bd-cts (wp (SetPC a (λ-. dist-remove p x)))
  by(auto)

from up have upx: unitary (λ-. p x) by(auto)
```

from *pxn0 pxn1 fsupp hra* **show** *?thesis*
by(*simp add*:*SetPC-remove*,
blast intro:*cts-wp-PC cax cts-dr hax healthy-intros*
unitary-sound[*OF udp*] *sdp upx*)
qed
qed
with *assms* **show** *?thesis* **by**(*auto*)
qed

lemma *cts-wp-SetPC*:
fixes a::$'a \Rightarrow\ 's$ *prog*
assumes *ca*: $\bigwedge x\ s.\ x \in (supp\ (p\ s)) \Longrightarrow bd\text{-}cts\ (wp\ (a\ x))$
and *ha*: $\bigwedge x\ s.\ x \in (supp\ (p\ s)) \Longrightarrow healthy\ (wp\ (a\ x))$
and *up*: $\bigwedge s.\ unitary\ (p\ s)$
and *sump*: $\bigwedge s.\ sum\ (p\ s)\ (supp\ (p\ s)) \leq 1$
and *fsupp*: $\bigwedge s.\ finite\ (supp\ (p\ s))$
shows *bd-cts* (*wp* (*SetPC a p*))
proof −
from *assms* **have** *bd-cts* (*wp* (*Bind p* (λ*p*. *SetPC a* (λ-. *p*))))
by(*iprover intro*!:*cts-wp-Bind cts-wp-SetPC-const*)
thus *?thesis* **by**(*simp add*:*SetPC-Bind*[*symmetric*])
qed

lemma *wp-SetDC-Bind*:
SetDC a S = *Bind S* (λ*S*. *SetDC a* (λ-. *S*))
by(*intro ext*, *simp add*:*SetDC-def Bind-def*)

lemma *SetDC-finite-insert*:
assumes *fS*: *finite S*
and *neS*: $S \neq \{\}$
shows *SetDC a* (λ-. *insert x S*) = *a x* $\sqcap$ *SetDC a* (λ-. *S*)
proof (*intro ext*, *simp add*: *SetDC-def DC-def cong del*: *image-cong-simp cong add*: *INF-cong-simp*)
fix *ab P s*
from *fS* **have** *A*: *finite* (*insert* (*a x ab P s*) ((λ*x*. *a x ab P s*) ‘ *S*))
and *B*: *finite* (((λ*x*. *a x ab P s*) ‘ *S*)) **by**(*auto*)
from *neS* **have** *C*: *insert* (*a x ab P s*) ((λ*x*. *a x ab P s*) ‘ *S*) ≠ {}
and *D*: (λ*x*. *a x ab P s*) ‘ *S* ≠ {} **by**(*auto*)
from *A C* **have** *Inf* (*insert* (*a x ab P s*) ((λ*x*. *a x ab P s*) ‘ *S*)) =
Min (*insert* (*a x ab P s*) ((λ*x*. *a x ab P s*) ‘ *S*))
by(*auto intro*:*cInf-eq-Min*)
also from *B D* **have** ... = *min* (*a x ab P s*) (*Min* ((λ*x*. *a x ab P s*) ‘ *S*))
by(*auto intro*:*Min-insert*)
also from *B D* **have** ... = *min* (*a x ab P s*) (*Inf* ((λ*x*. *a x ab P s*) ‘ *S*))
by(*simp add*:*cInf-eq-Min*)
finally show (*INF x*∈*insert x S*. *a x ab P s*) =
min (*a x ab P s*) (*INF x*∈*S*. *a x ab P s*)
by (*simp cong del*: *INF-cong-simp*)
qed

```
lemma SetDC-singleton:
  SetDC a (λ-. {x}) = a x
  by (simp add: SetDC-def cong del: INF-cong-simp)

lemma cts-wp-SetDC-const:
  fixes a::'a ⇒ 's prog
  assumes ca: ⋀x. x ∈ S ⟹ bd-cts (wp (a x))
      and ha: ⋀x. x ∈ S ⟹ healthy (wp (a x))
      and fS: finite S
      and neS: S ≠ {}
  shows bd-cts (wp (SetDC a (λ-. S)))
proof −
  have finite S ⟹ S ≠ {} ⟹
      (∀x∈S. bd-cts (wp (a x))) ⟶
      (∀x∈S. healthy (wp (a x))) ⟶
      bd-cts (wp (SetDC a (λ-. S)))
  proof(induct S rule:finite-induct, simp, clarsimp)
    fix x::'a and F::'a set
    assume fF: finite F
       and IH: F ≠ {} ⟹ bd-cts (wp (SetDC a (λ-. F)))
       and cax: bd-cts (wp (a x))
       and hax: healthy (wp (a x))
       and haF: ∀x∈F. healthy (wp (a x))
    show bd-cts (wp (SetDC a (λ-. insert x F)))
    proof(cases F = {}, simp add:SetDC-singleton cax)
      assume F ≠ {}
      with fF cax hax haF IH show bd-cts (wp (SetDC a (λ-. insert x F)))
       by(auto intro!:cts-wp-DC healthy-intros simp:SetDC-finite-insert)
    qed
  qed
  with assms show ?thesis by(auto)
qed

lemma cts-wp-SetDC:
  fixes a::'a ⇒ 's prog
  assumes ca: ⋀x s. x ∈ S s ⟹ bd-cts (wp (a x))
      and ha: ⋀x s. x ∈ S s ⟹ healthy (wp (a x))
      and fS: ⋀s. finite (S s)
      and neS: ⋀s. S s ≠ {}
  shows bd-cts (wp (SetDC a S))
proof −
  from assms have bd-cts (wp (Bind S (λS. SetDC a (λ-. S))))
    by(iprover intro!:cts-wp-Bind cts-wp-SetDC-const)
  thus ?thesis by(simp add:wp-SetDC-Bind[symmetric])
qed

lemma cts-wp-repeat:
  bd-cts (wp a) ⟹ healthy (wp a) ⟹ bd-cts (wp (repeat n a))
```

```
  by(induct n, auto intro:cts-wp-Skip cts-wp-Seq healthy-intros)

lemma cts-wp-Embed:
  bd-cts t ⟹ bd-cts (wp (Embed t))
  by(simp add:wp-eval)
```

4.3.2 Continuity of a Single Loop Step

A single loop iteration is continuous, in the more general sense defined above for transformer transformers.

```
lemma cts-wp-loopstep:
  fixes body::'s prog
  assumes hb: healthy (wp body)
     and cb: bd-cts (wp body)
  shows bd-cts-tr (λx. wp (body ;; Embed x «G»⊕ Skip)) (is bd-cts-tr ?F)
proof(rule bd-cts-trI, rule le-trans-antisym)
  fix M::nat ⇒ 's trans and b::real
  assume chain: ⋀i. le-trans (M i) (M (Suc i))
    and fM:    ⋀i. feasible (M i)
  show fw: le-trans (Sup-trans (range (?F o M))) (?F (Sup-trans (range M)))
  proof(rule le-transI[OF Sup-trans-least2], clarsimp)
    fix P Q::'s expect and t
    assume sP: sound P
    assume nQ: nneg Q and bP: bounded-by (bound-of P) Q
    hence sQ: sound Q by(auto)

    from fM have fSup: feasible (Sup-trans (range M))
      by(auto intro:feasible-Sup-trans)

    from sQ fM have M t Q ⊩ Sup-trans (range M) Q
      by(auto intro:Sup-trans-upper2)
    moreover from sQ fM fSup
      have sMtP: sound (M t Q) sound (Sup-trans (range M) Q) by(auto)
    ultimately have wp body (M t Q) ⊩ wp body (Sup-trans (range M) Q)
      using healthy-monoD[OF hb] by(auto)
    hence ⋀s. wp body (M t Q) s ≤ wp body (Sup-trans (range M) Q) s
      by(rule le-funD)
    thus ?F (M t) Q ⊩ ?F (Sup-trans (range M)) Q
      by(intro le-funI, simp add:wp-eval mult-left-mono)

    show nneg (wp (body ;; Embed (Sup-trans (range M)) «G»⊕ Skip) Q)
    proof(rule nnegI, simp add:wp-eval)
      fix s::'s
        from fSup sQ have sound (Sup-trans (range M) Q) by(auto)
        with hb have sound (wp body (Sup-trans (range M) Q)) by(auto)
        hence 0 ≤ wp body (Sup-trans (range M) Q) s by(auto)
        moreover from sQ have 0 ≤ Q s by(auto)
        ultimately show 0 ≤ «G» s ∗ wp body (Sup-trans (range M) Q) s + (1 − «G» s) ∗
Q s
```

```
      by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)
  qed
next
  fix P::'s expect assume sP: sound P
  thus nneg P bounded-by (bound-of P) P by(auto)
  show ∀ u∈range ((λx. wp (body ;; Embed x « G »⊕ Skip)) ∘ M).
        ∀ R. nneg R ∧ bounded-by (bound-of P) R ⟶
          nneg (u R) ∧ bounded-by (bound-of P) (u R)
  proof(clarsimp, intro conjI nnegI bounded-byI, simp-all add:wp-eval)
    fix u::nat and R::'s expect and s::'s
    assume nR: nneg R and bR: bounded-by (bound-of P) R
    hence sR: sound R by(auto)
    with fM have sMuR: sound (M u R) by(auto)
    with hb have sound (wp body (M u R)) by(auto)
    hence 0 ≤ wp body (M u R) s by(auto)
    moreover from nR have 0 ≤ R s by(auto)
    ultimately show 0 ≤ «G» s ∗ wp body (M u R) s + (1 − «G» s) ∗ R s
      by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)

    from sR bR fM have bounded-by (bound-of P) (M u R) by(auto)
    with sMuR hb have bounded-by (bound-of P) (wp body (M u R)) by(auto)
    hence wp body (M u R) s ≤ bound-of P by(auto)
    moreover from bR have R s ≤ bound-of P by(auto)
    ultimately have «G» s ∗ wp body (M u R) s + (1 − «G» s) ∗ R s ≤
              «G» s ∗ bound-of P + (1 − «G» s) ∗ bound-of P
      by(auto intro:add-mono mult-left-mono)
    also have ... = bound-of P by(simp add:algebra-simps)
    finally show «G» s ∗ wp body (M u R) s + (1 − «G» s) ∗ R s ≤ bound-of P .
  qed
qed

show le-trans (?F (Sup-trans (range M))) (Sup-trans (range (?F o M)))
 proof(rule le-transI, rule le-funI, simp add: wp-eval cong del: image-cong-simp)
  fix P::'s expect and s::'s
  assume sP: sound P
  have {t P |t. t ∈ range M} = range (λi. M i P)
    by(blast)
  hence wp body (Sup-trans (range M) P) s = wp body (Sup-exp (range (λi. M i P))) s
    by(simp add:Sup-trans-def)
  also {
    from sP fM have ⋀i. sound (M i P) by(auto)
    moreover from sP chain have ⋀i. M i P ⊩ M (Suc i) P by(auto)
    moreover {
      from sP have bounded-by (bound-of P) P by(auto)
      with sP fM have ⋀i. bounded-by (bound-of P) (M i P) by(auto)
    }
    ultimately have wp body (Sup-exp (range (λi. M i P))) s =
              Sup-exp (range (λi. wp body (M i P))) s
      by(subst bd-ctsD[OF cb], auto simp:o-def)
```

```
  }
  also have Sup-exp (range (λi. wp body (M i P))) s =
          Sup {f s |f. f ∈ range (λi. wp body (M i P))}
    by(simp add:Sup-exp-def)
  finally have «G» s ∗ wp body (Sup-trans (range M) P) s + (1 − «G» s) ∗ P s =
            «G» s ∗ Sup {f s |f. f ∈ range (λi. wp body (M i P))} + (1 − «G» s) ∗ P s
    by(simp)
  also {
    from sP fM have ⋀i. sound (M i P) by(auto)
    moreover from sP fM have ⋀i. bounded-by (bound-of P) (M i P) by(auto)
    ultimately have ⋀i. bounded-by (bound-of P) (wp body (M i P)) using hb by(auto)
    hence bound: ⋀i. wp body (M i P) s ≤ bound-of P by(auto)
    moreover
    have {« G » s ∗ x |x. x ∈ {f s |f. f ∈ range (λi. wp body (M i P))}} =
        {« G » s ∗ f s |f. f ∈ range (λi. wp body (M i P))}
      by(blast)
    ultimately
    have «G» s ∗ Sup {f s |f. f ∈ range (λi. wp body (M i P))} =
        Sup {«G» s ∗ f s |f. f ∈ range (λi. wp body (M i P))}
      by(subst cSup-mult, auto)
    moreover {
      have {x + (1−«G» s) ∗ P s |x.
          x ∈ {«G» s ∗ f s |f. f ∈ range (λi. wp body (M i P))}} =
          {«G» s ∗ f s + (1−«G» s) ∗ P s |f. f ∈ range (λi. wp body (M i P))}
        by(blast)
      moreover from bound sP have ⋀i. «G» s ∗ wp body (M i P) s ≤ bound-of P
        by(cases G s, auto)
      ultimately
      have Sup {«G» s ∗ f s |f. f ∈ range (λi. wp body (M i P))} + (1−«G» s) ∗ P s =
          Sup {«G» s ∗ f s + (1−«G» s) ∗ P s |f. f ∈ range (λi. wp body (M i P))}
        by(subst cSup-add, auto)
    }
    ultimately
    have «G» s ∗ Sup {f s |f. f ∈ range (λi. wp body (M i P))} + (1−«G» s) ∗ P s =
        Sup {«G» s ∗ f s + (1−«G» s) ∗ P s |f. f ∈ range (λi. wp body (M i P))}
      by(simp)
  }
  also {
    have ⋀i. «G» s ∗ wp body (M i P) s + (1−«G» s) ∗ P s =
          ((λx. wp (body ;; Embed x « G »⊕ Skip)) ∘ M) i P s
      by(simp add:wp-eval)
    also have ⋀i. ... i ≤
            Sup {f s |f. f ∈ {t P |t. t ∈ range ((λx. wp (body ;; Embed x « G »⊕ Skip)) ∘
M)}}
    proof(intro cSup-upper bdd-aboveI, blast, clarsimp simp:wp-eval)
      fix i
      from sP have bP: bounded-by (bound-of P) P by(auto)
      with sP fM have sound (M i P) bounded-by (bound-of P) (M i P) by(auto)
      with hb have bounded-by (bound-of P) (wp body (M i P)) by(auto)
```

```
      with bP have wp body (M i P) s ≤ bound-of P P s ≤ bound-of P by(auto)
      hence «G» s ∗ wp body (M i P) s + (1−«G» s) ∗ P s ≤
            «G» s ∗ (bound-of P) + (1−«G» s) ∗ (bound-of P)
        by(auto intro:add-mono mult-left-mono)
      also have ... = bound-of P by(simp add:algebra-simps)
      finally show «G» s ∗ wp body (M i P) s + (1−«G» s) ∗ P s ≤ bound-of P .
    qed
    finally
    have Sup {«G» s ∗ f s + (1−«G» s) ∗ P s |f. f ∈ range (λi. wp body (M i P))} ≤
          Sup {f s |f. f ∈ {t P |t. t ∈ range ((λx. wp (body ;; Embed x « G »⊕ Skip)) ∘ M)}}
      by(blast intro:cSup-least)
  }
  also have Sup {f s |f. f ∈ {t P |t. t ∈ range ((λx. wp (body ;; Embed x « G »⊕ Skip)) ∘
M)}} =
            Sup-trans (range ((λx. wp (body ;; Embed x « G »⊕ Skip)) ∘ M)) P s
    by(simp add:Sup-trans-def Sup-exp-def)
  finally show «G» s ∗ wp body (Sup-trans (range M) P) s + (1−«G» s) ∗ P s ≤
              Sup-trans (range ((λx. wp (body ;; Embed x « G »⊕ Skip)) ∘ M)) P s .
 qed
qed

end
```

4.4 Continuity and Induction for Loops

theory *LoopInduction* **imports** *Healthiness Continuity* **begin**

Showing continuity for loops requires a stronger induction principle than we have used so far, which in turn relies on the continuity of loops (inductively). Thus, the proofs are intertwined, and broken off from the main set of continuity proofs. This result is also essential in showing the sublinearity of loops.

A loop step is monotonic.

```
lemma wp-loop-step-mono-trans:
 fixes body::'s prog
 assumes sP: sound P
   and hb: healthy (wp body)
 shows mono-trans (λQ s. « G » s ∗ wp body Q s + « 𝒩 G » s ∗ P s)
proof(intro mono-transI le-funI, simp)
 fix Q R::'s expect and s::'s
 assume sQ: sound Q and sR: sound R and le: Q ⊩ R
 hence wp body Q ⊩ wp body R
  by(rule mono-transD[OF healthy-monoD, OF hb])
 thus «G» s ∗ wp body Q s ≤ «G» s ∗ wp body R s
  by(auto dest:le-funD intro:mult-left-mono)
qed
```

We can therefore apply the standard fixed-point lemmas to unfold it:

```
lemma lfp-wp-loop-unfold:
  fixes body::'s prog
  assumes hb: healthy (wp body)
    and sP: sound P
  shows lfp-exp (λQ s. «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s) =
      (λs. «G» s ∗ wp body (lfp-exp (λQ s. «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s)) s +
        «𝒩 G» s ∗ P s)
proof(rule lfp-exp-unfold)
  from assms show mono-trans (λQ s. «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s)
    by(blast intro:wp-loop-step-mono-trans)
  from assms show λs. «G» s ∗ wp body (λs. bound-of P) s + «𝒩 G» s ∗ P s ⊩ λs.
bound-of P
    by(blast intro:lfp-loop-fp)
  from sP show sound (λs. bound-of P)
    by(auto)
  fix Q::'s expect
  assume sound Q
  with assms show sound (λs. «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s)
    by(intro wp-loop-step-sound[unfolded wp-eval, simplified, folded negate-embed], auto)
qed

lemma wp-loop-step-unitary:
  fixes body::'s prog
  assumes hb: healthy (wp body)
    and uP: unitary P and uQ: unitary Q
  shows unitary (λs. «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s)
proof(intro unitaryI2 nnegI bounded-byI)
  fix s::'s
  from uQ hb have uwQ: unitary (wp body Q) by(auto)
  with uP have 0 ≤ wp body Q s 0 ≤ P s by(auto)
  thus 0 ≤ «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s
    by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)

  from uP uwQ have wp body Q s ≤ 1 P s ≤ 1 by(auto)
  hence «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s ≤ «G» s ∗ 1 + «𝒩 G» s ∗ 1
    by(blast intro:add-mono mult-left-mono)
  also have ... = 1 by(simp add:negate-embed)
  finally show «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s ≤ 1 .
qed

lemma lfp-loop-unitary:
  fixes body::'s prog
  assumes hb: healthy (wp body)
    and uP: unitary P
  shows unitary (lfp-exp (λQ s. «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s))
  using assms by(blast intro:lfp-exp-unitary wp-loop-step-unitary)
```

From the lattice structure on transformers, we establish a transfinite induction principle for loops. We use this to show a number of properties, particularly subdistributivity, for loops. This proof follows the pattern of lemma lfp_ordinal_induct in HOL/Inductive.

```
lemma loop-induct:
 fixes body::'s prog
 assumes hwp: healthy (wp body)
   and hwlp: nearly-healthy (wlp body)
   — The body must be healthy, both in strict and liberal semantics.
   and Limit: ⋀S. ⟦ ∀x∈S. P (fst x) (snd x); ∀x∈S. feasible (fst x);
                 ∀x∈S. ∀Q. unitary Q ⟶ unitary (snd x Q) ⟧ ⟹
            P (Sup-trans (fst ‘ S)) (Inf-utrans (snd ‘ S))
   — The property holds at limit points.
   and IH: ⋀t u. ⟦ P t u; feasible t; ⋀Q. unitary Q ⟹ unitary (u Q) ⟧ ⟹
              P (wp (body ;; Embed t «G»⊕ Skip))
               (wlp (body ;; Embed u «G»⊕ Skip))
   — The inductive step. The property is preserved by a single loop iteration.
   and P-equiv: ⋀t t' u u'. ⟦ P t u; equiv-trans t t'; equiv-utrans u u' ⟧ ⟹ P t' u'
   — The property must be preserved by equivalence
 shows P (wp (do G ⟶ body od)) (wlp (do G ⟶ body od))
 — The property can refer to both interpretations simultaneously. The unifier will happily
apply the rule to just one or the other, however.
proof(simp add:wp-eval)
 let ?X t = wp (body ;; Embed t «G»⊕ Skip)
 let ?Y t = wlp (body ;; Embed t «G»⊕ Skip)

 let ?M = {x. P (fst x) (snd x) ∧
          feasible (fst x) ∧
          (∀Q. unitary Q ⟶ unitary (snd x Q)) ∧
          le-trans (fst x) (lfp-trans ?X) ∧
          le-utrans (gfp-trans ?Y) (snd x)}

 have fSup: feasible (Sup-trans (fst ‘ ?M))
 proof(intro feasibleI bounded-byI2 nnegI2)
  fix Q::'s expect and b::real
  assume nQ: nneg Q and bQ: bounded-by b Q
  show Sup-trans (fst ‘ ?M) Q ⊩ λs. b
   unfolding Sup-trans-def
   using nQ bQ by(auto intro!:Sup-exp-least)
  show λs. 0 ⊩ Sup-trans (fst ‘ ?M) Q
  proof(cases)
   assume empty: ?M = {}
   show ?thesis by(simp add:Sup-trans-def Sup-exp-def empty)
  next
   assume ne: ?M ≠ {}
   then obtain x where xin: x ∈ ?M by auto
   hence ffx: feasible (fst x) by(simp)
   with nQ bQ have λs. 0 ⊩ fst x Q by(auto)
   also from xin have fst x Q ⊩ Sup-trans (fst ‘ ?M) Q
     apply(intro Sup-trans-upper2[OF imageI - nQ bQ], assumption)
     apply(clarsimp, blast intro: sound-nneg[OF feasible-sound] feasible-boundedD)
```

```
      done
    finally show λs. 0 ⊩ Sup-trans (fst ‘ ?M) Q .
  qed
qed

have uInf: ⋀P. unitary P ⟹ unitary (Inf-utrans (snd ‘ ?M) P)
proof(cases ?M = {})
  fix P
  assume empty: ?M = {}
  show ?thesis P by(simp only:empty, simp add:Inf-utrans-def)
next
  fix P::'s expect
  assume uP: unitary P
    and ne: ?M ≠ {}
  show ?thesis P
  proof(intro unitaryI2 nnegI2 bounded-byI2)
    from ne obtain x where xin: x ∈ ?M by auto
    hence sxin: snd x ∈ snd ‘ ?M by(simp)
    hence le-utrans (Inf-utrans (snd ‘ ?M)) (snd x)
      by(intro Inf-utrans-lower, auto)
    with uP
    have Inf-utrans (snd ‘ ?M) P ⊩ snd x P by(auto)
    also {
      from xin uP have unitary (snd x P) by(simp)
      hence snd x P ⊩ λs. 1 by(auto)
    }
    finally show Inf-utrans (snd ‘ ?M) P ⊩ λs. 1 .

    have λs. 0 ⊩ Inf-trans (snd ‘ ?M) P
      unfolding Inf-trans-def
    proof(rule Inf-exp-greatest)
      from sxin show {t P |t. t ∈ snd ‘ ?M} ≠ {} by(auto)
      show ∀P∈{t P |t. t ∈ snd ‘ ?M}. λs. 0 ⊩ P
      proof(clarsimp)
        fix t::'s trans
        assume ∀Q. unitary Q ⟶ unitary (t Q)
        with uP have unitary (t P) by(auto)
        thus λs. 0 ⊩ t P by(auto)
      qed
    qed
    also {
      from ne have X: (snd ‘ ?M = {}) = False by(simp)
      have Inf-trans (snd ‘ ?M) P = Inf-utrans (snd ‘ ?M) P
        unfolding Inf-utrans-def by(subst X, simp)
    }
    finally show λs. 0 ⊩ Inf-utrans (snd ‘ ?M) P .
  qed
qed
```

```
have wp-loop-mono: ⋀t u. ⟦ le-trans t u; ⋀P. sound P ⟹ sound (t P);
                  ⋀P. sound P ⟹ sound (u P) ⟧ ⟹ le-trans (?X t) (?X u)
proof(intro le-transI le-funI, simp add:wp-eval)
  fix t u::'s trans and P::'s expect and s::'s
  assume le: le-trans t u
     and st: ⋀P. sound P ⟹ sound (t P)
     and su: ⋀P. sound P ⟹ sound (u P)
     and sP: sound P
  hence sound (t P) sound (u P) by(auto)
  with healthy-monoD[OF hwp] le sP have wp body (t P) ⊩ wp body (u P) by(auto)
  hence wp body (t P) s ≤ wp body (u P) s by(auto)
  thus «G» s * wp body (t P) s ≤ «G» s * wp body (u P) s by(auto intro:mult-left-mono)
qed

have wlp-loop-mono: ⋀t u. ⟦ le-utrans t u; ⋀P. unitary P ⟹ unitary (t P);
                  ⋀P. unitary P ⟹ unitary (u P) ⟧ ⟹ le-utrans (?Y t) (?Y u)
proof(intro le-utransI le-funI, simp add:wp-eval)
  fix t u::'s trans and P::'s expect and s::'s
  assume le: le-utrans t u
     and ut: ⋀P. unitary P ⟹ unitary (t P)
     and uu: ⋀P. unitary P ⟹ unitary (u P)
     and uP: unitary P
  hence unitary (t P) unitary (u P) by(auto)
  with le uP have wlp body (t P) ⊩ wlp body (u P)
    by(auto intro:nearly-healthy-monoD[OF hwlp])
  hence wlp body (t P) s ≤ wlp body (u P) s by(auto)
  thus «G» s * wlp body (t P) s ≤ «G» s * wlp body (u P) s
    by(auto intro:mult-left-mono)
qed

from hwp have hX: ⋀t. healthy t ⟹ healthy (?X t)
  by(auto intro:healthy-intros)

from hwlp have hY: ⋀t. nearly-healthy t ⟹ nearly-healthy (?Y t)
  by(auto intro!:healthy-intros)

have PLimit: P (Sup-trans (fst ` ?M)) (Inf-utrans (snd ` ?M))
  by(auto intro:Limit)

have feasible-lfp-loop:
  feasible (lfp-trans ?X)
proof(intro feasibleI bounded-byI2 nnegI2,
    simp-all add:wp-Loop1[simplified wp-eval] soundI2 hwp)
  fix P::'s expect and b::real
  assume bP: bounded-by b P and nP: nneg P
  hence sP: sound P by(auto)
  show lfp-exp (λQ s. « G » s * wp body Q s + « 𝒩 G » s * P s) ⊩ λs. b
  proof(intro lfp-exp-lowerbound le-funI)
    fix s::'s
```

from *bP nP* **have** *nnb*: *0* ≤ *b* **by**(*auto*)
hence *sound* (λ*s*. *b*) *bounded-by b* (λ*s*. *b*) **by**(*auto*)
with *hwp* **have** *bounded-by b* (*wp body* (λ*s*. *b*)) **by**(*auto*)
with *bP* **have** *wp body* (λ*s*. *b*) *s* ≤ *b P s* ≤ *b* **by**(*auto*)
hence «*G*» *s* ∗ *wp body* (λ*s*. *b*) *s* + «𝒩 *G*» *s* ∗ *P s* ≤ «*G*» *s* ∗ *b* + «𝒩 *G*» *s* ∗ *b*
by(*auto intro*:*add-mono mult-left-mono*)
thus «*G*» *s* ∗ *wp body* (λ*s*. *b*) *s* + «𝒩 *G*» *s* ∗ *P s* ≤ *b*
by(*simp add*:*negate-embed algebra-simps*)
from *nnb* **show** *sound* (λ*s*. *b*) **by**(*auto*)
qed
from *hwp sP* **show** λ*s*. *0* ⊩ *lfp-exp* (λ*Q s*. « *G* » *s* ∗ *wp body Q s* + « 𝒩 *G* » *s* ∗ *P s*)
by(*blast intro*!:*lfp-exp-greatest lfp-loop-fp*)
qed

have *unitary-gfp*:
⋀*P*. *unitary P* ⟹ *unitary* (*gfp-trans ?Y P*)
proof(*intro unitaryI2 nnegI2 bounded-byI2*,
simp-all add:*wlp-Loop1*[*simplified wp-eval*] *hwlp*)
fix *P*::′*s expect*
assume *uP*: *unitary P*
show λ*s*. *0* ⊩ *gfp-exp* (λ*Q s*. « *G* » *s* ∗ *wlp body Q s* + « 𝒩 *G* » *s* ∗ *P s*)
proof(*rule gfp-exp-upperbound*[*OF le-funI*])
fix *s*::′*s*
from *hwlp uP* **have** *0* ≤ *wlp body* (λ*s*. *0*) *s 0* ≤ *P s* **by**(*auto dest*!:*unitary-sound*)
thus *0* ≤ «*G*» *s* ∗ *wlp body* (λ*s*. *0*) *s* + «𝒩 *G*» *s* ∗ *P s*
by(*auto intro*:*add-nonneg-nonneg mult-nonneg-nonneg*)
show *unitary* (λ*s*. *0*) **by**(*auto*)
qed
show *gfp-exp* (λ*Q s*. « *G* » *s* ∗ *wlp body Q s* + « 𝒩 *G* » *s* ∗ *P s*) ⊩ λ*s*. *1*
by(*auto intro*:*gfp-exp-least*)
qed

have *fX*:
⋀*t*. *feasible t* ⟹ *feasible* (*?X t*)
proof(*intro feasibleI nnegI bounded-byI*, *simp-all add*:*wp-eval*)
fix *t*::′*s trans* **and** *Q*::′*s expect* **and** *b*::*real* **and** *s*::′*s*
assume *ft*: *feasible t* **and** *bQ*: *bounded-by b Q* **and** *nQ*: *nneg Q*
hence *nneg* (*t Q*) *bounded-by b* (*t Q*) **by**(*auto*)
moreover hence *stQ*: *sound* (*t Q*) **by**(*auto*)
ultimately have *wp body* (*t Q*) *s* ≤ *b* **using** *hwp* **by**(*auto*)
moreover from *bQ* **have** *Q s* ≤ *b* **by**(*auto*)
ultimately have «*G*» *s* ∗ *wp body* (*t Q*) *s* + (*1* − «*G*» *s*) ∗ *Q s* ≤
«*G*» *s* ∗ *b* + (*1* − « *G* » *s*) ∗ *b*
by(*auto intro*:*add-mono mult-left-mono*)
thus «*G*» *s* ∗ *wp body* (*t Q*) *s* + (*1* − «*G*» *s*) ∗ *Q s* ≤ *b*
by(*simp add*:*algebra-simps*)

from *nQ stQ hwp* **have** *0* ≤ *wp body* (*t Q*) *s 0* ≤ *Q s* **by**(*auto*)
thus *0* ≤ «*G*» *s* ∗ *wp body* (*t Q*) *s* + (*1* − «*G*» *s*) ∗ *Q s*

```
    by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)
  qed

  have uY:
    ⋀t P. (⋀P. unitary P ⟹ unitary (t P)) ⟹ unitary P ⟹ unitary (?Y t P)
  proof(intro unitaryI2 nnegI bounded-byI, simp-all add:wp-eval)
    fix t::'s trans and P::'s expect and s::'s
    assume ut: ⋀P. unitary P ⟹ unitary (t P)
      and uP: unitary P
    hence utP: unitary (t P) by(auto)
    with hwlp have ubtP: unitary (wlp body (t P)) by(auto)
    with uP have 0 ≤ P s 0 ≤ wlp body (t P) s by(auto)
    thus 0 ≤ «G» s ∗ wlp body (t P) s + (1−«G» s) ∗ P s
      by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)

    from uP ubtP have P s ≤ 1 wlp body (t P) s ≤ 1 by(auto)
    hence «G» s ∗ wlp body (t P) s + (1−«G» s) ∗ P s ≤ «G» s ∗ 1 + (1−«G» s) ∗ 1
      by(blast intro:add-mono mult-left-mono)
    also have ... = 1 by(simp add:algebra-simps)
    finally show «G» s ∗ wlp body (t P) s + (1−«G» s) ∗ P s ≤ 1 .
  qed

  have fw-lfp: le-trans (Sup-trans (fst ‘ ?M)) (lfp-trans ?X)
    using feasible-nnegD[OF feasible-lfp-loop]
    by(intro le-transI[OF Sup-trans-least2], blast+)
  hence le-trans (?X (Sup-trans (fst ‘ ?M))) (?X (lfp-trans ?X))
    by(auto intro:wp-loop-mono feasible-sound[OF fSup]
               feasible-sound[OF feasible-lfp-loop])
  also have equiv-trans ... (lfp-trans ?X)
  proof(rule iffD1[OF equiv-trans-comm, OF lfp-trans-unfold], iprover intro:wp-loop-mono)
    fix t::'s trans and P::'s expect
    assume st: ⋀Q. sound Q ⟹ sound (t Q)
      and sP: sound P
    show sound (?X t P)
    proof(intro soundI2 bounded-byI nnegI, simp-all add:wp-eval)
      fix s::'s
      from sP st hwp have 0 ≤ P s 0 ≤ wp body (t P) s by(auto)
      thus 0 ≤ «G» s ∗ wp body (t P) s + (1 − «G» s) ∗ P s
        by(blast intro:add-nonneg-nonneg mult-nonneg-nonneg)
      from sP st have bounded-by (bound-of (t P)) (t P) by(auto)
      with sP st hwp have bounded-by (bound-of (t P)) (wp body (t P)) by(auto)
      hence wp body (t P) s ≤ bound-of (t P) by(auto)
      moreover from sP st hwp have P s ≤ bound-of P by(auto)
      moreover have «G» s ≤ 1 1 − «G» s ≤ 1 by(auto)
      moreover from sP st hwp have 0 ≤ wp body (t P) s 0 ≤ P s by(auto)
      moreover have (0::real) ≤ 1 by(simp)
      ultimately show «G» s ∗ wp body (t P) s + (1 − «G» s) ∗ P s ≤
                 1 ∗ bound-of (t P) + 1 ∗ bound-of P
        by(blast intro:add-mono mult-mono)
```

```
    qed
  next
  let ?fp = λR s. bound-of R
  show le-trans (?X ?fp) ?fp by(auto intro:healthy-intros hwp)
  fix P::'s expect assume sound P
  thus sound (?fp P) by(auto)
 qed
 finally have le-lfp: le-trans (?X (Sup-trans (fst ‘ ?M))) (lfp-trans ?X) .

 have fw-gfp: le-utrans (gfp-trans ?Y) (Inf-utrans (snd ‘ ?M))
  by(auto intro:Inf-utrans-greatest unitary-gfp)

 have equiv-utrans (gfp-trans ?Y) (?Y (gfp-trans ?Y))
  by(auto intro!:gfp-trans-unfold wlp-loop-mono uY)
 also from fw-gfp have le-utrans (?Y (gfp-trans ?Y)) (?Y (Inf-utrans (snd ‘ ?M)))
  by(auto intro:wlp-loop-mono uInf unitary-gfp)
 finally have ge-gfp: le-utrans (gfp-trans ?Y) (?Y (Inf-utrans (snd ‘ ?M))) .
 from PLimit fX uY fSup uInf have P (?X (Sup-trans (fst ‘ ?M))) (?Y (Inf-utrans (snd ‘
?M)))
  by(iprover intro:IH)
 moreover from fSup have feasible (?X (Sup-trans (fst ‘ ?M))) by(rule fX)
 moreover have ⋀P. unitary P ⟹ unitary (?Y (Inf-utrans (snd ‘ ?M)) P)
  by(auto intro:uY uInf)
 moreover note le-lfp ge-gfp
 ultimately have pair-in: (?X (Sup-trans (fst ‘ ?M)), ?Y (Inf-utrans (snd ‘ ?M))) ∈ ?M
  by(simp)

 have ?X (Sup-trans (fst ‘ ?M)) ∈ fst ‘ ?M
  by(rule imageI[OF pair-in, of fst, simplified])
 hence le-trans (?X (Sup-trans (fst ‘ ?M))) (Sup-trans (fst ‘ ?M))
 proof(rule le-transI[OF Sup-trans-upper2[where t=?X (Sup-trans (fst ‘ ?M))
                              and S=fst ‘ ?M]])
  fix P::'s expect
  assume sP: sound P
  thus nneg P by(auto)
  from sP show bounded-by (bound-of P) P by(auto)
  from sP show ∀ u∈fst ‘ ?M. ∀ Q. nneg Q ∧ bounded-by (bound-of P) Q ⟶
                    nneg (u Q) ∧ bounded-by (bound-of P) (u Q)
    by(auto)
 qed
 hence le-trans (lfp-trans ?X) (Sup-trans (fst ‘ ?M))
  by(auto intro:lfp-trans-lowerbound feasible-sound[OF fSup])
 with fw-lfp have eqt: equiv-trans (Sup-trans (fst ‘ ?M)) (lfp-trans ?X)
  by(rule le-trans-antisym)

 have ?Y (Inf-utrans (snd ‘ ?M)) ∈ snd ‘ ?M
  by(rule imageI[OF pair-in, of snd, simplified])
 hence le-utrans (Inf-utrans (snd ‘ ?M)) (?Y (Inf-utrans (snd ‘ ?M)))
  by(intro Inf-utrans-lower, auto)
```

```
 hence le-utrans (Inf-utrans (snd ` ?M)) (gfp-trans ?Y)
  by(blast intro:gfp-trans-upperbound uInf)
 with fw-gfp have equ: equiv-utrans (Inf-utrans (snd ` ?M)) (gfp-trans ?Y)
  by(auto intro:le-utrans-antisym)
 from PLimit eqt equ show P (lfp-trans ?X) (gfp-trans ?Y) by(rule P-equiv)
qed
```

4.4.1 The Limit of Iterates

The iterates of a loop are its sequence of finite unrollings. We show shortly that this converges on the least fixed point. This is enormously useful, as we can appeal to various properties of the finite iterates (which will follow by finite induction), which we can then transfer to the limit.

```
definition iterates :: 's prog ⇒ ('s ⇒ bool) ⇒ nat ⇒ 's trans
where iterates body G i = ((λx. wp (body ;; Embed x « G »⊕ Skip)) ^^ i) (λP s. 0)

lemma iterates-0[simp]:
 iterates body G 0 = (λP s. 0)
 by(simp add:iterates-def)

lemma iterates-Suc[simp]:
 iterates body G (Suc i) = wp (body ;; Embed (iterates body G i) «G»⊕ Skip)
 by(simp add:iterates-def)
```

All iterates are healthy.

```
lemma iterates-healthy:
 healthy (wp body) ⟹ healthy (iterates body G i)
 by(induct i, auto intro:healthy-intros)
```

The iterates are an ascending chain.

```
lemma iterates-increasing:
 fixes body::'s prog
 assumes hb: healthy (wp body)
 shows le-trans (iterates body G i) (iterates body G (Suc i))
proof(induct i)
 show le-trans (iterates body G 0) (iterates body G (Suc 0))
 proof(simp add:iterates-def, rule le-transI)
  fix P::'s expect
  assume sound P
  with hb have sound (wp (body ;; Embed (λP s. 0) « G »⊕ Skip) P)
   by(auto intro!:wp-loop-step-sound)
  thus λs. 0 ⊩ wp (body ;; Embed (λP s. 0) « G »⊕ Skip) P
   by(auto)
 qed

 fix i
 assume IH: le-trans (iterates body G i) (iterates body G (Suc i))
 have equiv-trans (iterates body G (Suc i))
```

(*wp* (*body* ;; *Embed* (*iterates body G i*) « *G* »⊕ *Skip*))
by(*simp*)
also from *iterates-healthy*[*OF hb*]
have *le-trans* ... (*wp* (*body* ;; *Embed* (*iterates body G* (*Suc i*)) « *G* »⊕ *Skip*))
by(*blast intro*:*wp-loop-step-mono*[*OF hb IH*])
also have *equiv-trans* ... (*iterates body G* (*Suc* (*Suc i*)))
by(*simp*)
finally show *le-trans* (*iterates body G* (*Suc i*)) (*iterates body G* (*Suc* (*Suc i*))) **.**
qed

lemma *wp-loop-step-bounded*:
fixes *t*::*'s trans* **and** *Q*::*'s expect*
assumes *nQ*: *nneg Q*
and *bQ*: *bounded-by b Q*
and *ht*: *healthy t*
and *hb*: *healthy* (*wp body*)
shows *bounded-by b* (*wp* (*body* ;; *Embed t* « *G* »⊕ *Skip*) *Q*)
proof(*rule bounded-byI*, *simp add*:*wp-eval*)
fix *s*::*'s*
from *nQ bQ* **have** *sQ*: *sound Q* **by**(*auto*)
with *bQ ht* **have** *sound* (*t Q*) *bounded-by b* (*t Q*) **by**(*auto*)
with *hb* **have** *bounded-by b* (*wp body* (*t Q*)) **by**(*auto*)
with *bQ* **have** *wp body* (*t Q*) *s* ≤ *b Q s* ≤ *b* **by**(*auto*)
hence «*G*» *s* ∗ *wp body* (*t Q*) *s* + (*1*−«*G*» *s*) ∗ *Q s* ≤
«*G*» *s* ∗ *b* + (*1*−«*G*» *s*) ∗ *b*
by(*auto intro*:*add-mono mult-left-mono*)
also have ... = *b* **by**(*simp add*:*algebra-simps*)
finally show «*G*» *s* ∗ *wp body* (*t Q*) *s* + (*1*−«*G*» *s*) ∗ *Q s* ≤ *b* **.**
qed

This is the key result: The loop is equivalent to the supremum of its iterates. This proof follows the pattern of lemma continuous_lfp in HOL/Library/Continuity.

lemma *lfp-iterates*:
fixes *body*::*'s prog*
assumes *hb*: *healthy* (*wp body*)
and *cb*: *bd-cts* (*wp body*)
shows *equiv-trans* (*wp* (*do G* ⟶ *body od*)) (*Sup-trans* (*range* (*iterates body G*)))
(**is** *equiv-trans ?X ?Y*)
proof(*rule le-trans-antisym*)
let *?F* = λ*x*. *wp* (*body* ;; *Embed x* « *G* »⊕ *Skip*)
let *?bot* = λ(*P*::*'s* ⇒ *real*) *s*::*'s*. *0*::*real*

have *HF*: ⋀*i*. *healthy* ((*?F* ^^ *i*) *?bot*)
proof −
fix *i* **from** *hb* **show** (*?thesis i*)
by(*induct i*, *simp-all add*:*healthy-intros*)
qed

from *iterates-healthy*[*OF hb*]

```
have ⋀i. feasible (iterates body G i) by(auto)
hence fSup: feasible (Sup-trans (range (iterates body G)))
  by(auto intro:feasible-Sup-trans)

{
  fix i
  have le-trans ((?F ^^ i) ?bot) ?X
  proof(induct i)
    show le-trans ((?F ^^ 0) ?bot) ?X
    proof(simp, intro le-transI)
      fix P::'s expect
      assume sound P
      with hb healthy-wp-loop
      have sound (wp (μ x. body ;; x « G »⊕ Skip) P)
        by(auto)
      thus λs. 0 ⊩ wp (μ x. body ;; x « G »⊕ Skip) P
        by(auto)
    qed
    fix i
    assume IH: le-trans ((?F ^^ i) ?bot) ?X
    have equiv-trans ((?F ^^ (Suc i)) ?bot) (?F ((?F ^^ i) ?bot)) by(simp)
    also have le-trans ... (?F ?X)
    proof(rule wp-loop-step-mono[OF hb IH])
      fix P::'s expect
      assume sP: sound P
      with hb healthy-wp-loop
      show sound (wp (μ x. body ;; x « G »⊕ Skip) P)
        by(auto)
      from sP show sound ((?F ^^ i) ?bot P)
        by(rule healthy-sound[OF HF])
    qed
    also {
      from hb have X: le-trans (wp (body ;; Embed (λP s. bound-of P) « G »⊕ Skip))
                          (λP s. bound-of P)
       by(intro le-transI, simp add:wp-eval, auto intro: lfp-loop-fp[unfolded negate-embed])
      have equiv-trans (?F ?X) ?X
      apply (simp only: wp-eval)
      by(intro iffD1[OF equiv-trans-comm, OF lfp-trans-unfold]
           wp-loop-step-mono[OF hb] wp-loop-step-sound[OF hb], (blast|rule X)+)
    }
    finally show le-trans ((?F ^^ (Suc i)) ?bot) ?X .
  qed
}
hence ⋀i. le-trans (iterates body G i) (wp do G ⟶ body od)
  by(simp add:iterates-def)
thus le-trans ?Y ?X
  by(auto intro!:le-transI[OF Sup-trans-least2] sound-nneg
           healthy-sound[OF iterates-healthy, OF hb]
           healthy-bounded-byD[OF iterates-healthy, OF hb]
```

```
          healthy-sound[OF healthy-wp-loop] hb)

  show le-trans ?X ?Y
  proof(simp only: wp-eval, rule lfp-trans-lowerbound)
    from hb cb have bd-cts-tr ?F by(rule cts-wp-loopstep)
    with iterates-increasing[OF hb] iterates-healthy[OF hb]
    have equiv-trans (?F ?Y) (Sup-trans (range (?F o (iterates body G))))
      by (auto intro!: healthy-feasibleD bd-cts-trD cong del: image-cong-simp)
    also have le-trans (Sup-trans (range (?F o (iterates body G)))) ?Y
    proof(rule le-transI)
      fix P::'s expect
      assume sP: sound P
      show (Sup-trans (range (?F o (iterates body G)))) P ⊩ ?Y P
      proof(rule Sup-trans-least2, clarsimp)
        show ∀ u∈range ((λx. wp (body ;; Embed x « G »⊕ Skip)) ∘ iterates body G).
             ∀ R. nneg R ∧ bounded-by (bound-of P) R ⟶
               nneg (u R) ∧ bounded-by (bound-of P) (u R)
        proof(clarsimp, intro conjI)
          fix Q::'s expect and i
          assume nQ: nneg Q and bQ: bounded-by (bound-of P) Q
          hence sound Q by(auto)
          moreover from iterates-healthy[OF hb]
          have ⋀P. sound P ⟹ sound (iterates body G i P) by(auto)
          moreover note hb
          ultimately have sound (wp (body ;; Embed (iterates body G i) « G »⊕ Skip) Q)
            by(iprover intro:wp-loop-step-sound)
          thus nneg (wp (body ;; Embed (iterates body G i) « G »⊕ Skip) Q)
            by(auto)
          from nQ bQ iterates-healthy[OF hb] hb
         show bounded-by (bound-of P) (wp (body ;; Embed (iterates body G i) « G »⊕ Skip)
Q)
            by(rule wp-loop-step-bounded)
        qed
        from sP show nneg P bounded-by (bound-of P) P by(auto)
      next
        fix Q::'s expect
        assume nQ: nneg Q and bQ: bounded-by (bound-of P) Q
        hence sound Q by(auto)
        with fSup have sound (Sup-trans (range (iterates body G)) Q) by(auto)
        thus nneg (Sup-trans (range (iterates body G)) Q) by(auto)

        fix i
        show wp (body ;; Embed (iterates body G i) « G »⊕ Skip) Q ⊩
             Sup-trans (range (iterates body G)) Q
        proof(rule Sup-trans-upper2[OF - - nQ bQ])
          from iterates-healthy[OF hb]
          show ∀ u∈range (iterates body G).
               ∀ R. nneg R ∧ bounded-by (bound-of P) R ⟶
                 nneg (u R) ∧ bounded-by (bound-of P) (u R)
```

```
      by(auto)
    have wp (body ;; Embed (iterates body G i) « G »⊕ Skip) = iterates body G (Suc i)
      by(simp)
    also have ... ∈ range (iterates body G)
      by(blast)
    finally show wp (body ;; Embed (iterates body G i) « G »⊕ Skip) ∈
             range (iterates body G) .
   qed
  qed
 qed
 finally show le-trans (?F ?Y) ?Y .

 fix P::'s expect
 assume sound P
 with fSup show sound (?Y P) by(auto)
qed
qed
```

Therefore, evaluated at a given point (state), the sequence of iterates gives a sequence of real values that converges on that of the loop itself.

```
corollary loop-iterates:
 fixes body::'s prog
 assumes hb: healthy (wp body)
    and cb: bd-cts (wp body)
    and sP: sound P
 shows (λi. iterates body G i P s) ———→ wp (do G —→ body od) P s
proof −
 let ?X = {f s |f. f ∈ {t P |t. t ∈ range (iterates body G)}}
 have closure-Sup: Sup ?X ∈ closure ?X
 proof(rule closure-contains-Sup, simp, clarsimp)
  fix i
  from sP have bounded-by (bound-of P) P by(auto)
   with iterates-healthy[OF hb] sP have ⋀j. bounded-by (bound-of P) (iterates body G j
P)
    by(auto)
  thus iterates body G i P s ≤ bound-of P by(auto)
 qed

 have (λi. iterates body G i P s) ———→ Sup {f s |f. f ∈ {t P |t. t ∈ range (iterates body
G)}}
 proof(rule LIMSEQ-I)
  fix r::real assume posr: 0 < r
  with closure-Sup obtain y where yin: y ∈ ?X and ey: dist y (Sup ?X) < r
   by(simp only:closure-approachable, blast)
  from yin obtain i where yit: y = iterates body G i P s by(auto)
  {
   fix j
   have i ≤ j —→ le-trans (iterates body G i) (iterates body G j)
   proof(induct j, simp, clarify)
```

fix *k*
assume *IH*: $i \leq k \longrightarrow$ *le-trans* (*iterates body G i*) (*iterates body G k*)
and *le*: $i \leq$ *Suc k*
show *le-trans* (*iterates body G i*) (*iterates body G* (*Suc k*))
proof(*cases i* = *Suc k*, *simp*)
assume $i \neq$ *Suc k*
with *le* **have** $i \leq k$ **by**(*auto*)
with *IH* **have** *le-trans* (*iterates body G i*) (*iterates body G k*) **by**(*auto*)
also note *iterates-increasing*[*OF hb*]
finally show *le-trans* (*iterates body G i*) (*iterates body G* (*Suc k*)) **.**
qed
qed
}
with *sP* **have** $\forall j \geq i$. *iterates body G i P s* $\leq$ *iterates body G j P s*
by(*auto*)
moreover {
from *sP* **have** *bounded-by* (*bound-of P*) *P* **by**(*auto*)
with *iterates-healthy*[*OF hb*] *sP* **have** $\bigwedge j$. *bounded-by* (*bound-of P*) (*iterates body G j*
P)
by(*auto*)
hence $\bigwedge j$. *iterates body G j P s* $\leq$ *bound-of P* **by**(*auto*)
hence $\bigwedge j$. *iterates body G j P s* $\leq$ *Sup ?X*
by(*intro cSup-upper bdd-aboveI*, *auto*)
}
ultimately have $\bigwedge j$. $i \leq j \Longrightarrow$
norm (*iterates body G j P s* $-$ *Sup ?X*) $\leq$
norm (*iterates body G i P s* $-$ *Sup ?X*)
by(*auto*)
also from *ey yit* **have** *norm* (*iterates body G i P s* $-$ *Sup ?X*) $< r$
by(*simp add*:*dist-real-def*)
finally show $\exists$ *no*. $\forall n \geq no$. *norm* (*iterates body G n P s* $-$
Sup {*f s* |*f*. *f* $\in$ {*t P* |*t*. *t* $\in$ *range* (*iterates body G*)}}) $< r$
by(*auto*)
qed
moreover
from *hb cb sP* **have** *wp do G* $\longrightarrow$ *body od P s* = *Sup-trans* (*range* (*iterates body G*)) *P s*
by(*simp add*:*equiv-transD*[*OF lfp-iterates*])
moreover have ... = *Sup* {*f s* |*f*. *f* $\in$ {*t P* |*t*. *t* $\in$ *range* (*iterates body G*)}}
by(*simp add*:*Sup-trans-def Sup-exp-def*)
ultimately show *?thesis* **by**(*simp*)
qed

The iterates themselves are all continuous.

lemma *cts-iterates*:
fixes *body*::*'s prog*
assumes *hb*: *healthy* (*wp body*)
and *cb*: *bd-cts* (*wp body*)
shows *bd-cts* (*iterates body G i*)
proof(*induct i*, *simp-all*)

```
  have range (λ(n::nat) (s::'s). 0::real) = {λs. 0::real}
    by(auto)
  thus bd-cts (λP (s::'s). 0)
    by(intro bd-ctsI, simp add:o-def Sup-exp-def)
next
  fix i
  assume IH: bd-cts (iterates body G i)
  thus bd-cts (wp (body ;; Embed (iterates body G i) « G »⊕ Skip))
    by(blast intro:cts-wp-PC cts-wp-Seq cts-wp-Embed cts-wp-Skip
               healthy-intros iterates-healthy cb hb)
qed
```

Therefore so is the loop itself.

```
lemma cts-wp-loop:
  fixes body::'s prog
  assumes hb: healthy (wp body)
     and cb: bd-cts (wp body)
  shows bd-cts (wp do G ⟶ body od)
proof(rule bd-ctsI)
  fix M::nat ⇒ 's expect and b::real
  assume chain: ⋀i. M i ⊩ M (Suc i)
    and sM: ⋀i. sound (M i)
    and bM: ⋀i. bounded-by b (M i)

  from sM bM iterates-healthy[OF hb]
  have ⋀j i. bounded-by b (iterates body G i (M j)) by(blast)
  hence iB: ⋀j i s. iterates body G i (M j) s ≤ b by(auto)

  from sM bM have sSup: sound (Sup-exp (range M))
    by(auto intro:Sup-exp-sound)
  with lfp-iterates[OF hb cb]
  have wp do G ⟶ body od (Sup-exp (range M)) =
      Sup-trans (range (iterates body G)) (Sup-exp (range M))
    by(simp add:equiv-transD)
  also {
    from chain sM bM
    have ⋀i. iterates body G i (Sup-exp (range M)) = Sup-exp (range (iterates body G i o
M))
      by(blast intro:bd-ctsD cts-iterates[OF hb cb])
    hence {t (Sup-exp (range M)) |t. t ∈ range (iterates body G)} =
          {Sup-exp (range (t o M)) |t. t ∈ range (iterates body G)}
      by(auto intro:sym)
    hence Sup-trans (range (iterates body G)) (Sup-exp (range M)) =
          Sup-exp {Sup-exp (range (t ∘ M)) |t. t ∈ range (iterates body G)}
      by(simp add:Sup-trans-def)
  }
  also {
    have ⋀s. {f s |f. ∃t. f = (λs. Sup {f s |f. f ∈ range (t ∘ M)}) ∧
                    t ∈ range (iterates body G)} =
```

```
      range (λi. Sup (range (λj. iterates body G i (M j) s)))
    (is ⋀s. ?X s = ?Y s)
  proof(intro antisym subsetI)
    fix s x
    assume x ∈ ?X s
    then obtain t where rwx: x = Sup {f s |f. f ∈ range (t ∘ M)}
                and t ∈ range (iterates body G) by(auto)
    then obtain i where t = iterates body G i by(auto)
    with rwx have x = Sup {f s |f. f ∈ range (λj. iterates body G i (M j))}
      by(simp add:o-def)
    moreover have {f s |f. f ∈ range (λj. iterates body G i (M j))} =
               range (λj. iterates body G i (M j) s) by(auto)
    ultimately have x = Sup (range (λj. iterates body G i (M j) s))
      by(simp)
    thus x ∈ range (λi. Sup (range (λj. iterates body G i (M j) s)))
      by(auto)
  next
    fix s x
    assume x ∈ ?Y s
    then obtain i where A: x = Sup (range (λj. iterates body G i (M j) s))
      by(auto)

    have ⋀s. {f s |f. f ∈ range (λj. iterates body G i (M j))} =
        range (λj. iterates body G i (M j) s) by(auto)
    hence B: (λs. Sup (range (λj. iterates body G i (M j) s))) =
          (λs. Sup {f s |f. f ∈ range (iterates body G i o M)})
      by(simp add:o-def)

    have C: iterates body G i ∈ range (iterates body G) by(auto)

    have ∃f. x = f s ∧
           (∃t. f = (λs. Sup {f s |f. f ∈ range (t ∘ M)}) ∧
              t ∈ range (iterates body G))
      by(iprover intro:A B C)
    thus x ∈ ?X s by(simp)
  qed
  hence Sup-exp {Sup-exp (range (t ∘ M)) |t. t ∈ range (iterates body G)} =
        (λs. Sup (range (λi. Sup (range (λj. iterates body G i (M j) s)))))
    by(simp add:Sup-exp-def)
}
also have (λs. Sup (range (λi. Sup (range (λj. iterates body G i (M j) s))))) =
        (λs. Sup (range (λ(i,j). iterates body G i (M j) s)))
  (is ?X = ?Y)
proof(rule ext, rule antisym)
  fix s::'s
  show ?Y s ≤ ?X s
  proof(rule cSup-least, blast, clarify)
    fix i j::nat
    from iB have iterates body G i (M j) s ≤ Sup (range (λj. iterates body G i (M j) s))
```

```
      by(intro cSup-upper bdd-aboveI, auto)
    also from iB have ... ≤ Sup (range (λi. Sup (range (λj. iterates body G i (M j) s))))
      by(intro cSup-upper cSup-least bdd-aboveI, (blast intro:cSup-least)+)
    finally show iterates body G i (M j) s ≤
              Sup (range (λi. Sup (range (λj. iterates body G i (M j) s)))) .
  qed
  have ⋀i j. iterates body G i (M j) s ≤
          Sup (range (λ(i, j). iterates body G i (M j) s))
    by(rule cSup-upper, auto intro:iB)
  thus ?X s ≤ ?Y s
    by(intro cSup-least, blast, clarify, simp, blast intro:cSup-least)
qed
also have ... = (λs. Sup (range (λj .Sup (range (λi. iterates body G i (M j) s)))))
  (is ?X = ?Y)
proof(rule ext, rule antisym)
  fix s::'s
  have ⋀i j. iterates body G i (M j) s ≤
          Sup (range (λ(i, j). iterates body G i (M j) s))
    by(rule cSup-upper, auto intro:iB)
  thus ?Y s ≤ ?X s
    by(intro cSup-least, blast, clarify, simp, blast intro:cSup-least)
  show ?X s ≤ ?Y s
  proof(rule cSup-least, blast, clarify)
    fix i j::nat
    from iB have iterates body G i (M j) s ≤ Sup (range (λi. iterates body G i (M j) s))
      by(intro cSup-upper bdd-aboveI, auto)
    also from iB have ... ≤ Sup (range (λj. Sup (range (λi. iterates body G i (M j) s))))
      by(intro cSup-upper cSup-least bdd-aboveI, blast, blast intro:cSup-least)
    finally show iterates body G i (M j) s ≤
              Sup (range (λj. Sup (range (λi. iterates body G i (M j) s)))) .
  qed
qed
also {
  have ⋀s. range (λj. Sup (range (λi. iterates body G i (M j) s))) =
          {f s |f. f ∈ range ((λP s. Sup {f s |f. ∃t. f = t P ∧
          t ∈ range (iterates body G)}) ∘ M)} (is ⋀s. ?X s = ?Y s)
  proof(intro antisym subsetI)
    fix s x
    assume x ∈ ?X s
    then obtain j where rwx: x = Sup (range (λi. iterates body G i (M j) s)) by(auto)
    moreover {
      have ⋀s. range (λi. iterates body G i (M j) s) =
              {f s |f. ∃t. f = t (M j) ∧ t ∈ range (iterates body G)}
        by(auto)
      hence (λs. Sup (range (λi. iterates body G i (M j) s))) ∈
          range ((λP s. Sup {f s |f.
                  ∃t. f = t P ∧ t ∈ range (iterates body G)}) ∘ M)
        by (simp add: o-def cong del: SUP-cong-simp)
    }
```

```
    ultimately show x ∈ ?Y s by(auto)
  next
    fix s x
    assume x ∈ ?Y s
    then obtain P where rwx: x = P s
                and Pin: P ∈ range ((λP s. Sup {f s |f.
                      ∃ t. f = t P ∧ t ∈ range (iterates body G)}) ∘ M)
      by(auto)
    then obtain j where P = (λs. Sup {f s |f. ∃ t. f = t (M j) ∧
                                 t ∈ range (iterates body G)})
      by(auto)
    also {
      have ⋀s. {f s |f. ∃ t. f = t (M j) ∧ t ∈ range (iterates body G)} =
           range (λi. iterates body G i (M j) s) by(auto)
      hence (λs. Sup {f s |f. ∃ t. f = t (M j) ∧ t ∈ range (iterates body G)}) =
           (λs. Sup (range (λi. iterates body G i (M j) s)))
        by(simp)
    }
    finally have x = Sup (range (λi. iterates body G i (M j) s))
      by(simp add:rwx)
    thus x ∈ ?X s by(simp)
  qed
  hence (λs. Sup (range (λj .Sup (range (λi. iterates body G i (M j) s))))) =
       Sup-exp (range (Sup-trans (range (iterates body G)) o M))
    by (simp add: Sup-exp-def Sup-trans-def cong del: SUP-cong-simp)
}
also have Sup-exp (range (Sup-trans (range (iterates body G)) o M)) =
        Sup-exp (range (wp do G ⟶ body od o M))
  by(simp add:o-def equiv-transD[OF lfp-iterates, OF hb cb, OF sM])
finally show wp do G ⟶ body od (Sup-exp (range M)) =
          Sup-exp (range (wp do G ⟶ body od o M)) .
qed

lemmas cts-intros =
 cts-wp-Abort  cts-wp-Skip
 cts-wp-Seq    cts-wp-PC
 cts-wp-DC     cts-wp-Embed
 cts-wp-Apply  cts-wp-SetDC
 cts-wp-SetPC  cts-wp-Bind
 cts-wp-repeat

end
```

4.5 Sublinearity

theory *Sublinearity* **imports** *Embedding Healthiness LoopInduction* **begin**

4.5.1 Nonrecursive Primitives

Sublinearity of non-recursive programs is generally straightforward, and follows from the alebraic properties of the underlying operations, together with healthiness.

```
lemma sublinear-wp-Skip:
  sublinear (wp Skip)
  by(auto simp:wp-eval)

lemma sublinear-wp-Abort:
  sublinear (wp Abort)
  by(auto simp:wp-eval)

lemma sublinear-wp-Apply:
  sublinear (wp (Apply f))
  by(auto simp:wp-eval)

lemma sublinear-wp-Seq:
  fixes x::'s prog
  assumes slx: sublinear (wp x) and sly: sublinear (wp y)
     and hx: healthy (wp x)  and hy: healthy (wp y)
  shows sublinear (wp (x ;; y))
proof(rule sublinearI, simp add:wp-eval)
  fix P::'s ⇒ real and Q::'s ⇒ real and s::'s
  and a::real and b::real and c::real
  assume sP: sound P and sQ: sound Q
    and nna: 0 ≤ a and nnb: 0 ≤ b and nnc: 0 ≤ c

  with slx hy have a ∗ wp x (wp y P) s + b ∗ wp x (wp y Q) s ⊖ c ≤
              wp x (λs. a ∗ wp y P s + b ∗ wp y Q s ⊖ c) s
    by(blast intro:sublinearD)
  also {
    from sP sQ nna nnb nnc sly
    have ⋀s. a ∗ wp y P s + b ∗ wp y Q s ⊖ c ≤
           wp y (λs. a ∗ P s + b ∗ Q s ⊖ c) s
      by(blast intro:sublinearD)
    moreover from sP sQ hy
    have sound (wp y P) and sound (wp y Q) by(auto)
    moreover with nna nnb nnc
    have sound (λs. a ∗ wp y P s + b ∗ wp y Q s ⊖ c)
      by(auto intro!:sound-intros tminus-sound)
    moreover from sP sQ nna nnb nnc
    have sound (λs. a ∗ P s + b ∗ Q s ⊖ c)
      by(auto intro!:sound-intros tminus-sound)
    moreover with hy have sound (wp y (λs. a ∗ P s + b ∗ Q s ⊖ c))
      by(blast)
    ultimately
    have wp x (λs. a ∗ wp y P s + b ∗ wp y Q s ⊖ c) s ≤
        wp x (wp y (λs. a ∗ P s + b ∗ Q s ⊖ c)) s
      by(blast intro!:le-funD[OF mono-transD[OF healthy-monoD[OF hx]]])
```

```
 }
 finally show a * wp x (wp y P) s + b * wp x (wp y Q) s ⊖ c ≤
           wp x (wp y (λs. a * P s + b * Q s ⊖ c)) s .
qed

lemma sublinear-wp-PC:
 fixes x::'s prog
 assumes slx: sublinear (wp x) and sly: sublinear (wp y)
   and uP: unitary P
 shows sublinear (wp (x P⊕ y))
proof(rule sublinearI, simp add:wp-eval)
 fix R::'s ⇒ real and Q::'s ⇒ real and s::'s
 and a::real and b::real and c::real
 assume sR: sound R and sQ: sound Q
   and nna: 0 ≤ a and nnb: 0 ≤ b and nnc: 0 ≤ c

 have a * (P s * wp x Q s + (1 − P s) * wp y Q s) +
     b * (P s * wp x R s + (1 − P s) * wp y R s) ⊖ c =
     (P s * a * wp x Q s + (1 − P s) * a * wp y Q s) +
     (P s * b * wp x R s + (1 − P s) * b * wp y R s) ⊖ c
  by(simp add:field-simps)
 also
 have ... = (P s * a * wp x Q s + P s * b * wp x R s) +
         ((1 − P s) * a * wp y Q s + (1 − P s) * b * wp y R s) ⊖ c
  by(simp add:ac-simps)
 also
 have ... = P s * (a * wp x Q s + b * wp x R s) +
         (1 − P s) * (a * wp y Q s + b * wp y R s) ⊖
         (P s * c + (1 − P s) * c)
  by(simp add:field-simps)
 also
 have ... ≤ (P s * (a * wp x Q s + b * wp x R s) ⊖ P s * c) +
         ((1 − P s) * (a * wp y Q s + b * wp y R s) ⊖ (1 − P s) * c)
  by(rule tminus-add-mono)
 also {
   from uP have 0 ≤ P s and 0 ≤ 1 − P s
    by auto
   hence (P s * (a * wp x Q s + b * wp x R s) ⊖ P s * c) +
        ((1 − P s) * (a * wp y Q s + b * wp y R s) ⊖ (1 − P s) * c) =
        P s * (a * wp x Q s + b * wp x R s ⊖ c) +
        (1 − P s) * (a * wp y Q s + b * wp y R s ⊖ c)
    by(simp add:tminus-left-distrib)
 }
 also {
  from sQ sR nna nnb nnc slx
  have a * wp x Q s + b * wp x R s ⊖ c ≤
       wp x (λs. a * Q s + b * R s ⊖ c) s
   by(blast)
  moreover
```

```
    from sQ sR nna nnb nnc sly
    have a ∗ wp y Q s + b ∗ wp y R s ⊖ c ≤
         wp y (λs. a ∗ Q s + b ∗ R s ⊖ c) s
      by(blast)
    moreover
    from uP have 0 ≤ P s and 0 ≤ 1 − P s
      by auto
    ultimately
    have P s ∗ (a ∗ wp x Q s + b ∗ wp x R s ⊖ c) +
         (1 − P s) ∗ (a ∗ wp y Q s + b ∗ wp y R s  ⊖ c) ≤
         P s ∗ wp x (λs. a ∗ Q s + b ∗ R s ⊖ c) s +
         (1 − P s) ∗ wp y (λs. a ∗ Q s + b ∗ R s ⊖ c) s
      by(blast intro:add-mono mult-left-mono)
  }
  finally
  show  a ∗ (P s ∗ wp x Q s + (1 − P s) ∗ wp y Q s) +
        b ∗ (P s ∗ wp x R s + (1 − P s) ∗ wp y R s) ⊖ c ≤
        P s ∗ wp x (λs. a ∗ Q s + b ∗ R s ⊖ c) s +
        (1 − P s) ∗ wp y (λs. a ∗ Q s + b ∗ R s ⊖ c) s .
qed

lemma sublinear-wp-DC:
  fixes x::'s prog
  assumes slx: sublinear (wp x) and sly: sublinear (wp y)
  shows sublinear (wp (x ⨅ y))
proof(rule sublinearI, simp only:wp-eval)
  fix R::'s ⇒ real and Q::'s ⇒ real and s::'s
  and a::real and b::real and c::real
  assume sR: sound R and sQ: sound Q
    and nna: 0 ≤ a and nnb: 0 ≤ b and nnc: 0 ≤ c

  from nna nnb
  have a ∗ min (wp x Q s) (wp y Q s) +
       b ∗ min (wp x R s) (wp y R s) ⊖ c =
       min (a ∗ wp x Q s) (a ∗ wp y Q s) +
       min (b ∗ wp x R s) (b ∗ wp y R s) ⊖ c
    by(simp add:min-distrib)
  also
  have ... ≤ min (a ∗ wp x Q s + b ∗ wp x R s)
                 (a ∗ wp y Q s + b ∗ wp y R s) ⊖ c
    by(auto intro!:tminus-left-mono)
  also
  have ... = min (a ∗ wp x Q s + b ∗ wp x R s ⊖ c)
                 (a ∗ wp y Q s + b ∗ wp y R s ⊖ c)
    by(rule min-tminus-distrib)
  also {
    from slx sQ sR nna nnb nnc
    have a ∗ wp x Q s + b ∗ wp x R s ⊖ c ≤
         wp x (λs. a ∗ Q s + b ∗ R s ⊖ c) s
```

```
    by(blast)
  moreover
  from sly sQ sR nna nnb nnc
  have a ∗ wp y Q s + b ∗ wp y R s ⊖ c ≤
      wp y (λs. a ∗ Q s + b ∗ R s ⊖ c) s
    by(blast)
  ultimately
  have min (a ∗ wp x Q s + b ∗ wp x R s ⊖ c)
         (a ∗ wp y Q s + b ∗ wp y R s ⊖ c) ≤
      min (wp x (λs. a ∗ Q s + b ∗ R s ⊖ c) s)
         (wp y (λs. a ∗ Q s + b ∗ R s ⊖ c) s)
    by(auto)
 }
 finally show a ∗ min (wp x Q s) (wp y Q s) +
          b ∗ min (wp x R s) (wp y R s) ⊖ c ≤
          min (wp x (λs. a ∗ Q s + b ∗ R s ⊖ c) s)
             (wp y (λs. a ∗ Q s + b ∗ R s ⊖ c) s) .
qed
```

As for continuity, we insist on a finite support.

```
lemma sublinear-wp-SetPC:
 fixes p::'a ⇒ 's prog
 assumes slp: ⋀s a. a ∈ supp (P s) ⟹ sublinear (wp (p a))
    and sum: ⋀s. (∑ a∈supp (P s). P s a) ≤ 1
    and nnP: ⋀s a. 0 ≤ P s a
    and fin: ⋀s. finite (supp (P s))
 shows sublinear (wp (SetPC p P))
proof(rule sublinearI, simp add:wp-eval)
 fix R::'s ⇒ real and Q::'s ⇒ real and s::'s
 and a::real and b::real and c::real
 assume sR: sound R and sQ: sound Q
   and nna: 0 ≤ a and nnb: 0 ≤ b and nnc: 0 ≤ c
 have a ∗ (∑ a'∈supp (P s). P s a' ∗ wp (p a') Q s) +
     b ∗ (∑ a'∈supp (P s). P s a' ∗ wp (p a') R s) ⊖ c =
     (∑ a'∈supp (P s). P s a' ∗ (a ∗ wp (p a') Q s + b ∗ wp (p a') R s)) ⊖ c
  by(simp add:field-simps sum-distrib-left sum.distrib)
 also have ... ≤
         (∑ a'∈supp (P s). P s a' ∗ (a ∗ wp (p a') Q s + b ∗ wp (p a') R s)) ⊖
         (∑ a'∈supp (P s). P s a' ∗ c)
 proof(rule tminus-right-antimono)
  have (∑ a'∈supp (P s). P s a' ∗ c) ≤ (∑ a'∈supp (P s). P s a') ∗ c
    by(simp add:sum-distrib-right)
  also from sum and nnc have ... ≤ 1 ∗ c
    by(rule mult-right-mono)
  finally show (∑ a'∈supp (P s). P s a' ∗ c) ≤ c by(simp)
 qed
 also from fin
 have ... ≤ (∑ a'∈supp (P s). P s a' ∗ (a ∗ wp (p a') Q s + b ∗ wp (p a') R s) ⊖ P s a' ∗
c)
```

```
    by(blast intro:tminus-sum-mono)
  also have ... = (∑ a'∈supp (P s). P s a' ∗ (a ∗ wp (p a') Q s + b ∗ wp (p a') R s ⊖ c))
    by(simp add:nnP tminus-left-distrib)
  also {
    from slp sQ sR nna nnb nnc
    have ⋀a'. a' ∈ supp (P s) ⟹ a ∗ wp (p a') Q s + b ∗ wp (p a') R s ⊖ c ≤
                          wp (p a') (λs. a ∗ Q s + b ∗ R s ⊖ c) s
      by(blast)
    with nnP
    have (∑ a'∈supp (P s). P s a' ∗ (a ∗ wp (p a') Q s + b ∗ wp (p a') R s ⊖ c)) ≤
         (∑ a'∈supp (P s). P s a' ∗ wp (p a') (λs. a ∗ Q s + b ∗ R s ⊖ c) s)
      by(blast intro:sum-mono mult-left-mono)
  }
  finally
  show a ∗ (∑ a'∈supp (P s). P s a' ∗ wp (p a') Q s) +
       b ∗ (∑ a'∈supp (P s). P s a' ∗ wp (p a') R s) ⊖ c ≤
       (∑ a'∈supp (P s). P s a' ∗ wp (p a') (λs. a ∗ Q s + b ∗ R s ⊖ c) s) .
qed

lemma sublinear-wp-SetDC:
  fixes p::'a ⇒ 's prog
  assumes slp: ⋀s a. a ∈ S s ⟹ sublinear (wp (p a))
      and hp:  ⋀s a. a ∈ S s ⟹ healthy (wp (p a))
      and ne:  ⋀s. S s ≠ {}
  shows sublinear (wp (SetDC p S))
proof(rule sublinearI, simp add:wp-eval, rule cInf-greatest)
  fix P::'s ⇒ real and Q::'s ⇒ real and s::'s and x y
  and a::real and b::real and c::real
  assume sP: sound P and sQ: sound Q
     and nna: 0 ≤ a and nnb: 0 ≤ b and nnc: 0 ≤ c

  from ne show (λpr. wp (p pr) (λs. a ∗ P s + b ∗ Q s ⊖ c) s) ‘ S s ≠ {} by(auto)

  assume yin: y ∈ (λpr. wp (p pr) (λs. a ∗ P s + b ∗ Q s ⊖ c) s) ‘ S s
  then obtain x where xin: x ∈ S s and rwy: y = wp (p x) (λs. a ∗ P s + b ∗ Q s ⊖ c) s
    by(auto)

  from xin hp sP nna
  have a ∗ Inf ((λa. wp (p a) P s) ‘ S s) ≤ a ∗ wp (p x) P s
    by(intro mult-left-mono[OF cInf-lower] bdd-belowI[where m=0], blast+)
  moreover from xin hp sQ nnb
  have b ∗ Inf ((λa. wp (p a) Q s) ‘ S s) ≤ b ∗ wp (p x) Q s
    by(intro mult-left-mono[OF cInf-lower] bdd-belowI[where m=0], blast+)
  ultimately
  have a ∗ Inf ((λa. wp (p a) P s) ‘ S s) +
       b ∗ Inf ((λa. wp (p a) Q s) ‘ S s) ⊖ c ≤
       a ∗ wp (p x) P s + b ∗ wp (p x) Q s ⊖ c
    by(blast intro:tminus-left-mono add-mono)
```

also from *xin slp sP sQ nna nnb nnc*
have ... $\leq$ *wp* (*p x*) ($\lambda s.$ *a* $*$ *P s* + *b* $*$ *Q s* $\ominus$ *c*) *s*
by(*blast*)

finally show *a* $*$ *Inf* (($\lambda a.$ *wp* (*p a*) *P s*) ' *S s*) + *b* $*$ *Inf* (($\lambda a.$ *wp* (*p a*) *Q s*) ' *S s*) $\ominus$ *c* $\leq$ *y*
by(*simp add*:*rwy*)
qed

lemma *sublinear-wp-Embed*:
sublinear t $\Longrightarrow$ *sublinear* (*wp* (*Embed t*))
by(*simp add*:*wp-eval*)

lemma *sublinear-wp-repeat*:
[[*sublinear* (*wp p*); *healthy* (*wp p*)]] $\Longrightarrow$ *sublinear* (*wp* (*repeat n p*))
by(*induct n*, *simp-all add*:*sublinear-wp-Seq sublinear-wp-Skip healthy-wp-repeat*)

lemma *sublinear-wp-Bind*:
[[$\bigwedge s.$ *sublinear* (*wp* (*a* (*f s*)))]] $\Longrightarrow$ *sublinear* (*wp* (*Bind f a*))
by(*rule sublinearI*, *simp add*:*wp-eval*, *auto*)

4.5.2 Sublinearity for Loops

We break the proof of sublinearity loops into separate proofs of sub-distributivity and sub-additivity. The first follows by transfinite induction.

lemma *sub-distrib-wp-loop*:
fixes *body*::'*s prog*
assumes *sdb*: *sub-distrib* (*wp body*)
and *hb*: *healthy* (*wp body*)
and *nhb*: *nearly-healthy* (*wlp body*)
shows *sub-distrib* (*wp* (*do G* $\longrightarrow$ *body od*))
proof –
have $\forall$ *P s*. *sound P* $\longrightarrow$ *wp* (*do G* $\longrightarrow$ *body od*) *P s* $\ominus$ *1* $\leq$
wp (*do G* $\longrightarrow$ *body od*) ($\lambda s.$ *P s* $\ominus$ *1*) *s*
proof(*rule loop-induct*[*OF hb nhb*], *safe*)
fix *S*::('*s trans* $\times$ '*s trans*) *set* **and** *P*::'*s expect* **and** *s*::'*s*
assume *saS*: $\forall$ *x*$\in$*S*. $\forall$ *P s*. *sound P* $\longrightarrow$ *fst x P s* $\ominus$ *1* $\leq$ *fst x* ($\lambda s.$ *P s* $\ominus$ *1*) *s*
and *sP*: *sound P*
and *fS*: $\forall$ *x*$\in$*S*. *feasible* (*fst x*)

from *sP* **have** *sPm*: *sound* ($\lambda s.$ *P s* $\ominus$ *1*) **by**(*auto intro*:*tminus-sound*)

have *nnSup*: $\bigwedge s.$ *0* $\leq$ *Sup-trans* (*fst* ' *S*) ($\lambda s.$ *P s* $\ominus$ *1*) *s*
proof(*cases S*={}, *simp add*:*Sup-trans-def Sup-exp-def*)
fix *s*
assume *S* $\neq$ {}
then obtain *x* **where** *xin*: *x*$\in$*S* **by**(*auto*)
with *fS sPm* **have** *0* $\leq$ *fst x* ($\lambda s.$ *P s* $\ominus$ *1*) *s* **by**(*auto*)
also from *xin fS sPm* **have** ... $\leq$ *Sup-trans* (*fst* ' *S*) ($\lambda s.$ *P s* $\ominus$ *1*) *s*

```
    by(auto intro!: le-funD[OF Sup-trans-upper2])
  finally show ?thesis s .
 qed

 have ⋀x s. fst x P s ≤ (fst x P s ⊖ 1) + 1 by(simp add:tminus-def)
 also from saS sP
 have ⋀x s. x∈S ⟹ (fst x P s ⊖ 1) + 1 ≤ fst x (λs. P s ⊖ 1) s + 1
  by(auto intro:add-right-mono)
 also {
  from sP have sound (λs. P s ⊖ 1) by(auto intro:tminus-sound)
  with fS have ⋀x s. x∈S ⟹ fst x (λs. P s ⊖ 1) s + 1 ≤
                  Sup-trans (fst ‘ S) (λs. P s ⊖ 1) s + 1
   by(blast intro!: add-right-mono le-funD[OF Sup-trans-upper2])
 }
 finally have le: ⋀s. ∀x∈S. fst x P s ≤ Sup-trans (fst ‘ S) (λs. P s ⊖ 1) s + 1
  by(auto)
 moreover from nnSup have nn: ⋀s. 0 ≤ Sup-trans (fst ‘ S) (λs. P s ⊖ 1) s + 1
  by(auto intro:add-nonneg-nonneg)
 ultimately
 have leSup: Sup-trans (fst ‘ S) P s ≤ Sup-trans (fst ‘ S) (λs. P s ⊖ 1) s + 1
  unfolding Sup-trans-def
  by(intro le-funD[OF Sup-exp-least], auto)

 show Sup-trans (fst ‘ S) P s ⊖ 1 ≤ Sup-trans (fst ‘ S) (λs. P s ⊖ 1) s
 proof(cases Sup-trans (fst ‘ S) P s ≤ 1, simp-all add:nnSup)
  from leSup have Sup-trans (fst ‘ S) P s − 1 ≤
             Sup-trans (fst ‘ S) (λs. P s ⊖ 1) s + 1 − 1
   by(auto)
  thus Sup-trans (fst ‘ S) P s − 1 ≤ Sup-trans (fst ‘ S) (λs. P s ⊖ 1) s by(simp)
 qed
next
 fix t::'s trans and P::'s expect and s::'s
 assume IH: ∀P s. sound P ⟶ t P s ⊖ 1 ≤ t (λa. P a ⊖ 1) s
   and ft: feasible t
   and sP: sound P

  from sP have sound (λs. P s ⊖ 1) by(auto intro:tminus-sound)
  with ft have s2: sound (t (λs. P s ⊖ 1)) by(auto)
  from sP ft have sound (t P) by(auto)
  hence s3: sound (λs. t P s ⊖ 1) by(auto intro!:tminus-sound)

 show wp (body ;; Embed t « G »⊕ Skip) P s ⊖ 1 ≤
     wp (body ;; Embed t « G »⊕ Skip) (λa. P a ⊖ 1) s
 proof(simp add:wp-eval)
  have «G» s ∗ wp body (t P) s + (1 − «G» s) ∗ P s ⊖ 1 =
      «G» s ∗ wp body (t P) s + (1 − «G» s) ∗ P s ⊖ («G» s + (1 − «G» s))
   by(simp)
  also have ... ≤ («G» s ∗ wp body (t P) s ⊖ «G» s) +
            ((1 − «G» s) ∗ P s ⊖ (1 − «G» s))
```

by(*rule tminus-add-mono*)
also have ... $=$ «G» $s * (wp\ body\ (t\ P)\ s \ominus 1) + (1 -$ «G» $s) * (P\ s \ominus 1)$
by(*simp add*:*tminus-left-distrib*)
also {
from *ft sP* **have** $wp\ body\ (t\ P)\ s \ominus 1 \leq wp\ body\ (\lambda s.\ t\ P\ s \ominus 1)\ s$
by(*auto intro*:*sub-distribD*[*OF sdb*])
also {
from *IH sP* **have** $\lambda s.\ t\ P\ s \ominus 1 \Vdash t\ (\lambda s.\ P\ s \ominus 1)$ **by**(*auto*)
with *sP ft s2 s3* **have** $wp\ body\ (\lambda s.\ t\ P\ s \ominus 1)\ s \leq wp\ body\ (t\ (\lambda s.\ P\ s \ominus 1))\ s$
by(*blast intro*:*le-funD*[*OF mono-transD*, *OF healthy-monoD*, *OF hb*])
}
finally have «G» $s * (wp\ body\ (t\ P)\ s \ominus 1) + (1 -$ «G» $s) * (P\ s \ominus 1) \leq$
«G» $s * wp\ body\ (t\ (\lambda s.\ P\ s \ominus 1))\ s + (1 -$ «G» $s) * (P\ s \ominus 1)$
by(*auto intro*:*add-right-mono mult-left-mono*)
}
finally show «G» $s * wp\ body\ (t\ P)\ s + (1 -$ «G» $s) * P\ s \ominus 1 \leq$
«G» $s * wp\ body\ (t\ (\lambda s.\ P\ s \ominus 1))\ s + (1 -$ «G» $s) * (P\ s \ominus 1)$ **.**
qed
next
fix $t\ t'$::$'s$ *trans* **and** P::$'s$ *expect* **and** s::$'s$
assume *IH*: $\forall P\ s.\ sound\ P \longrightarrow t\ P\ s \ominus 1 \leq t\ (\lambda a.\ P\ a \ominus 1)\ s$
and *eq*: *equiv-trans* $t\ t'$ **and** *sP*: *sound P*

from *sP* **have** $t'\ P\ s \ominus 1 = t\ P\ s \ominus 1$ **by**(*simp add*:*equiv-transD*[*OF eq*])
also from *sP IH* **have** ... $\leq t\ (\lambda s.\ P\ s \ominus 1)\ s$ **by**(*auto*)
also {
from *sP* **have** $sound\ (\lambda s.\ P\ s \ominus 1)$ **by**(*simp add*:*tminus-sound*)
hence $t\ (\lambda s.\ P\ s \ominus 1)\ s = t'\ (\lambda s.\ P\ s \ominus 1)\ s$ **by**(*simp add*:*equiv-transD*[*OF eq*])
}
finally show $t'\ P\ s \ominus 1 \leq t'\ (\lambda s.\ P\ s \ominus 1)\ s$ **.**
qed
thus *?thesis* **by**(*auto intro*!:*sub-distribI*)
qed

For sub-additivity, we again use the limit-of-iterates characterisation. Firstly, all iterates are sublinear:

lemma *sublinear-iterates*:
assumes *hb*: *healthy* (*wp body*)
and *sb*: *sublinear* (*wp body*)
shows *sublinear* (*iterates body G i*)
by(*induct i*, *auto intro*!:*sublinear-wp-PC sublinear-wp-Seq sublinear-wp-Skip sublinear-wp-Embed*
assms healthy-intros iterates-healthy)

From this, sub-additivity follows for the limit (i.e. the loop), by appealing to the property at all steps.

lemma *sub-add-wp-loop*:
fixes *body*::$'s$ *prog*
assumes *sb*: *sublinear* (*wp body*)

```
    and cb: bd-cts (wp body)
    and hwp: healthy (wp body)
  shows sub-add (wp (do G ⟶ body od))
proof
  fix P Q::'s expect and s::'s
  assume sP: sound P and sQ: sound Q

  from hwp cb sP have (λi. iterates body G i P s) ⇢ wp do G ⟶ body od P s
    by(rule loop-iterates)
  moreover
  from hwp cb sQ have (λi. iterates body G i Q s) ⇢ wp do G ⟶ body od Q s
    by(rule loop-iterates)
  ultimately
  have (λi. iterates body G i P s + iterates body G i Q s) ⇢
      wp do G ⟶ body od P s + wp do G ⟶ body od Q s
    by(rule tendsto-add)
  moreover {
    from sublinear-subadd[OF sublinear-iterates, OF hwp sb,
                 OF healthy-feasibleD[OF iterates-healthy, OF hwp]] sP sQ
    have ⋀i. iterates body G i P s + iterates body G i Q s ≤ iterates body G i (λs. P s + Q
s) s
      by(rule sub-addD)
  }
  moreover {
    from sP sQ have sound (λs. P s + Q s) by(blast intro:sound-intros)
    with hwp cb have (λi. iterates body G i (λs. P s + Q s) s) ⇢
                 wp do G ⟶ body od (λs. P s + Q s) s
      by(blast intro:loop-iterates)
  }
  ultimately
  show wp do G ⟶ body od P s + wp do G ⟶ body od Q s ≤ wp do G ⟶ body od (λs.
P s + Q s) s
    by(blast intro:LIMSEQ-le)
qed

lemma sublinear-wp-loop:
  fixes body::'s prog
  assumes hb: healthy (wp body)
    and nhb: nearly-healthy (wlp body)
    and sb: sublinear (wp body)
    and cb: bd-cts (wp body)
  shows sublinear (wp (do G ⟶ body od))
   using sublinear-sub-distrib[OF sb] sublinear-subadd[OF sb]
      hb healthy-feasibleD[OF hb]
   by(iprover intro:sd-sa-sublinear[OF - - healthy-wp-loop[OF hb]]
           sub-distrib-wp-loop sub-add-wp-loop assms)

lemmas sublinear-intros =
  sublinear-wp-Abort
```

sublinear-wp-Skip
sublinear-wp-Apply
sublinear-wp-Seq
sublinear-wp-PC
sublinear-wp-DC
sublinear-wp-SetPC
sublinear-wp-SetDC
sublinear-wp-Embed
sublinear-wp-repeat
sublinear-wp-Bind
sublinear-wp-loop

end

4.6 Determinism

theory *Determinism* **imports** *WellDefined* **begin**

We provide a set of lemmas for establishing that appropriately restricted programs are fully additive, and maximal in the refinement order. This is particularly useful with data refinement, as it implies correspondence.

4.6.1 Additivity

lemma *additive-wp-Abort*:
additive (*wp* (*Abort*))
by(*auto simp*:*wp-eval*)

wlp Abort is not additive.

lemma *additive-wp-Skip*:
additive (*wp* (*Skip*))
by(*auto simp*:*wp-eval*)

lemma *additive-wp-Apply*:
additive (*wp* (*Apply f*))
by(*auto simp*:*wp-eval*)

lemma *additive-wp-Seq*:
fixes *a*::*'s prog*
assumes *adda*: *additive* (*wp a*)
and *addb*: *additive* (*wp b*)
and *wb*: *well-def b*
shows *additive* (*wp* (*a* ;; *b*))
proof(*rule additiveI*, *unfold wp-eval o-def*)
fix P::*'s* ⇒ *real* **and** Q::*'s* ⇒ *real* **and** *s*::*'s*
assume *sP*: *sound P* **and** *sQ*: *sound Q*

note *hb* = *well-def-wp-healthy*[*OF wb*]

from *addb sP sQ*
have $wp\ b\ (\lambda s.\ P\ s + Q\ s) = (\lambda s.\ wp\ b\ P\ s + wp\ b\ Q\ s)$
by(*blast dest*:*additiveD*)
with *adda sP sQ hb*
show $wp\ a\ (wp\ b\ (\lambda s.\ P\ s + Q\ s))\ s =$
$wp\ a\ (wp\ b\ P)\ s + (wp\ a\ (wp\ b\ Q))\ s$
by(*auto intro*:*fun-cong*[*OF additiveD*])
qed

lemma *additive-wp-PC*:
$[\![$ *additive* (*wp a*); *additive* (*wp b*) $]\!] \Longrightarrow$ *additive* (*wp* (*a* $_P\oplus$ *b*))
by(*rule additiveI*, *simp add*:*additiveD field-simps wp-eval*)

DC is not additive.

lemma *additive-wp-SetPC*:
$[\![\bigwedge x\ s.\ x \in supp\ (p\ s) \Longrightarrow additive\ (wp\ (a\ x)); \bigwedge s.\ finite\ (supp\ (p\ s))\ ]\!] \Longrightarrow$
additive (*wp* (*SetPC a p*))
by(*rule additiveI*,
simp add:*wp-eval additiveD distrib-left sum*.*distrib*)

lemma *additive-wp-Bind*:
$[\![\bigwedge x.\ additive\ (wp\ (a\ (f\ x)))\ ]\!] \Longrightarrow additive\ (wp\ (Bind\ f\ a))$
by(*simp add*:*wp-eval additive-def*)

lemma *additive-wp-Embed*:
$[\![$ *additive t* $]\!] \Longrightarrow$ *additive* (*wp* (*Embed t*))
by(*simp add*:*wp-eval*)

lemma *additive-wp-repeat*:
additive (*wp a*) $\Longrightarrow$ *well-def a* $\Longrightarrow$ *additive* (*wp* (*repeat n a*))
by(*induct n*, *auto simp*:*additive-wp-Skip intro*:*additive-wp-Seq wd-intros*)

lemmas *fa-intros* =
additive-wp-Abort additive-wp-Skip
additive-wp-Apply additive-wp-Seq
additive-wp-PC additive-wp-SetPC
additive-wp-Bind additive-wp-Embed
additive-wp-repeat

4.6.2 Maximality

lemma *max-wp-Skip*:
maximal (*wp Skip*)
by(*simp add*:*maximal-def wp-eval*)

lemma *max-wp-Apply*:
maximal (*wp* (*Apply f*))
by(*auto simp*:*wp-eval o-def*)

lemma *max-wp-Seq*:
$[\![$ *maximal* (*wp a*); *maximal* (*wp b*) $]\!] \Longrightarrow$ *maximal* (*wp* (*a* ;; *b*))
by(*simp add*:*wp-eval maximal-def*)

lemma *max-wp-PC*:
$[\![$ *maximal* (*wp a*); *maximal* (*wp b*) $]\!] \Longrightarrow$ *maximal* (*wp* ($a\ _P\!\oplus b$))
by(*rule maximalI*, *simp add*:*maximalD field-simps wp-eval*)

lemma *max-wp-DC*:
$[\![$ *maximal* (*wp a*); *maximal* (*wp b*) $]\!] \Longrightarrow$ *maximal* (*wp* ($a \sqcap b$))
by(*rule maximalI*, *simp add*:*wp-eval maximalD*)

lemma *max-wp-SetPC*:
$[\![\bigwedge s\ a.\ a \in supp\ (P\ s) \Longrightarrow maximal\ (wp\ (p\ a));\ \bigwedge s.\ (\sum a{\in}supp\ (P\ s).\ P\ s\ a) = 1\ ]\!] \Longrightarrow$
maximal (*wp* (*SetPC p P*))
by(*auto simp*:*maximalD wp-def SetPC-def sum-distrib-right*[*symmetric*])

lemma *max-wp-SetDC*:
fixes $p{::}'a \Rightarrow\ 's\ prog$
assumes *mp*: $\bigwedge s\ a.\ a \in S\ s \Longrightarrow maximal\ (wp\ (p\ a))$
and *ne*: $\bigwedge s.\ S\ s \neq \{\}$
shows *maximal* (*wp* (*SetDC p S*))
proof(*rule maximalI*, *rule ext*, *unfold wp-eval*)
fix *c*::*real* **and** *s*::*'s*
assume $0 \leq c$
hence $Inf\ ((\lambda a.\ wp\ (p\ a)\ (\lambda\text{-}.\ c)\ s)\ `\ S\ s) = Inf\ ((\lambda\text{-}.\ c)\ `\ S\ s)$
using *mp* **by**(*simp add*:*maximalD cong*:*image-cong*)
also {
from *ne* **obtain** *a* **where** $a \in S\ s$ **by** *blast*
hence $Inf\ ((\lambda\text{-}.\ c)\ `\ S\ s) = c$
by (*auto simp add*: *image-constant-conv cong del*: *INF-cong-simp*)
}
finally show $Inf\ ((\lambda a.\ wp\ (p\ a)\ (\lambda\text{-}.\ c)\ s)\ `\ S\ s) = c$ **.**
qed

lemma *max-wp-Embed*:
maximal t $\Longrightarrow$ *maximal* (*wp* (*Embed t*))
by(*simp add*:*wp-eval*)

lemma *max-wp-repeat*:
maximal (*wp a*) $\Longrightarrow$ *maximal* (*wp* (*repeat n a*))
by(*induct n*, *simp-all add*:*max-wp-Skip max-wp-Seq*)

lemma *max-wp-Bind*:
assumes *ma*: $\bigwedge s.\ maximal\ (wp\ (a\ (f\ s)))$
shows *maximal* (*wp* (*Bind f a*))
proof(*rule maximalI*, *rule ext*, *simp add*:*wp-eval*)
fix *c*::*real* **and** *s*

```
  assume 0 ≤ c
  with ma have wp (a (f s)) (λ-. c) = (λ-. c) by(blast)
  thus wp (a (f s)) (λ-. c) s = c by(auto)
qed

lemmas max-intros =
  max-wp-Skip  max-wp-Apply
  max-wp-Seq   max-wp-PC
  max-wp-DC    max-wp-SetPC
  max-wp-SetDC max-wp-Embed
  max-wp-Bind  max-wp-repeat
```

A healthy transformer that terminates is maximal.

```
lemma healthy-term-max:
  assumes ht: healthy t
    and trm: λs. 1 ⊩ t (λs. 1)
  shows maximal t
proof(intro maximalI ext)
  fix c::real and s
  assume nnc: 0 ≤ c

  have t (λs. c) s = t (λs. 1 ∗ c) s by(simp)
  also from nnc healthy-scalingD[OF ht]
  have ... = c ∗ t (λs. 1) s by(simp add:scalingD)
  also {
    from ht have t (λs. 1) ⊩ λs. 1 by(auto)
    with trm have t (λs. 1) = (λs. 1) by(auto)
    hence c ∗ t (λs. 1) s = c by(simp)
  }
  finally show t (λs. c) s = c .
qed
```

4.6.3 Determinism

```
lemma det-wp-Skip:
  determ (wp Skip)
  using max-intros fa-intros by(blast)

lemma det-wp-Apply:
  determ (wp (Apply f))
  by(intro determI fa-intros max-intros)

lemma det-wp-Seq:
  determ (wp a) ⟹ determ (wp b) ⟹ well-def b ⟹ determ (wp (a ;; b))
  by(intro determI fa-intros max-intros, auto)

lemma det-wp-PC:
  determ (wp a) ⟹ determ (wp b) ⟹ determ (wp (a P⊕ b))
  by(intro determI fa-intros max-intros, auto)
```

lemma *det-wp-SetPC*:
$(\bigwedge x\ s.\ x \in supp\ (p\ s) \Longrightarrow determ\ (wp\ (a\ x))) \Longrightarrow$
$(\bigwedge s.\ finite\ (supp\ (p\ s))) \Longrightarrow$
$(\bigwedge s.\ sum\ (p\ s)\ (supp\ (p\ s)) = 1) \Longrightarrow$
$determ\ (wp\ (SetPC\ a\ p))$
by(*intro determI fa-intros max-intros*, *auto*)

lemma *det-wp-Bind*:
$(\bigwedge x.\ determ\ (wp\ (a\ (f\ x)))) \Longrightarrow determ\ (wp\ (Bind\ f\ a))$
by(*intro determI fa-intros max-intros*, *auto*)

lemma *det-wp-Embed*:
$determ\ t \Longrightarrow determ\ (wp\ (Embed\ t))$
by(*simp add*:*wp-eval*)

lemma *det-wp-repeat*:
$determ\ (wp\ a) \Longrightarrow well\text{-}def\ a \Longrightarrow determ\ (wp\ (repeat\ n\ a))$
by(*intro determI fa-intros max-intros*, *auto*)

lemmas *determ-intros* =
det-wp-Skip det-wp-Apply
det-wp-Seq det-wp-PC
det-wp-SetPC det-wp-Bind
det-wp-Embed det-wp-repeat

end

4.7 Well-Defined Programs.

theory *WellDefined* **imports**
Healthiness
Sublinearity
LoopInduction
begin

The definition of a well-defined program collects the various notions of healthiness and well-behavedness that we have so far established: healthiness of the strict and liberal transformers, continuity and sublinearity of the strict transformers, and two new properties. These are that the strict transformer always lies below the liberal one (i.e. that it is at least as *strict*, recalling the standard embedding of a predicate), and that expectation conjunction is distributed between then in a particular manner, which will be crucial in establishing the loop rules.

4.7.1 Strict Implies Liberal

This establishes the first connection between the strict and liberal interpretations (*wp* and *wlp*).

```
definition
  wp-under-wlp :: 's prog ⇒ bool
where
  wp-under-wlp prog ≡ ∀ P. unitary P ⟶ wp prog P ⊩ wlp prog P

lemma wp-under-wlpI[intro]:
  ⟦ ⋀P. unitary P ⟹ wp prog P ⊩ wlp prog P ⟧ ⟹ wp-under-wlp prog
  unfolding wp-under-wlp-def by(simp)

lemma wp-under-wlpD[dest]:
  ⟦ wp-under-wlp prog; unitary P ⟧ ⟹ wp prog P ⊩ wlp prog P
  unfolding wp-under-wlp-def by(simp)

lemma wp-under-le-trans:
  wp-under-wlp a ⟹ le-utrans (wp a) (wlp a)
  by(blast)

lemma wp-under-wlp-Abort:
  wp-under-wlp Abort
  by(rule wp-under-wlpI, unfold wp-eval, auto)

lemma wp-under-wlp-Skip:
  wp-under-wlp Skip
  by(rule wp-under-wlpI, unfold wp-eval, blast)

lemma wp-under-wlp-Apply:
  wp-under-wlp (Apply f)
  by(auto simp:wp-eval)

lemma wp-under-wlp-Seq:
  assumes h-wlp-a: nearly-healthy (wlp a)
    and h-wp-b:  healthy (wp b)
    and h-wlp-b: nearly-healthy (wlp b)
    and wp-u-a:  wp-under-wlp a
    and wp-u-b:  wp-under-wlp b
  shows wp-under-wlp (a ;; b)
proof(rule wp-under-wlpI, unfold wp-eval o-def)
  fix P::'a ⇒ real assume uP: unitary P
  with h-wp-b have unitary (wp b P) by(blast)
  with wp-u-a have wp a (wp b P) ⊩ wlp a (wp b P) by(auto)
  also {
   from wp-u-b and uP have wp b P ⊩ wlp b P by(blast)
   with h-wlp-a and h-wlp-b and h-wp-b and uP
   have wlp a (wp b P) ⊩ wlp a (wlp b P)
     by(blast intro:nearly-healthy-monoD[OF h-wlp-a])
  }
  finally show wp a (wp b P) ⊩ wlp a (wlp b P) .
qed
```

lemma *wp-under-wlp-PC*:
assumes *h-wp-a*: *healthy* (*wp a*)
and *h-wlp-a*: *nearly-healthy* (*wlp a*)
and *h-wp-b*: *healthy* (*wp b*)
and *h-wlp-b*: *nearly-healthy* (*wlp b*)
and *wp-u-a*: *wp-under-wlp a*
and *wp-u-b*: *wp-under-wlp b*
and *uP*: *unitary P*
shows *wp-under-wlp* ($a\ {}_P\!\oplus\ b$)
proof(*rule wp-under-wlpI*, *unfold wp-eval*, *rule le-funI*)
fix $Q::'a \Rightarrow real$ **and** *s*
assume *uQ*: *unitary Q*
from *uP* **have** $P\ s \leq 1$ **by**(*blast*)
hence $0 \leq 1 - P\ s$ **by**(*simp*)
moreover
from *uQ* **and** *wp-u-b* **have** $wp\ b\ Q\ s \leq wlp\ b\ Q\ s$ **by**(*blast*)
ultimately
have $(1 - P\ s) * wp\ b\ Q\ s \leq (1 - P\ s) * wlp\ b\ Q\ s$
by(*blast intro*:*mult-left-mono*)

moreover {
from *uQ* **and** *wp-u-a* **have** $wp\ a\ Q\ s \leq wlp\ a\ Q\ s$ **by**(*blast*)
with *uP* **have** $P\ s * wp\ a\ Q\ s \leq P\ s * wlp\ a\ Q\ s$
by(*blast intro*:*mult-left-mono*)
}

ultimately
show $P\ s * wp\ \ a\ Q\ s + (1 - P\ s) * wp\ \ b\ Q\ s \leq$
$P\ s * wlp\ a\ Q\ s + (1 - P\ s) * wlp\ b\ Q\ s$
by(*blast intro*: *add-mono*)
qed

lemma *wp-under-wlp-DC*:
assumes *wp-u-a*: *wp-under-wlp a*
and *wp-u-b*: *wp-under-wlp b*
shows *wp-under-wlp* ($a \sqcap b$)
proof(*rule wp-under-wlpI*, *unfold wp-eval*, *rule le-funI*)
fix $Q::'a \Rightarrow real$ **and** *s*
assume *uQ*: *unitary Q*

from *wp-u-a uQ* **have** $wp\ a\ Q\ s \leq wlp\ a\ Q\ s$ **by**(*blast*)
moreover
from *wp-u-b uQ* **have** $wp\ b\ Q\ s \leq wlp\ b\ Q\ s$ **by**(*blast*)
ultimately
show $min\ (wp\ a\ Q\ s)\ (wp\ b\ Q\ s) \leq min\ (wlp\ a\ Q\ s)\ (wlp\ b\ Q\ s)$
by(*auto*)
qed

lemma *wp-under-wlp-SetPC*:

```
  assumes wp-u-f: ⋀s a. a ∈ supp (P s) ⟹ wp-under-wlp (f a)
    and nP:     ⋀s a. a ∈ supp (P s) ⟹ 0 ≤ P s a
  shows wp-under-wlp (SetPC f P)
proof(rule wp-under-wlpI, unfold wp-eval, rule le-funI)
  fix Q::'a ⇒ real and s
  assume uQ: unitary Q

  from wp-u-f uQ nP
  show (∑ a∈supp (P s). P s a ∗ wp (f a) Q s) ≤ (∑ a∈supp (P s). P s a ∗ wlp (f a) Q s)
    by(auto intro!:sum-mono mult-left-mono)
qed

lemma wp-under-wlp-SetDC:
  assumes wp-u-f: ⋀s a. a ∈ S s ⟹ wp-under-wlp (f a)
    and hf:     ⋀s a. a ∈ S s ⟹ healthy (wp (f a))
    and nS:     ⋀s. S s ≠ {}
  shows wp-under-wlp (SetDC f S)
proof(rule wp-under-wlpI, rule le-funI, unfold wp-eval)
  fix Q::'a ⇒ real and s
  assume uQ: unitary Q

  show Inf ((λa. wp (f a) Q s) ‘ S s) ≤ Inf ((λa. wlp (f a) Q s) ‘ S s)
  proof(rule cInf-mono)
    from nS show (λa. wlp (f a) Q s) ‘ S s ≠ {} by(blast)

    fix x assume xin: x ∈ (λa. wlp (f a) Q s) ‘ S s
    then obtain a where ain: a ∈ S s and xrw: x = wlp (f a) Q s
      by(blast)
    with wp-u-f uQ
    have wp (f a) Q s ≤ wlp (f a) Q s by(blast)
    moreover from ain have wp (f a) Q s ∈ (λa. wp (f a) Q s) ‘ S s
      by(blast)
    ultimately show ∃ y∈ (λa. wp (f a) Q s) ‘ S s. y ≤ x
      by(auto simp:xrw)

  next
    fix y assume yin: y ∈ (λa. wp (f a) Q s) ‘ S s
    then obtain a where ain: a ∈ S s and yrw: y = wp (f a) Q s
      by(blast)
    with hf uQ have unitary (wp (f a) Q) by(auto)
    with yrw show 0 ≤ y by(auto)
  qed
qed

lemma wp-under-wlp-Embed:
  wp-under-wlp (Embed t)
  by(rule wp-under-wlpI, unfold wp-eval, blast)

lemma wp-under-wlp-loop:
```

```
  fixes body::'s prog
  assumes hwp: healthy (wp body)
    and hwlp: nearly-healthy (wlp body)
    and wp-under: wp-under-wlp body
  shows wp-under-wlp (do G ⟶ body od)
proof(rule wp-under-wlpI)
  fix P::'s expect
  assume uP: unitary P hence sP: sound P by(auto)

  let ?X Q s = «G» s * wp body Q s + «𝒩 G» s * P s
  let ?Y Q s = «G» s * wlp body Q s + «𝒩 G» s * P s

  show wp (do G ⟶ body od) P ⊩ wlp (do G ⟶ body od) P
  proof(simp add:hwp hwlp sP uP wp-Loop1 wlp-Loop1, rule gfp-exp-upperbound)
   thm lfp-loop-fp
   from hwp sP have lfp-exp ?X = ?X (lfp-exp ?X)
    by(rule lfp-wp-loop-unfold)
   hence lfp-exp ?X ⊩ ?X (lfp-exp ?X) by(simp)
   also {
    from hwp uP have wp body (lfp-exp ?X) ⊩ wlp body (lfp-exp ?X)
     by(auto intro:wp-under-wlpD[OF wp-under] lfp-loop-unitary)
    hence ?X (lfp-exp ?X) ⊩ ?Y (lfp-exp ?X)
     by(auto intro:add-mono mult-left-mono)
   }
   finally show lfp-exp ?X ⊩ ?Y (lfp-exp ?X) .
   from hwp uP show unitary (lfp-exp ?X)
    by(auto intro:lfp-loop-unitary)
  qed
qed

lemma wp-under-wlp-repeat:
  ⟦ healthy (wp a); nearly-healthy (wlp a); wp-under-wlp a ⟧ ⟹
   wp-under-wlp (repeat n a)
  by(induct n, auto intro!:wp-under-wlp-Skip wp-under-wlp-Seq healthy-intros)

lemma wp-under-wlp-Bind:
  ⟦ ⋀s. wp-under-wlp (a (f s)) ⟧ ⟹ wp-under-wlp (Bind f a)
  unfolding wp-under-wlp-def by(auto simp:wp-eval)

lemmas wp-under-wlp-intros =
  wp-under-wlp-Abort wp-under-wlp-Skip
  wp-under-wlp-Apply wp-under-wlp-Seq
  wp-under-wlp-PC    wp-under-wlp-DC
  wp-under-wlp-SetPC wp-under-wlp-SetDC
  wp-under-wlp-Embed wp-under-wlp-loop
  wp-under-wlp-repeat wp-under-wlp-Bind
```

4.7.2 Sub-Distributivity of Conjunction

```
definition
  sub-distrib-pconj :: 's prog ⇒ bool
where
  sub-distrib-pconj prog ≡
  ∀ P Q. unitary P ⟶ unitary Q ⟶
       wlp prog P && wp prog Q ⊩ wp prog (P && Q)

lemma sub-distrib-pconjI[intro]:
  ⟦⋀P Q. ⟦ unitary P; unitary Q ⟧ ⟹ wlp prog P && wp prog Q ⊩ wp prog (P && Q) ⟧
⟹
    sub-distrib-pconj prog
  unfolding sub-distrib-pconj-def by(simp)

lemma sub-distrib-pconjD[dest]:
  ⋀P Q. ⟦ sub-distrib-pconj prog; unitary P; unitary Q ⟧ ⟹
   wlp prog P && wp prog Q ⊩ wp prog (P && Q)
  unfolding sub-distrib-pconj-def by(simp)

lemma sdp-Abort:
  sub-distrib-pconj Abort
  by(rule sub-distrib-pconjI, unfold wp-eval, auto intro:exp-conj-rzero)

lemma sdp-Skip:
  sub-distrib-pconj Skip
  by(rule sub-distrib-pconjI, simp add:wp-eval)

lemma sdp-Seq:
  fixes a and b
  assumes sdp-a:   sub-distrib-pconj a
    and sdp-b:   sub-distrib-pconj b
    and h-wp-a:  healthy (wp a)
    and h-wp-b:  healthy (wp b)
    and h-wlp-b: nearly-healthy (wlp b)
  shows sub-distrib-pconj (a ;; b)
proof(rule sub-distrib-pconjI, unfold wp-eval o-def)
  fix P::'a ⇒ real and Q::'a ⇒ real
  assume uP: unitary P and uQ: unitary Q

  with h-wp-b and h-wlp-b
  have wlp a (wlp b P) && wp a (wp b Q) ⊩ wp a (wlp b P && wp b Q)
    by(blast intro!:sub-distrib-pconjD[OF sdp-a])
  also {
    from sdp-b and uP and uQ
    have wlp b P && wp b Q ⊩ wp b (P && Q) by(blast)
    with h-wp-a h-wp-b h-wlp-b uP uQ
    have wp a (wlp b P && wp b Q) ⊩ wp a (wp b (P && Q))
      by(blast intro!:mono-transD[OF healthy-monoD, OF h-wp-a] unitary-sound
                unitary-intros sound-intros)
```

```
 }
 finally show wlp a (wlp b P) && wp a (wp b Q) ⊩ wp a (wp b (P && Q)) .
qed

lemma sdp-Apply:
 sub-distrib-pconj (Apply f)
 by(rule sub-distrib-pconjI, simp add:wp-eval)

lemma sdp-DC:
 fixes a::'s prog and b
 assumes sdp-a:  sub-distrib-pconj a
   and sdp-b:  sub-distrib-pconj b
   and h-wp-a:  healthy (wp a)
   and h-wp-b:  healthy (wp b)
   and h-wlp-b: nearly-healthy (wlp b)
 shows sub-distrib-pconj (a ⨅ b)
proof(rule sub-distrib-pconjI, unfold wp-eval, rule le-funI)
 fix P::'s ⇒ real and Q::'s ⇒ real and s::'s
 assume uP: unitary P and uQ: unitary Q

 have ((λs. min (wlp a P s) (wlp b P s)) &&
     (λs. min (wp a Q s) (wp b Q s))) s ≤
    min (wlp a P s .& wp a Q s) (wlp b P s .& wp b Q s)
  unfolding exp-conj-def by(rule min-pconj)
 also {
  have (λs. wlp a P s .& wp a Q s) = wlp a P && wp a Q
   by(simp add:exp-conj-def)
  also from sdp-a uP uQ have ... ⊩ wp a (P && Q)
   by(blast dest:sub-distrib-pconjD)
  finally have wlp a P s .& wp a Q s ≤ wp a (P && Q) s
   by(rule le-funD)
  moreover {
   have (λs. wlp b P s .& wp b Q s) = wlp b P && wp b Q
    by(simp add:exp-conj-def)
  also from sdp-b uP uQ have ... ⊩ wp b (P && Q)
   by(blast)
  finally have wlp b P s .& wp b Q s ≤ wp b (P && Q) s
   by(rule le-funD)
  }
  ultimately
  have min (wlp a P s .& wp a Q s) (wlp b P s .& wp b Q s) ≤
      min (wp a (P && Q) s) (wp b (P && Q) s) by(auto)
 }
 finally
 show ((λs. min (wlp a P s) (wlp b P s)) &&
     (λs. min (wp a Q s) (wp b Q s))) s ≤
    min (wp a (P && Q) s) (wp b (P && Q) s) .
qed
```

```
lemma sdp-PC:
  fixes a::'s prog and b
  assumes sdp-a:   sub-distrib-pconj a
     and sdp-b:   sub-distrib-pconj b
     and h-wp-a:  healthy (wp a)
     and h-wp-b:  healthy (wp b)
     and h-wlp-b: nearly-healthy (wlp b)
     and uP:      unitary P
  shows sub-distrib-pconj (a P⊕ b)
proof(rule sub-distrib-pconjI, unfold wp-eval, rule le-funI)
  fix Q::'s ⇒ real and R::'s ⇒ real and s::'s
  assume uQ: unitary Q and uR: unitary R

  have nnA: 0 ≤ P s and nnB: P s ≤ 1
    using uP by auto
  note nn = nnA nnB

  have ((λs. P s ∗ wlp a Q s + (1 − P s) ∗ wlp b Q s) &&
       (λs. P s ∗  wp a R s + (1 − P s) ∗  wp b R s)) s =
      ((P s ∗ wlp a Q s + (1 − P s) ∗ wlp b Q s) +
       (P s ∗  wp a R s + (1 − P s) ∗  wp b R s)) ⊖ 1
    by(simp add:exp-conj-def pconj-def)
  also have ... = P s ∗       (wlp a Q s + wp a R s) +
             (1 − P s) ∗ (wlp b Q s + wp b R s) ⊖ 1
    by(simp add:field-simps)
  also have ... = P s ∗       (wlp a Q s + wp a R s) +
             (1 − P s) ∗ (wlp b Q s + wp b R s) ⊖
             (P s + (1 − P s))
    by(simp)
  also have ... ≤ (P s ∗       (wlp a Q s + wp a R s) ⊖ P s) +
             ((1 − P s) ∗ (wlp b Q s + wp b R s) ⊖ (1 − P s))
    by(rule tminus-add-mono)
  also have ... = (P s      ∗ (wlp a Q s + wp a R s ⊖ 1)) +
             ((1 − P s) ∗ (wlp b Q s + wp b R s ⊖ 1))
    by(simp add:nn tminus-left-distrib)
  also have ... = P s ∗       ((wlp a Q && wp a R) s) +
             (1 − P s) ∗ ((wlp b Q && wp b R) s)
    by(simp add:exp-conj-def pconj-def)
  also {
    from sdp-a sdp-b uQ uR
    have P s ∗ (wlp a Q && wp a R) s ≤ P s ∗ wp a (Q && R) s
     and (1 − P s) ∗ (wlp b Q && wp b R) s ≤ (1 − P s) ∗ wp b (Q && R) s
      by (simp-all add: entailsD mult-left-mono nn sub-distrib-pconjD)
    hence P s ∗      ((wlp a Q && wp a R) s) +
        (1 − P s) ∗ ((wlp b Q && wp b R) s) ≤
        P s ∗ wp a (Q && R) s + (1 − P s) ∗ wp b (Q && R) s
      by(auto)
  }
  finally show ((λs. P s ∗ wlp a Q s + (1 − P s) ∗ wlp b Q s) &&
```

```
      (λs. P s * wp a R s + (1 − P s) * wp b R s)) s ≤
      P s * wp a (Q && R) s + (1 − P s) * wp b (Q && R) s .
qed

lemma sdp-Embed:
 ⟦ ⋀P Q. ⟦ unitary P; unitary Q ⟧ ⟹ t P && t Q ⊩ t (P && Q) ⟧ ⟹
  sub-distrib-pconj (Embed t)
 by(auto simp:wp-eval)

lemma sdp-repeat:
 fixes a::'s prog
 assumes sdpa: sub-distrib-pconj a
    and hwp: healthy (wp a) and hwlp: nearly-healthy (wlp a)
 shows sub-distrib-pconj (repeat n a) (is ?X n)
proof(induct n)
 show ?X 0 by(simp add:sdp-Skip)
 fix n assume IH: ?X n
 show ?X (Suc n)
 proof(rule sub-distrib-pconjI, simp add:wp-eval)
  fix P::'s ⇒ real and Q::'s ⇒ real
  assume uP: unitary P and uQ: unitary Q
  from assms have hwlpa: nearly-healthy (wlp (repeat n a))
          and hwpa: healthy (wp (repeat n a))
   by(auto intro:healthy-intros)
  from uP and hwlpa have unitary (wlp (repeat n a) P) by(blast)
  moreover from uQ and hwpa have unitary (wp (repeat n a) Q) by(blast)
  ultimately
  have wlp a (wlp (repeat n a) P) && wp a (wp (repeat n a) Q) ⊩
     wp a (wlp (repeat n a) P && wp (repeat n a) Q)
   using sdpa by(blast)
  also {
   from hwlp have nearly-healthy (wlp (repeat n a)) by(rule healthy-intros)
   with uP have sound (wlp (repeat n a) P) by(auto)
   moreover from hwp uQ have sound (wp (repeat n a) Q)
    by(auto intro:healthy-intros)
   ultimately have sound (wlp (repeat n a) P && wp (repeat n a) Q)
    by(rule exp-conj-sound)
   moreover {
    from uP uQ have sound (P && Q) by(auto intro:exp-conj-sound)
    with hwp have sound (wp (repeat n a) (P && Q))
     by(auto intro:healthy-intros)
   }
   moreover from uP uQ IH
   have wlp (repeat n a) P && wp (repeat n a) Q ⊩ wp (repeat n a) (P && Q)
    by(blast)
   ultimately
   have wp a (wlp (repeat n a) P && wp (repeat n a) Q) ⊩
      wp a (wp (repeat n a) (P && Q))
    by(rule mono-transD[OF healthy-monoD, OF hwp])
```

```
    }
    finally show wlp a (wlp (repeat n a) P) && wp a (wp (repeat n a) Q) ⊩
               wp a (wp (repeat n a) (P && Q)) .
  qed
qed

lemma sdp-SetPC:
  fixes p::'a ⇒ 's prog
  assumes sdp: ⋀s a. a ∈ supp (P s) ⟹ sub-distrib-pconj (p a)
      and fin: ⋀s. finite (supp (P s))
      and nnp: ⋀s a. 0 ≤ P s a
      and sub: ⋀s. sum (P s) (supp (P s)) ≤ 1
  shows sub-distrib-pconj (SetPC p P)
proof(rule sub-distrib-pconjI, simp add:wp-eval, rule le-funI)
  fix Q::'s ⇒ real and R::'s ⇒ real and s::'s
  assume uQ: unitary Q and uR: unitary R
  have ((λs. ∑a∈supp (P s). P s a * wlp (p a) Q s) &&
        (λs. ∑a∈supp (P s). P s a * wp (p a) R s)) s =
       (∑a∈supp (P s). P s a * wlp (p a) Q s) + (∑a∈supp (P s). P s a * wp (p a) R s) ⊖
1
    by(simp add:exp-conj-def pconj-def)
  also have ... = (∑a∈supp (P s). P s a * (wlp (p a) Q s + wp (p a) R s)) ⊖ 1
    by(simp add: sum.distrib field-simps)
  also from sub
  have ... ≤ (∑a∈supp (P s). P s a * (wlp (p a) Q s + wp (p a) R s)) ⊖
            (∑a∈supp (P s). P s a)
    by(rule tminus-right-antimono)
  also from fin
  have ... ≤ (∑a∈supp (P s). P s a * (wlp (p a) Q s + wp (p a) R s) ⊖ P s a)
    by(rule tminus-sum-mono)
  also from nnp
  have ... = (∑a∈supp (P s). P s a * (wlp (p a) Q s + wp (p a) R s ⊖ 1))
    by(simp add:tminus-left-distrib)
  also have ... = (∑a∈supp (P s). P s a * (wlp (p a) Q && wp (p a) R) s)
    by(simp add:pconj-def exp-conj-def)
  also {
    from sdp uQ uR
    have ⋀a. a ∈ supp (P s) ⟹ wlp (p a) Q && wp (p a) R ⊩ wp (p a) (Q && R)
      by(blast intro:sub-distrib-pconjD)
    with nnp
    have (∑a∈supp (P s). P s a * (wlp (p a) Q && wp (p a) R) s) ≤
         (∑a∈supp (P s). P s a * (wp (p a) (Q && R)) s)
      by(blast intro:sum-mono mult-left-mono)
  }
  finally show ((λs. ∑a∈supp (P s). P s a * wlp (p a) Q s) &&
               (λs. ∑a∈supp (P s). P s a * wp (p a) R s)) s ≤
              (∑a∈supp (P s). P s a * wp (p a) (Q && R) s) .
qed
```

```
lemma sdp-SetDC:
  fixes p::'a ⇒ 's prog
  assumes sdp: ⋀s a. a ∈ S s ⟹ sub-distrib-pconj (p a)
      and hwp: ⋀s a. a ∈ S s ⟹ healthy (wp (p a))
      and hwlp: ⋀s a. a ∈ S s ⟹ nearly-healthy (wlp (p a))
      and ne: ⋀s. S s ≠ {}
  shows sub-distrib-pconj (SetDC p S)
proof(rule sub-distrib-pconjI, rule le-funI)
  fix P::'s ⇒ real and Q::'s ⇒ real and s::'s
  assume uP: unitary P and uQ: unitary Q

  from uP hwlp
  have ⋀x. x ∈ (λa. wlp (p a) P) ‘ S s ⟹ unitary x by(auto)
  hence ⋀y. y ∈ (λa. wlp (p a) P s) ‘ S s ⟹ 0 ≤ y by(auto)
  hence ⋀a. a ∈ S s ⟹ wlp (SetDC p S) P s ≤ wlp (p a) P s
    unfolding wp-eval by(intro cInf-lower bdd-belowI, auto)
  moreover {
    from uQ hwp have ⋀a. a ∈ S s ⟹  0 ≤ wp (p a) Q s by(blast)
    hence ⋀a. a ∈ S s ⟹ wp (SetDC p S) Q s ≤ wp (p a) Q s
    unfolding wp-eval by(intro cInf-lower bdd-belowI, auto)
  }
  ultimately
  have ⋀a. a ∈ S s ⟹ wlp (SetDC p S) P s + wp (SetDC p S) Q s ⊖ 1 ≤
                wlp (p a) P s + wp (p a) Q s ⊖ 1
    by(auto intro:tminus-left-mono add-mono)
  also have ⋀a. wlp (p a) P s + wp (p a) Q s ⊖ 1 = (wlp (p a) P && wp (p a) Q) s
    by(simp add:exp-conj-def pconj-def)
  also from sdp uP uQ
  have ⋀a. a ∈ S s ⟹ ... a ≤ wp (p a) (P && Q) s
    by(blast)
  also have ⋀a. ... a = wp (p a) (λs. P s + Q s ⊖ 1) s
    by(simp add:exp-conj-def pconj-def)
  finally
  show (wlp (SetDC p S) P && wp (SetDC p S) Q) s ≤ wp (SetDC p S) (P && Q) s
    unfolding exp-conj-def pconj-def wp-eval
    using ne by(blast intro!:cInf-greatest)
qed

lemma sdp-Bind:
  ⟦ ⋀s. sub-distrib-pconj (p (f s)) ⟧ ⟹ sub-distrib-pconj (Bind f p)
  unfolding sub-distrib-pconj-def wp-eval exp-conj-def pconj-def
  by(blast)
```

For loops, we again appeal to our transfinite induction principle, this time taking advantage of the simultaneous treatment of both strict and liberal transformers.

```
lemma sdp-loop:
  fixes body::'s prog
  assumes sdp-body: sub-distrib-pconj body
      and hwlp: nearly-healthy (wlp body)
```

```
    and hwp: healthy (wp body)
  shows sub-distrib-pconj (do G ⟶ body od)
proof(rule sub-distrib-pconjI, rule loop-induct[OF hwp hwlp])
  fix P Q::'s expect and S::('s trans × 's trans) set
  assume uP: unitary P and uQ: unitary Q
    and ffst: ∀x∈S. feasible (fst x)
    and usnd: ∀x∈S. ∀Q. unitary Q ⟶ unitary (snd x Q)
    and IH: ∀x∈S. snd x P && fst x Q ⊩ fst x (P && Q)

  show Inf-utrans (snd ‘ S) P && Sup-trans (fst ‘ S) Q ⊩
            Sup-trans (fst ‘ S) (P && Q)
  proof(cases)
    assume S = {}
    thus ?thesis
      by(simp add:Inf-trans-def Sup-trans-def Inf-utrans-def
            Inf-exp-def Sup-exp-def exp-conj-def)
  next
    assume ne: S ≠ {}

    let ?f s = 1 + Sup-trans (fst ‘ S) (P && Q) s − Inf-utrans (snd ‘ S) P s

    from ne obtain t where tin: t ∈ fst ‘ S by(auto)
    from ne obtain u where uin: u ∈ snd ‘ S by(auto)

    from tin ffst uP uQ have utPQ: unitary (t (P && Q))
      by(auto intro:exp-conj-unitary)
    hence ⋀s. 0 ≤ t (P && Q) s by(auto)
    also {
      from ffst tin have le: le-utrans t (Sup-trans (fst ‘ S))
        by(auto intro:Sup-trans-upper)
      with uP uQ have ⋀s. t (P && Q) s ≤ Sup-trans (fst ‘ S) (P && Q) s
        by(auto intro:exp-conj-unitary)
    }
    finally have nn-rhs: ⋀s. 0 ≤ Sup-trans (fst ‘ S) (P && Q) s .

    have ⋀R. Inf-utrans (snd ‘ S) P && R ⊩ Sup-trans (fst ‘ S) (P && Q) ⟹ R ≤ ?f
    proof(rule contrapos-pp, assumption)
      fix R
      assume ¬ R ≤ ?f
      then obtain s where ¬ R s ≤ ?f s by(auto)
      hence gt: ?f s < R s by(simp)

      from nn-rhs have g1: 1 ≤ 1 + Sup-trans (fst ‘ S) (P && Q) s by(auto)
      hence Sup-trans (fst ‘ S) (P && Q) s = Inf-utrans (snd ‘ S) P s .& ?f s
        by(simp add:pconj-def)
      also from g1 have ... = Inf-utrans (snd ‘ S) P s + ?f s − 1
        by(simp)
      also from gt have ... < Inf-utrans (snd ‘ S) P s + R s − 1
        by(simp)
```

```
    also {
      with g1 have 1 ≤ Inf-utrans (snd ‘ S) P s + R s
        by(simp)
      hence Inf-utrans (snd ‘ S) P s + R s − 1 = Inf-utrans (snd ‘ S) P s .& R s
        by(simp add:pconj-def)
    }
    finally
    have ¬ (Inf-utrans (snd ‘ S) P && R) s ≤ Sup-trans (fst ‘ S) (P && Q) s
      by(simp add:exp-conj-def)
    thus ¬ Inf-utrans (snd ‘ S) P && R ⊩ Sup-trans (fst ‘ S) (P && Q)
      by(auto)
  qed

  moreover have ∀ t∈fst ‘ S. Inf-utrans (snd ‘ S) P && t Q ⊩ Sup-trans (fst ‘ S) (P &&
Q)
  proof
    fix t assume tin: t ∈ fst ‘ S
    then obtain x where xin: x ∈ S and fx: t = fst x by(auto)

    from xin have snd x ∈ snd ‘ S by(auto)
    with uP usnd have Inf-utrans (snd ‘ S) P ⊩ snd x P
      by(auto intro:le-utransD[OF Inf-utrans-lower])
    hence Inf-utrans (snd ‘ S) P && fst x Q ⊩ snd x P && fst x Q
      by(auto intro:entails-frame)
    also from xin IH have ... ⊩ fst x (P && Q)
      by(auto)
    also from xin ffst exp-conj-unitary[OF uP uQ]
    have ... ⊩ Sup-trans (fst ‘ S) (P && Q)
      by(auto intro:le-utransD[OF Sup-trans-upper])
    finally show Inf-utrans (snd ‘ S) P && t Q ⊩ Sup-trans (fst ‘ S) (P && Q)
      by(simp add:fx)
  qed
  ultimately have bt: ∀ t∈fst ‘ S. t Q ⊩ ?f by(blast)

  have Sup-trans (fst ‘ S) Q = Sup-exp {t Q |t. t ∈ fst ‘ S}
    by(simp add:Sup-trans-def)
  also have ... ⊩ ?f
  proof(rule Sup-exp-least)
    from bt show ∀ R∈{t Q |t. t ∈ fst ‘ S}. R ⊩ ?f by(blast)
    from ne obtain t where tin: t ∈ fst ‘ S by(auto)
    with ffst uQ have unitary (t Q) by(auto)
    hence λs. 0 ⊩ t Q by(auto)
    also from tin bt have ... ⊩ ?f by(auto)
    finally show nneg (λs. 1 + Sup-trans (fst ‘ S) (P && Q) s −
                  Inf-utrans (snd ‘ S) P s)
      by(auto)
  qed
  finally have Inf-utrans (snd ‘ S) P && Sup-trans (fst ‘ S) Q ⊩
           Inf-utrans (snd ‘ S) P && ?f
```

```
    by(auto intro:entails-frame)
  also from nn-rhs have ... ⊩ Sup-trans (fst ‘ S) (P && Q)
    by(simp add:exp-conj-def pconj-def)
  finally show ?thesis .
 qed

next
 fix P Q::'s expect and t u::'s trans
 assume uP: unitary P and uQ: unitary Q
    and ft: feasible t
    and uu: ⋀Q. unitary Q ⟹ unitary (u Q)
    and IH: u P && t Q ⊩ t (P && Q)
 show wlp (body ;; Embed u « G »⊕ Skip) P &&
      wp  (body ;; Embed t « G »⊕ Skip) Q ⊩
      wp  (body ;; Embed t « G »⊕ Skip) (P && Q)
 proof(rule le-funI, simp add:wp-eval exp-conj-def pconj-def)
  fix s::'s
  have « G » s ∗ wlp body (u P) s + (1 − « G » s) ∗ P s +
       (« G » s ∗ wp body (t Q) s + (1 − « G » s) ∗ Q s) ⊖ 1 =
       (« G » s ∗ wlp body (u P) s + « G » s ∗ wp body (t Q) s) +
        ((1 − « G » s) ∗ P s + (1 − « G » s) ∗ Q s) ⊖ («G» s + (1 − «G» s))
    by(simp add:ac-simps)
  also have ... ≤
       (« G » s ∗ wlp body (u P) s + « G » s ∗ wp body (t Q) s ⊖ «G» s) +
        ((1 − « G » s) ∗ P s + (1 − « G » s) ∗ Q s ⊖ (1 − «G» s))
    by(rule tminus-add-mono)
  also have ... =
       « G » s ∗ (wlp body (u P) s + wp body (t Q) s ⊖ 1) +
        (1 − « G » s) ∗ (P s + Q s ⊖ 1)
     by(simp add:tminus-left-distrib distrib-left)
  also {
    from uP uQ ft uu
    have wlp body (u P) && wp body (t Q) ⊩ wp body (u P && t Q)
      by(auto intro:sub-distrib-pconjD[OF sdp-body])
    also from IH unitary-sound[OF uP] unitary-sound[OF uQ] ft
              unitary-sound[OF uu[OF uP]]
    have . . .  ≤ wp body (t (P && Q))
      by(blast intro!:mono-transD[OF healthy-monoD, OF hwp] exp-conj-sound)
    finally have wlp body (u P) s + wp body (t Q) s ⊖ 1 ≤
               wp body (t (λs. P s + Q s ⊖ 1)) s
      by(auto simp:exp-conj-def pconj-def)
    hence « G » s ∗ (wlp body (u P) s + wp body (t Q) s ⊖ 1) +
         (1 − « G » s) ∗ (P s + Q s ⊖ 1) ≤
         « G » s ∗ wp body (t (λs. P s + Q s ⊖ 1)) s +
         (1 − « G » s) ∗ (P s + Q s ⊖ 1)
      by(auto intro:add-right-mono mult-left-mono)
  }
  finally
  show « G » s ∗ wlp body (u P) s + (1 − « G » s) ∗ P s +
```

$(« G » s * wp\ body\ (t\ Q)\ s + (1 - « G » s) * Q\ s) \ominus 1 \leq$
$« G » s * wp\ body\ (t\ (\lambda s.\ P\ s + Q\ s \ominus 1))\ s +$
$(1 - « G » s) * (P\ s + Q\ s \ominus 1)$ **.**
qed
next
fix *P Q*::*'s expect* **and** *t t' u u'*::*'s trans*
assume *unitary P unitary Q*
equiv-trans t t' equiv-utrans u u'
$u\ P\ \&\&\ t\ Q \Vdash t\ (P\ \&\&\ Q)$
thus $u'\ P\ \&\&\ t'\ Q \Vdash t'\ (P\ \&\&\ Q)$
by(*simp add*:*equiv-transD unitary-sound equiv-utransD exp-conj-unitary*)
qed

lemmas *sdp-intros* =
sdp-Abort sdp-Skip sdp-Apply
sdp-Seq sdp-DC sdp-PC
sdp-SetPC sdp-SetDC sdp-Embed
sdp-repeat sdp-Bind sdp-loop

4.7.3 The Well-Defined Predicate.

definition
well-def :: *'s prog* ⇒ *bool*
where
well-def prog ≡ *healthy* (*wp prog*) ∧ *nearly-healthy* (*wlp prog*)
∧ *wp-under-wlp prog* ∧ *sub-distrib-pconj prog*
∧ *sublinear* (*wp prog*) ∧ *bd-cts* (*wp prog*)

lemma *well-defI*[*intro*]:
⟦ *healthy* (*wp prog*); *nearly-healthy* (*wlp prog*);
wp-under-wlp prog; *sub-distrib-pconj prog*; *sublinear* (*wp prog*);
bd-cts (*wp prog*) ⟧ ⟹
well-def prog
unfolding *well-def-def* **by**(*simp*)

lemma *well-def-wp-healthy*[*dest*]:
well-def prog ⟹ *healthy* (*wp prog*)
unfolding *well-def-def* **by**(*simp*)

lemma *well-def-wlp-nearly-healthy*[*dest*]:
well-def prog ⟹ *nearly-healthy* (*wlp prog*)
unfolding *well-def-def* **by**(*simp*)

lemma *well-def-wp-under*[*dest*]:
well-def prog ⟹ *wp-under-wlp prog*
unfolding *well-def-def* **by**(*simp*)

lemma *well-def-sdp*[*dest*]:
well-def prog ⟹ *sub-distrib-pconj prog*

unfolding *well-def-def* **by**(*simp*)

lemma *well-def-wp-sublinear*[*dest*]:
well-def prog $\Longrightarrow$ *sublinear* (*wp prog*)
unfolding *well-def-def* **by**(*simp*)

lemma *well-def-wp-cts*[*dest*]:
well-def prog $\Longrightarrow$ *bd-cts* (*wp prog*)
unfolding *well-def-def* **by**(*simp*)

lemmas *wd-dests* =
well-def-wp-healthy well-def-wlp-nearly-healthy
well-def-wp-under well-def-sdp
well-def-wp-sublinear well-def-wp-cts

lemma *wd-Abort*:
well-def Abort
by(*blast intro*:*healthy-wp-Abort nearly-healthy-wlp-Abort*
wp-under-wlp-Abort sdp-Abort sublinear-wp-Abort
cts-wp-Abort)

lemma *wd-Skip*:
well-def Skip
by(*blast intro*:*healthy-wp-Skip nearly-healthy-wlp-Skip*
wp-under-wlp-Skip sdp-Skip sublinear-wp-Skip
cts-wp-Skip)

lemma *wd-Apply*:
well-def (*Apply f*)
by(*blast intro*:*healthy-wp-Apply nearly-healthy-wlp-Apply*
wp-under-wlp-Apply sdp-Apply sublinear-wp-Apply
cts-wp-Apply)

lemma *wd-Seq*:
$[\![$ *well-def a*; *well-def b* $]\!] \Longrightarrow$ *well-def* (*a* ;; *b*)
by(*blast intro*:*healthy-wp-Seq nearly-healthy-wlp-Seq*
wp-under-wlp-Seq sdp-Seq sublinear-wp-Seq
cts-wp-Seq)

lemma *wd-PC*:
$[\![$ *well-def a*; *well-def b*; *unitary P* $]\!] \Longrightarrow$ *well-def* ($a\ {}_P\!\oplus\ b$)
by(*blast intro*:*healthy-wp-PC nearly-healthy-wlp-PC*
wp-under-wlp-PC sdp-PC sublinear-wp-PC
cts-wp-PC)

lemma *wd-DC*:
$[\![$ *well-def a*; *well-def b* $]\!] \Longrightarrow$ *well-def* ($a \bigsqcap b$)
by(*blast intro*:*healthy-wp-DC nearly-healthy-wlp-DC*
wp-under-wlp-DC sdp-DC sublinear-wp-DC

cts-wp-DC)

lemma *wd-SetDC*:
⟦ ⋀*x s*. *x* ∈ *S s* ⟹ *well-def* (*a x*); ⋀*s*. *S s* ≠ {};
⋀*s*. *finite* (*S s*) ⟧ ⟹ *well-def* (*SetDC a S*)
by (*simp add*: *cts-wp-SetDC ex-in-conv healthy-intros*(*17*) *healthy-intros*(*18*) *sdp-intros*(*8*) *sublinear-intros*(*8*) *well-def-def wp-under-wlp-intros*(*8*))

lemma *wd-SetPC*:
⟦ ⋀*x s*. *x* ∈ (*supp* (*p s*)) ⟹ *well-def* (*a x*); ⋀*s*. *unitary* (*p s*); ⋀*s*. *finite* (*supp* (*p s*));
⋀*s*. *sum* (*p s*) (*supp* (*p s*)) ≤ *1* ⟧ ⟹ *well-def* (*SetPC a p*)
by(*iprover intro*!:*well-defI healthy-wp-SetPC nearly-healthy-wlp-SetPC*
wp-under-wlp-SetPC sdp-SetPC sublinear-wp-SetPC cts-wp-SetPC
dest:*wd-dests unitary-sound sound-nneg*[*OF unitary-sound*] *nnegD*)

lemma *wd-Embed*:
fixes *t*::*'s trans*
assumes *ht*: *healthy t* **and** *st*: *sublinear t* **and** *ct*: *bd-cts t*
shows *well-def* (*Embed t*)
proof(*intro well-defI*)
from *ht* **show** *healthy* (*wp* (*Embed t*)) *nearly-healthy* (*wlp* (*Embed t*))
by(*simp add*:*wp-def wlp-def Embed-def healthy-nearly-healthy*)+
from *st* **show** *sublinear* (*wp* (*Embed t*)) **by**(*simp add*:*wp-def Embed-def*)
show *wp-under-wlp* (*Embed t*) **by**(*simp add*:*wp-under-wlp-def wp-eval*)
show *sub-distrib-pconj* (*Embed t*)
by(*rule sub-distrib-pconjI*,
auto intro:*le-funI*[*OF sublinearD*[*OF st*, **where** *a=1* **and** *b=1* **and** *c=1*, *simplified*]]
simp:*exp-conj-def pconj-def wp-def wlp-def Embed-def*)
from *ct* **show** *bd-cts* (*wp* (*Embed t*))
by(*simp add*:*wp-def Embed-def*)
qed

lemma *wd-repeat*:
well-def a ⟹ *well-def* (*repeat n a*)
by(*blast intro*:*healthy-wp-repeat nearly-healthy-wlp-repeat*
wp-under-wlp-repeat sdp-repeat sublinear-wp-repeat cts-wp-repeat)

lemma *wd-Bind*:
⟦ ⋀*s*. *well-def* (*a* (*f s*)) ⟧ ⟹ *well-def* (*Bind f a*)
by(*blast intro*:*healthy-wp-Bind nearly-healthy-wlp-Bind*
wp-under-wlp-Bind sdp-Bind sublinear-wp-Bind cts-wp-Bind)

lemma *wd-loop*:
well-def body ⟹ *well-def* (*do G* ⟶ *body od*)
by(*blast intro*:*healthy-wp-loop nearly-healthy-wlp-loop*
wp-under-wlp-loop sdp-loop sublinear-wp-loop cts-wp-loop)

lemmas *wd-intros* =

wd-Abort wd-Skip wd-Apply
wd-Embed wd-Seq wd-PC
wd-DC wd-SetPC wd-SetDC
wd-Bind wd-repeat wd-loop

end

4.8 The Loop Rules

theory *Loops* **imports** *WellDefined* **begin**

Given a well-defined body, we can annotate a loop using an invariant, just as in the classical setting.

4.8.1 Liberal and Strict Invariants.

A probabilistic invariant generalises a boolean one: it *entails* itself, given the loop guard.

definition
wp-inv :: $('s \Rightarrow bool) \Rightarrow {}'s\ prog \Rightarrow ('s \Rightarrow real) \Rightarrow bool$
where
wp-inv G body I $\longleftrightarrow (\forall s.$ «*G*» $s * I\ s \leq wp\ body\ I\ s)$

lemma *wp-invI*:
$\bigwedge I. (\bigwedge s.$ «*G*» $s * I\ s \leq wp\ body\ I\ s) \Longrightarrow wp\text{-}inv\ G\ body\ I$
by(*simp add*:*wp-inv-def*)

definition
wlp-inv :: $('s \Rightarrow bool) \Rightarrow {}'s\ prog \Rightarrow ('s \Rightarrow real) \Rightarrow bool$
where
wlp-inv G body I $\longleftrightarrow (\forall s.$ «*G*» $s * I\ s \leq wlp\ body\ I\ s)$

lemma *wlp-invI*:
$\bigwedge I. (\bigwedge s.$ «*G*» $s * I\ s \leq wlp\ body\ I\ s) \Longrightarrow wlp\text{-}inv\ G\ body\ I$
by(*simp add*:*wlp-inv-def*)

lemma *wlp-invD*:
wlp-inv G body I $\Longrightarrow$ «*G*» $s * I\ s \leq wlp\ body\ I\ s$
by(*simp add*:*wlp-inv-def*)

For standard invariants, the multiplication reduces to conjunction.

lemma *wp-inv-stdD*:
assumes *inv*: *wp-inv G body* «*I*»
and *hb*: *healthy* (*wp body*)
shows «*G*» && «*I*» $\Vdash$ *wp body* «*I*»
proof(*rule le-funI*)
fix *s*

```
 show («G» && «I») s ≤ wp body «I» s
 proof(cases G s)
  case False
  with hb show ?thesis
   by(auto simp:exp-conj-def)
 next
  case True
  hence («G» && «I») s = «G» s ∗ «I» s
   by(simp add:exp-conj-def)
  also from inv have «G» s ∗ «I» s ≤ wp body «I» s
   by(simp add:wp-inv-def)
  finally show ?thesis .
 qed
qed
```

4.8.2 Partial Correctness

Partial correctness for loops[McIver and Morgan, 2004, Lemma 7.2.2, §7, p. 185].

```
lemma wlp-Loop:
 assumes wd: well-def body
   and uI: unitary I
   and inv: wlp-inv G body I
 shows I ≤ wlp do G ⟶ body od (λs. «𝒩 G» s ∗ I s)
 (is I ≤ wlp do G ⟶ body od ?P)
proof −
 let ?f Q s = «G» s ∗ wlp body Q s + «𝒩 G» s ∗ ?P s
 have I ⊩ gfp-exp ?f
 proof(rule gfp-exp-upperbound[OF - uI])
  have I = (λs. («G» s + «𝒩 G» s) ∗ I s) by(simp add:negate-embed)
  also have ... = (λs. «G» s ∗ I s + «𝒩 G» s ∗ I s)
   by(simp add:algebra-simps)
  also have ... = (λs. «G» s ∗ («G» s ∗ I s) + «𝒩 G» s ∗ («𝒩 G» s ∗ I s))
   by(simp add:embed-bool-idem algebra-simps)
  also have ... ⊩ (λs. «G» s ∗ wlp body I s + «𝒩 G» s ∗ («𝒩 G» s ∗ I s))
   using inv by(auto dest:wlp-invD intro:add-mono mult-left-mono)
  finally show I ⊩ (λs. «G» s ∗ wlp body I s + «𝒩 G» s ∗ («𝒩 G» s ∗ I s)) .
 qed
 also from uI well-def-wlp-nearly-healthy[OF wd] have ... = wlp do G ⟶ body od ?P
  by(auto intro!:wlp-Loop1[symmetric] unitary-intros)
 finally show ?thesis .
qed
```

4.8.3 Total Correctness

The first total correctness lemma for loops which terminate with probability 1[McIver and Morgan, 2004, Lemma 7.3.1, §7, p. 186].

```
lemma wp-Loop:
 assumes wd:  well-def body
```

```
    and inv: wlp-inv G body I
    and unit: unitary I
  shows I && wp (do G ⟶ body od) (λs. 1) ⊩ wp (do G ⟶ body od) (λs. «𝒩 G» s ∗ I s)
   (is I && ?T ⊩ wp ?loop ?X)
proof –
```

We first appeal to the *liberal* loop rule:

```
  from assms have I && ?T ⊩ wlp ?loop ?X && ?T
    by(blast intro:exp-conj-mono-left wlp-Loop)
```

Next, by sub-conjunctivity:

```
  also {
    from wd have sdp-loop: sub-distrib-pconj (do G ⟶ body od)
      by(blast intro:sdp-intros)

    from wd unit have wlp ?loop ?X && ?T ⊩ wp ?loop (?X && (λs. 1))
      by(blast intro:sub-distrib-pconjD sdp-intros unitary-intros)
  }
```

Finally, the conjunction collapses:

```
  finally show ?thesis
    by(simp add:exp-conj-1-right sound-intros sound-nneg unit unitary-sound)
qed
```

4.8.4 Unfolding

```
lemma wp-loop-unfold:
  fixes body :: 's prog
  assumes sP: sound P
    and h: healthy (wp body)
  shows wp (do G ⟶ body od) P =
   (λs. «𝒩 G» s ∗ P s + «G» s ∗ wp body (wp (do G ⟶ body od) P) s)
proof (simp only: wp-eval)
  let ?X t = wp (body ;; Embed t « G »⊕ Skip)
  have equiv-trans (lfp-trans ?X)
   (wp (body ;; Embed (lfp-trans ?X) « G »⊕ Skip))
  proof(intro lfp-trans-unfold)
    fix t::'s trans and P::'s expect
    assume st: ⋀Q. sound Q ⟹ sound (t Q)
      and sP: sound P
    with h show sound (?X t P)
      by(rule wp-loop-step-sound)
  next
    fix t u::'s trans
    assume le-trans t u (⋀P. sound P ⟹ sound (t P))
        (⋀P. sound P ⟹ sound (u P))
    with h show le-trans (wp (body ;; Embed t « G »⊕ Skip))
                  (wp (body ;; Embed u « G »⊕ Skip))
```

```
    by(iprover intro:wp-loop-step-mono)
  next
    let ?v = λP s. bound-of P
    from h show le-trans (wp (body ;; Embed ?v «G»⊕ Skip)) ?v
      by(intro le-transI, simp add:wp-eval lfp-loop-fp[unfolded negate-embed])
    fix P::'s expect
    assume sound P thus sound (?v P) by(auto)
  qed
  also have equiv-trans ...
    (λP s. «𝒩 G» s ∗ P s + «G» s ∗ wp body (wp (Embed (lfp-trans ?X)) P) s)
    by(rule equiv-transI, simp add:wp-eval algebra-simps negate-embed)
  finally show lfp-trans ?X P =
    (λs. «𝒩 G» s ∗ P s + «G» s ∗ wp body (lfp-trans ?X P) s)
    using sP unfolding wp-eval by(blast)
qed

lemma wp-loop-nguard:
  ⟦ healthy (wp body); sound P; ¬ G s ⟧ ⟹ wp do G ⟶ body od P s = P s
  by(subst wp-loop-unfold, simp-all)

lemma wp-loop-guard:
  ⟦ healthy (wp body); sound P; G s ⟧ ⟹
   wp do G ⟶ body od P s = wp (body ;; do G ⟶ body od) P s
  by(subst wp-loop-unfold, simp-all add:wp-eval)

end
```

4.9 The Algebra of pGCL

theory *Algebra* **imports** *WellDefined* **begin**

Programs in pGCL have a rich algebraic structure, largely mirroring that for GCL. We show that programs form a lattice under refinement, with $a \bigsqcap b$ and $a \bigsqcup b$ as the meet and join operators, respectively. We also take advantage of the algebraic structure to establish a framwork for the modular decomposition of proofs.

4.9.1 Program Refinement

Refinement in pGCL relates to refinement in GCL exactly as probabilistic entailment relates to implication. It turns out to have a very similar algebra, the rules of which we establish shortly.

```
definition
  refines :: 's prog ⇒ 's prog ⇒ bool (infix ‹⊑› 70)
where
  prog ⊑ prog' ≡ ∀P. sound P ⟶ wp prog P ⊩ wp prog' P

lemma refinesI[intro]:
```

$[\![\bigwedge P.\ sound\ P \Longrightarrow wp\ prog\ P \Vdash wp\ prog'\ P]\!] \Longrightarrow prog \sqsubseteq prog'$
unfolding *refines-def* **by**(*simp*)

lemma *refinesD*[*dest*]:
$[\![prog \sqsubseteq prog';\ sound\ P]\!] \Longrightarrow wp\ prog\ P \Vdash wp\ prog'\ P$
unfolding *refines-def* **by**(*simp*)

The equivalence relation below will turn out to be that induced by refinement. It is also the application of *equiv-trans* to the weakest precondition.

definition
pequiv :: *'s prog ⇒ 's prog ⇒ bool* (**infix** ‹≃› *70*)
where
$prog \simeq prog' \equiv \forall P.\ sound\ P \longrightarrow wp\ prog\ P = wp\ prog'\ P$

lemma *pequivI*[*intro*]:
$[\![\bigwedge P.\ sound\ P \Longrightarrow wp\ prog\ P = wp\ prog'\ P]\!] \Longrightarrow prog \simeq prog'$
unfolding *pequiv-def* **by**(*simp*)

lemma *pequivD*[*dest*,*simp*]:
$[\![prog \simeq prog';\ sound\ P]\!] \Longrightarrow wp\ prog\ P = wp\ prog'\ P$
unfolding *pequiv-def* **by**(*simp*)

lemma *pequiv-equiv-trans*:
$a \simeq b \longleftrightarrow equiv\text{-}trans\ (wp\ a)\ (wp\ b)$
by(*auto*)

4.9.2 Simple Identities

The following identities involve only the primitive operations as defined in Section 4.1.1, and refinement as defined above.

Laws following from the basic arithmetic of the operators seperately

lemma *DC-comm*[*ac-simps*]:
$a \bigsqcap b = b \bigsqcap a$
unfolding *DC-def* **by**(*simp add*:*ac-simps*)

lemma *DC-assoc*[*ac-simps*]:
$a \bigsqcap (b \bigsqcap c) = (a \bigsqcap b) \bigsqcap c$
unfolding *DC-def* **by**(*simp add*:*ac-simps*)

lemma *DC-idem*:
$a \bigsqcap a = a$
unfolding *DC-def* **by**(*simp*)

lemma *AC-comm*[*ac-simps*]:
$a \bigsqcup b = b \bigsqcup a$
unfolding *AC-def* **by**(*simp add*:*ac-simps*)

lemma *AC-assoc*[*ac-simps*]:
$a \bigsqcup (b \bigsqcup c) = (a \bigsqcup b) \bigsqcup c$
unfolding *AC-def* **by**(*simp add*:*ac-simps*)

lemma *AC-idem*:
$a \bigsqcup a = a$
unfolding *AC-def* **by**(*simp*)

lemma *PC-quasi-comm*:
$a\ {}_{p}{\oplus}\ b = b\ {}_{(\lambda s.\ 1 - p\ s)}{\oplus}\ a$
unfolding *PC-def* **by**(*simp add*:*algebra-simps*)

lemma *PC-idem*:
$a\ {}_{p}{\oplus}\ a = a$
unfolding *PC-def* **by**(*simp add*:*algebra-simps*)

lemma *Seq-assoc*[*ac-simps*]:
$A ;; (B ;; C) = A ;; B ;; C$
by(*simp add*:*Seq-def o-def*)

lemma *Abort-refines*[*intro*]:
well-def a $\Longrightarrow$ *Abort* $\sqsubseteq$ *a*
by(*rule refinesI*, *unfold wp-eval*, *auto dest*!:*well-def-wp-healthy*)

Laws relating demonic choice and refinement

lemma *left-refines-DC*:
$(a \bigsqcap b) \sqsubseteq a$
by(*auto intro*!:*refinesI simp*:*wp-eval*)

lemma *right-refines-DC*:
$(a \bigsqcap b) \sqsubseteq b$
by(*auto intro*!:*refinesI simp*:*wp-eval*)

lemma *DC-refines*:
fixes *a*::*'s prog* **and** *b* **and** *c*
assumes *rab*: $a \sqsubseteq b$ **and** *rac*: $a \sqsubseteq c$
shows $a \sqsubseteq (b \bigsqcap c)$
proof
fix *P*::*'s* $\Rightarrow$ *real* **assume** *sP*: *sound P*
with *assms* **have** *wp a P* $\Vdash$ *wp b P* **and** *wp a P* $\Vdash$ *wp c P*
by(*auto dest*:*refinesD*)
thus *wp a P* $\Vdash$ *wp* $(b \bigsqcap c)$ *P*
by(*auto simp*:*wp-eval intro*:*min.boundedI*)
qed

lemma *DC-mono*:
fixes *a*::*'s prog*

assumes *rab*: $a \sqsubseteq b$ **and** *rcd*: $c \sqsubseteq d$
shows $(a \sqcap c) \sqsubseteq (b \sqcap d)$
proof(*rule refinesI*, *unfold wp-eval*, *rule le-funI*)
fix $P::'s \Rightarrow real$ **and** $s::'s$
assume *sP*: *sound P*
with *assms* **have** $wp\ a\ P\ s \leq wp\ b\ P\ s$ **and** $wp\ c\ P\ s \leq wp\ d\ P\ s$
by(*auto*)
thus $min\ (wp\ a\ P\ s)\ (wp\ c\ P\ s) \leq min\ (wp\ b\ P\ s)\ (wp\ d\ P\ s)$
by(*auto*)
qed

Laws relating angelic choice and refinement

lemma *left-refines-AC*:
$a \sqsubseteq (a \sqcup b)$
by(*auto intro*!:*refinesI simp*:*wp-eval*)

lemma *right-refines-AC*:
$b \sqsubseteq (a \sqcup b)$
by(*auto intro*!:*refinesI simp*:*wp-eval*)

lemma *AC-refines*:
fixes $a::'s\ prog$ **and** b **and** c
assumes *rac*: $a \sqsubseteq c$ **and** *rbc*: $b \sqsubseteq c$
shows $(a \sqcup b) \sqsubseteq c$
proof
fix $P::'s \Rightarrow real$ **assume** *sP*: *sound P*
with *assms* **have** $\bigwedge s.\ wp\ a\ P\ s \leq wp\ c\ P\ s$
and $\bigwedge s.\ wp\ b\ P\ s \leq wp\ c\ P\ s$
by(*auto dest*:*refinesD*)
thus $wp\ (a \sqcup b)\ P \Vdash wp\ c\ P$
unfolding *wp-eval* **by**(*auto*)
qed

lemma *AC-mono*:
fixes $a::'s\ prog$
assumes *rab*: $a \sqsubseteq b$ **and** *rcd*: $c \sqsubseteq d$
shows $(a \sqcup c) \sqsubseteq (b \sqcup d)$
proof(*rule refinesI*, *unfold wp-eval*, *rule le-funI*)
fix $P::'s \Rightarrow real$ **and** $s::'s$
assume *sP*: *sound P*
with *assms* **have** $wp\ a\ P\ s \leq wp\ b\ P\ s$ **and** $wp\ c\ P\ s \leq wp\ d\ P\ s$
by(*auto*)
thus $max\ (wp\ a\ P\ s)\ (wp\ c\ P\ s) \leq max\ (wp\ b\ P\ s)\ (wp\ d\ P\ s)$
by(*auto*)
qed

Laws depending on the arithmetic of $a\ {}_p\!\oplus\ b$ and $a \sqcap b$ together

lemma *DC-refines-PC*:

```
  assumes unit: unitary p
  shows (a ⊓ b) ⊑ (a p⊕ b)
proof(rule refinesI, unfold wp-eval, rule le-funI)
  fix s and P::'a ⇒ real assume sound: sound P
  from unit have nn-p: 0 ≤ p s by(blast)
  from unit have p s ≤ 1 by(blast)
  hence nn-np: 0 ≤ 1 − p s by(simp)
  show min (wp a P s) (wp b P s) ≤ p s ∗ wp a P s + (1 − p s) ∗ wp b P s
  proof(cases wp a P s ≤ wp b P s,
      simp-all add:min.absorb1 min.absorb2)
    case True note le = this
    have wp a P s = (p s + (1 − p s)) ∗ wp a P s by(simp)
    also have ... = p s ∗ wp a P s + (1 − p s) ∗ wp a P s
      by(simp only: distrib-right)
    also {
      from le and nn-np have (1 − p s) ∗ wp a P s ≤ (1 − p s) ∗ wp b P s
        by(rule mult-left-mono)
      hence p s ∗ wp a P s + (1 − p s) ∗ wp a P s ≤
        p s ∗ wp a P s + (1 − p s) ∗ wp b P s
        by(rule add-left-mono)
    }
    finally show wp a P s ≤ p s ∗ wp a P s + (1 − p s) ∗ wp b P s .
  next
    case False
    then have le: wp b P s ≤ wp a P s by(simp)
    have wp b P s = (p s + (1 − p s)) ∗ wp b P s by(simp)
    also have ... = p s ∗ wp b P s + (1 − p s) ∗ wp b P s
      by(simp only:distrib-right)
    also {
      from le and nn-p have p s ∗ wp b P s ≤ p s ∗ wp a P s
        by(rule mult-left-mono)
      hence p s ∗ wp b P s + (1 − p s) ∗ wp b P s ≤
        p s ∗ wp a P s + (1 − p s) ∗ wp b P s
        by(rule add-right-mono)
    }
    finally show wp b P s ≤ p s ∗ wp a P s + (1 − p s) ∗ wp b P s .
  qed
qed
```

Laws depending on the arithmetic of $a\ {}_p\!\oplus\ b$ and $a \bigsqcup b$ together

```
lemma PC-refines-AC:
  assumes unit: unitary p
  shows (a p⊕ b) ⊑ (a ⊔ b)
proof(rule refinesI, unfold wp-eval, rule le-funI)
  fix s and P::'a ⇒ real assume sound: sound P

  from unit have p s ≤ 1 by(blast)
  hence nn-np: 0 ≤ 1 − p s by(simp)
```

```
 show p s * wp a P s + (1 − p s) * wp b P s ≤
     max (wp a P s) (wp b P s)
 proof(cases wp a P s ≤ wp b P s)
  case True note leab = this
  with unit nn-np
  have p s * wp a P s + (1 − p s) * wp b P s ≤
      p s * wp b P s + (1 − p s) * wp b P s
   by(auto intro:add-mono mult-left-mono)
  also have ... = wp b P s
   by(auto simp:field-simps)
  also from leab
  have ... = max (wp a P s) (wp b P s)
   by(auto)
  finally show ?thesis .
 next
  case False note leba = this
  with unit nn-np
  have p s * wp a P s + (1 − p s) * wp b P s ≤
      p s * wp a P s + (1 − p s) * wp a P s
   by(auto intro:add-mono mult-left-mono)
  also have ... = wp a P s
   by(auto simp:field-simps)
  also from leba
  have ... = max (wp a P s) (wp b P s)
   by(auto)
  finally show ?thesis .
 qed
qed
```

Laws depending on the arithmetic of $a \bigsqcup b$ and $a \bigsqcap b$ together

```
lemma DC-refines-AC:
 (a ⨅ b) ⊑ (a ⨆ b)
 by(auto intro!:refinesI simp:wp-eval)
```

Laws Involving Refinement and Equivalence

```
lemma pr-trans[trans]:
 fixes A::'a prog
 assumes prAB: A ⊑ B
   and prBC: B ⊑ C
 shows A ⊑ C
proof
 fix P::'a ⇒ real assume sP: sound P
 with prAB have wp A P ⊩ wp B P by(blast)
 also from sP and prBC have ... ⊩ wp C P by(blast)
 finally show wp A P ⊩ ... .
qed
```

lemma *pequiv-refl*[*intro*!,*simp*]:
$a \simeq a$
by(*auto*)

lemma *pequiv-comm*[*ac-simps*]:
$a \simeq b \longleftrightarrow b \simeq a$
unfolding *pequiv-def*
by(*rule iffI*, *safe*, *simp-all*)

lemma *pequiv-pr*[*dest*]:
$a \simeq b \Longrightarrow a \sqsubseteq b$
by(*auto*)

lemma *pequiv-trans*[*intro*,*trans*]:
$[\![a \simeq b; b \simeq c]\!] \Longrightarrow a \simeq c$
unfolding *pequiv-def* **by**(*auto intro*!:*order-trans*)

lemma *pequiv-pr-trans*[*intro*,*trans*]:
$[\![a \simeq b; b \sqsubseteq c]\!] \Longrightarrow a \sqsubseteq c$
unfolding *pequiv-def refines-def* **by**(*simp*)

lemma *pr-pequiv-trans*[*intro*,*trans*]:
$[\![a \sqsubseteq b; b \simeq c]\!] \Longrightarrow a \sqsubseteq c$
unfolding *pequiv-def refines-def* **by**(*simp*)

Refinement induces equivalence by antisymmetry:

lemma *pequiv-antisym*:
$[\![a \sqsubseteq b; b \sqsubseteq a]\!] \Longrightarrow a \simeq b$
by(*auto intro*:*antisym*)

lemma *pequiv-DC*:
$[\![a \simeq c; b \simeq d]\!] \Longrightarrow (a \bigsqcap b) \simeq (c \bigsqcap d)$
by(*auto intro*!:*DC-mono pequiv-antisym simp*:*ac-simps*)

lemma *pequiv-AC*:
$[\![a \simeq c; b \simeq d]\!] \Longrightarrow (a \bigsqcup b) \simeq (c \bigsqcup d)$
by(*auto intro*!:*AC-mono pequiv-antisym simp*:*ac-simps*)

4.9.3 Deterministic Programs are Maximal

Any sub-additive refinement of a deterministic program is in fact an equivalence. Deterministic programs are thus maximal (under the refinement order) among sub-additive programs.

lemma *refines-determ*:
fixes *a*::*'s prog*
assumes *da*: *determ* (*wp a*)
and *wa*: *well-def a*
and *wb*: *well-def b*

and *dr*: $a \sqsubseteq b$
shows $a \simeq b$

Proof by contradiction.

proof(*rule pequivI*, *rule contrapos-pp*)
from *wb* **have** *feasible* (*wp b*) **by**(*auto*)
with *wb* **have** *sab*: *sub-add* (*wp b*)
by(*auto dest*: *sublinear-subadd*[*OF well-def-wp-sublinear*])
fix *P*::*'s* $\Rightarrow$ *real* **assume** *sP*: *sound P*

Assume that *a* and *b* are not equivalent:

assume *ne*: $wp\ a\ P \neq wp\ b\ P$

Find a point at which they differ. As $a \sqsubseteq b$, *wp b P s* must by strictly greater than *wp a P s* here:

hence $\exists s.\ wp\ a\ P\ s < wp\ b\ P\ s$
proof(*rule contrapos-np*)
assume $\neg(\exists s.\ wp\ a\ P\ s < wp\ b\ P\ s)$
hence $\forall s.\ wp\ b\ P\ s \leq wp\ a\ P\ s$ **by**(*auto simp*:*not-less*)
hence $wp\ b\ P \Vdash wp\ a\ P$ **by**(*auto*)
moreover from *sP dr* **have** $wp\ a\ P \Vdash wp\ b\ P$ **by**(*auto*)
ultimately show $wp\ a\ P = wp\ b\ P$ **by**(*auto*)
qed
then obtain *s* **where** *less*: $wp\ a\ P\ s < wp\ b\ P\ s$ **by**(*blast*)

Take a carefully constructed expectation:

let $?Pc = \lambda s.\ bound\text{-}of\ P - P\ s$
have *sPc*: *sound ?Pc*
proof(*rule soundI*)
from *sP* **have** $\bigwedge s.\ 0 \leq P\ s$ **by**(*auto*)
hence $\bigwedge s.\ ?Pc\ s \leq bound\text{-}of\ P$ **by**(*auto*)
thus *bounded ?Pc* **by**(*blast*)
from *sP* **have** $\bigwedge s.\ P\ s \leq bound\text{-}of\ P$ **by**(*auto*)
hence $\bigwedge s.\ 0 \leq\ ?Pc\ s$
by *auto*
thus *nneg ?Pc* **by**(*auto*)
qed

We then show that *wp b* violates feasibility, and thus healthiness.

from *sP* **have** $0 \leq bound\text{-}of\ P$ **by**(*auto*)
with *da* **have** $bound\text{-}of\ P = wp\ a\ (\lambda s.\ bound\text{-}of\ P)\ s$
by(*simp add*:*maximalD determ-maximalD*)
also have $\ldots = wp\ a\ (\lambda s.\ ?Pc\ s + P\ s)\ s$
by(*simp*)
also from *da sP sPc* **have** $\ldots = wp\ a\ ?Pc\ s + wp\ a\ P\ s$
by(*subst additiveD*[*OF determ-additiveD*], *simp-all add*:*sP sPc*)
also from *sPc dr* **have** $\ldots \leq wp\ b\ ?Pc\ s + wp\ a\ P\ s$
by(*auto*)
also from *less* **have** $\ldots < wp\ b\ ?Pc\ s + wp\ b\ P\ s$

```
    by(auto)
  also from sab sP sPc have ... ≤ wp b (λs. ?Pc s + P s) s
    by(blast)
  finally have ¬wp b (λs. bound-of P) s ≤ bound-of P
    by(simp)
  thus ¬bounded-by (bound-of P) (wp b (λs. bound-of P))
    by(auto)
next
```

However,

```
  fix P::'s ⇒ real assume sP: sound P
  hence nneg (λs. bound-of P) by(auto)
  moreover have bounded-by (bound-of P) (λs. bound-of P) by(auto)
  ultimately
  show bounded-by (bound-of P) (wp b (λs. bound-of P))
    using wb by(auto dest!:well-def-wp-healthy)
qed
```

4.9.4 The Algebraic Structure of Refinement

Well-defined programs form a half-bounded semilattice under refinement, where *Abort* is bottom, and $a \bigsqcap b$ is *inf*. There is no unique top element, but all fully-deterministic programs are maximal.

The type that we construct here is not especially useful, but serves as a convenient way to express this result.

```
quotient-type 's program =
  's prog / partial : λa b. a ≃ b ∧ well-def a ∧ well-def b
proof(rule part-equivpI)
  have Skip ≃ Skip and well-def Skip by(auto intro:wd-intros)
  thus ∃x. x ≃ x ∧ well-def x ∧ well-def x by(blast)
  show symp (λa b. a ≃ b ∧ well-def a ∧ well-def b)
  proof(rule sympI, safe)
    fix a::'a prog and b
    assume a ≃ b
    hence equiv-trans (wp a) (wp b)
      by(simp add:pequiv-equiv-trans)
    thus b ≃ a by(simp add:ac-simps pequiv-equiv-trans)
  qed
  show transp (λa b. a ≃ b ∧ well-def a ∧ well-def b)
    by(rule transpI, safe, rule pequiv-trans)
qed

instantiation program :: (type) semilattice-inf begin
lift-definition
  less-eq-program :: 'a program ⇒ 'a program ⇒ bool is refines
proof(safe)
  fix a::'a prog and b c d
  assume a ≃ b hence b ≃ a by(simp add:ac-simps)
```

also assume $a \sqsubseteq c$
also assume $c \simeq d$
finally show $b \sqsubseteq d$ **.**
next
fix a::$'a$ *prog* **and** $b\ c\ d$
assume $a \simeq b$
also assume $b \sqsubseteq d$
also assume $c \simeq d$ **hence** $d \simeq c$ **by**(*simp add*:*ac-simps*)
finally show $a \sqsubseteq c$ **.**
qed

lift-definition
less-program :: $'a$ *program* $\Rightarrow$ $'a$ *program* $\Rightarrow$ *bool*
is $\lambda a\ b.\ a \sqsubseteq b \wedge \neg\, b \sqsubseteq a$
proof(*safe*)
fix a::$'a$ *prog* **and** $b\ c\ d$
assume $a \simeq b$ **hence** $b \simeq a$ **by**(*simp add*:*ac-simps*)
also assume $a \sqsubseteq c$
also assume $c \simeq d$
finally show $b \sqsubseteq d$ **.**
next
fix a::$'a$ *prog* **and** $b\ c\ d$
assume $a \simeq b$
also assume $b \sqsubseteq d$
also assume $c \simeq d$ **hence** $d \simeq c$ **by**(*simp add*:*ac-simps*)
finally show $a \sqsubseteq c$ **.**
next
fix $a\ b$ **and** c::$'a$ *prog* **and** d
assume $c \simeq d$
also assume $d \sqsubseteq b$
also assume $a \simeq b$ **hence** $b \simeq a$ **by**(*simp add*:*ac-simps*)
finally have $c \sqsubseteq a$ **.**
moreover assume $\neg\, c \sqsubseteq a$
ultimately show *False* **by**(*auto*)
next
fix $a\ b$ **and** c::$'a$ *prog* **and** d
assume $c \simeq d$ **hence** $d \simeq c$ **by**(*simp add*:*ac-simps*)
also assume $c \sqsubseteq a$
also assume $a \simeq b$
finally have $d \sqsubseteq b$ **.**
moreover assume $\neg\, d \sqsubseteq b$
ultimately show *False* **by**(*auto*)
qed

lift-definition
inf-program :: $'a$ *program* $\Rightarrow$ $'a$ *program* $\Rightarrow$ $'a$ *program* **is** *DC*
proof(*safe*)
fix $a\ b\ c\ d$::$'s$ *prog*
assume $a \simeq b$ **and** $c \simeq d$

thus $(a \bigsqcap c) \simeq (b \bigsqcap d)$ **by**(*rule pequiv-DC*)
next
fix *a c*::*'s prog*
assume *well-def a well-def c*
thus *well-def* $(a \bigsqcap c)$ **by**(*rule wd-intros*)
next
fix *a c*::*'s prog*
assume *well-def a well-def c*
thus *well-def* $(a \bigsqcap c)$ **by**(*rule wd-intros*)
qed

instance
proof
fix *x y*::*'a program*
show $(x < y) = (x \leq y \wedge \neg\, y \leq x)$
by(*transfer*, *simp*)
show $x \leq x$
by(*transfer*, *auto*)
show *inf x y* $\leq x$
by(*transfer*, *rule left-refines-DC*)
show *inf x y* $\leq y$
by(*transfer*, *rule right-refines-DC*)
assume $x \leq y$ **and** $y \leq x$ **thus** $x = y$
by(*transfer*, *iprover intro*:*pequiv-antisym*)
next
fix *x y z*::*'a program*
assume $x \leq y$ **and** $y \leq z$
thus $x \leq z$
by(*transfer*, *iprover intro*:*pr-trans*)
next
fix *x y z*::*'a program*
assume $x \leq y$ **and** $x \leq z$
thus $x \leq$ *inf y z*
by(*transfer*, *iprover intro*:*DC-refines*)
qed
end

instantiation *program* :: (*type*) *bot* **begin**
lift-definition
bot-program :: *'a program* **is** *Abort*
by(*auto intro*:*wd-intros*)

instance ..
end

lemma *eq-det*: $\bigwedge a\ b$::*'s prog*. $[\![\ a \simeq b;\ determ\ (wp\ a)\ ]\!] \Longrightarrow determ\ (wp\ b)$
proof(*intro determI additiveI maximalI*)
fix *a b*::*'s prog* **and** *P*::*'s* $\Rightarrow$ *real*
and *Q*::*'s* $\Rightarrow$ *real* **and** *s*::*'s*

assume *da*: *determ* (*wp a*)
assume *sP*: *sound P* **and** *sQ*: *sound Q*
and *eq*: $a \simeq b$
hence $wp\ b\ (\lambda s.\ P\ s + Q\ s)\ s =$
$wp\ a\ (\lambda s.\ P\ s + Q\ s)\ s$
by(*simp add*:*sound-intros*)
also from *da sP sQ*
have $... = wp\ a\ P\ s + wp\ a\ Q\ s$
by(*simp add*:*additiveD determ-additiveD*)
also from *eq sP sQ*
have $... = wp\ b\ P\ s + wp\ b\ Q\ s$
by(*simp add*:*pequivD*)
finally show $wp\ b\ (\lambda s.\ P\ s + Q\ s)\ s = wp\ b\ P\ s + wp\ b\ Q\ s$ **.**
next
fix *a b*::*'s prog* **and** *c*::*real*
assume *da*: *determ* (*wp a*)
assume $a \simeq b$ **hence** $b \simeq a$ **by**(*simp add*:*ac-simps*)
moreover assume *nn*: $0 \leq c$
ultimately have $wp\ b\ (\lambda\text{-}.\ c) = wp\ a\ (\lambda\text{-}.\ c)$
by(*simp add*:*pequivD const-sound*)
also from *da nn* **have** $... = (\lambda\text{-}.\ c)$
by(*simp add*:*determ-maximalD maximalD*)
finally show $wp\ b\ (\lambda\text{-}.\ c) = (\lambda\text{-}.\ c)$ **.**
qed

lift-definition
pdeterm :: *'s program* $\Rightarrow$ *bool*
is $\lambda a.$ *determ* (*wp a*)
proof(*safe*)
fix *a b*::*'s prog*
assume $a \simeq b$ **and** *determ* (*wp a*)
thus *determ* (*wp b*) **by**(*rule eq-det*)
next
fix *a b*::*'s prog*
assume $a \simeq b$ **hence** $b \simeq a$ **by**(*simp add*:*ac-simps*)
moreover assume *determ* (*wp b*)
ultimately show *determ* (*wp a*) **by**(*rule eq-det*)
qed

lemma *determ-maximal*:
$[\![$ *pdeterm a*; $a \leq x$ $]\!] \Longrightarrow a = x$
by(*transfer*, *auto intro*:*refines-determ*)

4.9.5 Data Refinement

A projective data refinement construction for pGCL. By projective, we mean that the abstract state is always a function (φ) of the concrete state. Refinement may be predicated (G) on the state.

definition
drefines :: $('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'a\ prog \Rightarrow 'b\ prog \Rightarrow bool$
where
$drefines\ \varphi\ G\ A\ B \equiv \forall P\ Q.\ (unitary\ P \wedge unitary\ Q \wedge (P \Vdash wp\ A\ Q)) \longrightarrow$
$(«G» \&\&\ (P\ o\ \varphi) \Vdash wp\ B\ (Q\ o\ \varphi))$

lemma *drefinesD*[*dest*]:
$[\![\ drefines\ \varphi\ G\ A\ B;\ unitary\ P;\ unitary\ Q;\ P \Vdash wp\ A\ Q\ ]\!] \Longrightarrow$
$«G» \&\&\ (P\ o\ \varphi) \Vdash wp\ B\ (Q\ o\ \varphi)$
unfolding *drefines-def* **by**(*blast*)

We can alternatively use G as an assumption:

lemma *drefinesD2*:
assumes *dr*: $drefines\ \varphi\ G\ A\ B$
and *uP*: *unitary P*
and *uQ*: *unitary Q*
and *wpA*: $P \Vdash wp\ A\ Q$
and *G*: *G s*
shows $(P\ o\ \varphi)\ s \leq wp\ B\ (Q\ o\ \varphi)\ s$
proof −
from *uP* **have** $0 \leq (P\ o\ \varphi)\ s$ **unfolding** *o-def* **by**(*blast*)
with *G* **have** $(P\ o\ \varphi)\ s = («G» \&\&\ (P\ o\ \varphi))\ s$
by(*simp add*:*exp-conj-def*)
also from *assms* **have** $... \leq wp\ B\ (Q\ o\ \varphi)\ s$ **by**(*blast*)
finally show $(P\ o\ \varphi)\ s \leq ...$ **.**
qed

This additional form is sometimes useful:

lemma *drefinesD3*:
assumes *dr*: $drefines\ \varphi\ G\ a\ b$
and *G*: *G s*
and *uQ*: *unitary Q*
and *wa*: *well-def a*
shows $wp\ a\ Q\ (\varphi\ s) \leq wp\ b\ (Q\ o\ \varphi)\ s$
proof −
let $?L\ s' = wp\ a\ Q\ s'$
from *uQ wa* **have** *sL*: *sound ?L* **by**(*blast*)
from *uQ wa* **have** *bL*: *bounded-by 1 ?L* **by**(*blast*)

have $?L \Vdash\ ?L$ **by**(*simp*)
with *sL* **and** *bL* **and** *assms*
show *?thesis*
by(*blast intro*:*drefinesD2*[*OF dr*, **where** *P*=*?L*, *simplified*])
qed

lemma *drefinesI*[*intro*]:
$[\![\ \bigwedge P\ Q.\ [\![\ unitary\ P;\ unitary\ Q;\ P \Vdash wp\ A\ Q\ ]\!] \Longrightarrow$
$«G» \&\&\ (P\ o\ \varphi) \Vdash wp\ B\ (Q\ o\ \varphi)\ ]\!] \Longrightarrow$
$drefines\ \varphi\ G\ A\ B$

```
  unfolding drefines-def by(blast)
```

Use G as an assumption, when showing refinement:

```
lemma drefinesI2:
  fixes  A::'a prog
    and  B::'b prog
    and  φ::'b ⇒ 'a
    and  G::'b ⇒ bool
  assumes wB: well-def B
      and withAs:
        ⋀P Q s. ⟦ unitary P; unitary Q;
               G s; P ⊩ wp A Q ⟧ ⟹ (P o φ) s ≤ wp B (Q o φ) s
  shows drefines φ G A B
proof
  fix P and Q
  assume uP: unitary P
     and uQ: unitary Q
     and wpA: P ⊩ wp A Q

  hence ⋀s. G s ⟹ (P o φ) s ≤ wp B (Q o φ) s
    using withAs by(blast)
  moreover
  from uQ have unitary (Q o φ)
    unfolding o-def by(blast)
  moreover
  from uP have unitary (P o φ)
    unfolding o-def by(blast)
  ultimately
  show «G» && (P o φ) ⊩ wp B (Q o φ)
    using wB by(blast intro:entails-pconj-assumption)
qed

lemma dr-strengthen-guard:
  fixes a::'s prog and b::'t prog
  assumes fg: ⋀s. F s ⟹ G s
      and drab: drefines φ G a b
  shows drefines φ F a b
proof(intro drefinesI)
  fix P Q::'s expect
  assume uP: unitary P and uQ: unitary Q
     and wp: P ⊩ wp a Q
  from fg have ⋀s. «F» s ≤ «G» s by(simp add:embed-bool-def)
   hence («F» && (P o φ)) ⊩ («G» && (P o φ)) by(auto intro:pconj-mono le-funI simp:exp-conj-def)
  also from drab uP uQ wp have ... ⊩ wp b (Q o φ) by(auto)
  finally show «F» && (P o φ) ⊩ wp b (Q o φ) .
qed
```

Probabilistic correspondence, *pcorres*, is equality on distribution transformers, modulo a guard. It is the analogue, for data refinement, of program equivalence for program refinement.

```
definition
  pcorres :: ('b ⇒ 'a) ⇒ ('b ⇒ bool) ⇒ 'a prog ⇒ 'b prog ⇒ bool
where
  pcorres φ G A B ⟷
   (∀ Q. unitary Q ⟶ «G» && (wp A Q o φ) = «G» && wp B (Q o φ))

lemma pcorresI:
  ⟦ ⋀Q. unitary Q ⟹ «G» && (wp A Q o φ) = «G» && wp B (Q o φ) ⟧ ⟹
   pcorres φ G A B
  by(simp add:pcorres-def)
```

Often easier to use, as it allows one to assume the precondition.

```
lemma pcorresI2[intro]:
  fixes A::'a prog and B::'b prog
  assumes withG: ⋀Q s. ⟦ unitary Q; G s ⟧ ⟹ wp A Q (φ s)= wp B (Q o φ) s
     and wA: well-def A
     and wB: well-def B
  shows pcorres φ G A B
proof(rule pcorresI, rule ext)
  fix Q::'a ⇒ real and s::'b
  assume uQ: unitary Q
  hence uQφ: unitary (Q o φ) by(auto)
  show («G» && (wp A Q ∘ φ)) s = («G» && wp B (Q ∘ φ)) s
  proof(cases G s)
   case True note this
   moreover
   from well-def-wp-healthy[OF wA] uQ have 0 ≤ wp A Q (φ s) by(blast)
   moreover
   from well-def-wp-healthy[OF wB] uQφ have 0 ≤ wp B (Q o φ) s by(blast)
   ultimately show ?thesis
    using uQ by(simp add:exp-conj-def withG)
  next
   case False note this
   moreover
   from well-def-wp-healthy[OF wA] uQ have wp A Q (φ s) ≤ 1 by(blast)
   moreover
   from well-def-wp-healthy[OF wB] uQφ have wp B (Q o φ) s ≤ 1
    by(blast dest!:healthy-bounded-byD intro:sound-nneg)
   ultimately show ?thesis by(simp add:exp-conj-def)
  qed
qed

lemma pcorresD:
  ⟦ pcorres φ G A B; unitary Q ⟧ ⟹ «G» && (wp A Q o φ) = «G» && wp B (Q o φ)
  unfolding pcorres-def by(simp)
```

Again, easier to use if the precondition is known to hold.

lemma *pcorresD2*:
assumes *pc*: *pcorres* φ *G A B*
and *uQ*: *unitary Q*
and *wA*: *well-def A* **and** *wB*: *well-def B*
and *G*: *G s*
shows *wp A Q* (φ *s*) = *wp B* (*Q o* φ) *s*
proof −
from *uQ well-def-wp-healthy*[*OF wA*] **have** $0 \leq$ *wp A Q* (φ *s*) **by**(*auto*)
with *G* **have** *wp A Q* (φ *s*) = «*G*» *s* .& *wp A Q* (φ *s*) **by**(*simp*)
also {
from *pc uQ* **have** «*G*» && (*wp A Q o* φ) = «*G*» && *wp B* (*Q o* φ)
by(*rule pcorresD*)
hence «*G*» *s* .& *wp A Q* (φ *s*) = «*G*» *s* .& *wp B* (*Q o* φ) *s*
unfolding *exp-conj-def o-def* **by**(*rule fun-cong*)
}
also {
from *uQ* **have** *sound Q* **by**(*auto*)
hence *sound* (*Q o* φ) **by**(*auto intro*:*sound-intros*)
with *well-def-wp-healthy*[*OF wB*] **have** $0 \leq$ *wp B* (*Q o* φ) *s* **by**(*auto*)
with *G* **have** «*G*» *s* .& *wp B* (*Q o* φ) *s* = *wp B* (*Q o* φ) *s* **by**(*simp*)
}
finally show *?thesis* **.**
qed

4.9.6 The Algebra of Data Refinement

Program refinement implies a trivial data refinement:

lemma *refines-drefines*:
fixes *a*::*'s prog*
assumes *rab*: *a* $\sqsubseteq$ *b* **and** *wb*: *well-def b*
shows *drefines* (λ*s*. *s*) *G a b*
proof(*intro drefinesI2 wb*, *simp add*:*o-def*)
fix *P*::*'s* $\Rightarrow$ *real* **and** *Q*::*'s* $\Rightarrow$ *real* **and** *s*::*'s*
assume *sQ*: *unitary Q*
assume *P* $\Vdash$ *wp a Q* **hence** *P s* $\leq$ *wp a Q s* **by**(*auto*)
also from *rab sQ* **have** ... $\leq$ *wp b Q s* **by**(*auto*)
finally show *P s* $\leq$ *wp b Q s* **.**
qed

Data refinement is transitive:

lemma *dr-trans*[*trans*]:
fixes *A*::*'a prog* **and** *B*::*'b prog* **and** *C*::*'c prog*
assumes *drAB*: *drefines* φ *G A B*
and *drBC*: *drefines* φ' *G' B C*
and *Gimp*: $\bigwedge$*s*. *G' s* $\Longrightarrow$ *G* (φ' *s*)
shows *drefines* (φ *o* φ') *G' A C*
proof(*rule drefinesI*)
fix *P*::*'a* $\Rightarrow$ *real* **and** *Q*::*'a* $\Rightarrow$ *real* **and** *s*::*'a*
assume *uP*: *unitary P* **and** *uQ*: *unitary Q*

```
   and wpA: P ⊩ wp A Q

  have «G′» && «G o φ′» = «G′»
  proof(rule ext, unfold exp-conj-def)
    fix x
    show «G′» x .& «G o φ′» x = «G′» x (is ?X)
    proof(cases G′ x)
      case False then show ?X by(simp)
    next
      case True
      moreover
      with Gimp have (G o φ′) x by(simp add:o-def)
      ultimately
      show ?X by(simp)
    qed
  qed

  with uP
  have «G′» && (P o (φ o φ′)) = «G′» && ((«G» && (P o φ)) o φ′)
    by(simp add:exp-conj-assoc o-assoc)

  also {
    from uP uQ wpA and drAB
    have «G» && (P o φ) ⊩ wp B (Q o φ)
      by(blast intro:drefinesD)

    with drBC and uP uQ
    have «G′» && ((«G» && (P o φ)) o φ′) ⊩ wp C ((Q o φ) o φ′)
      by(blast intro:unitary-intros drefinesD)
  }

  finally
  show «G′» && (P o (φ o φ′)) ⊩ wp C (Q o (φ o φ′))
    by(simp add:o-assoc)
qed
```

Data refinement composes with program refinement:

```
lemma pr-dr-trans[trans]:
  assumes prAB: A ⊑ B
      and drBC: drefines φ G B C
  shows drefines φ G A C
proof(rule drefinesI)
  fix P and Q
  assume uP:  unitary P
     and uQ:  unitary Q
     and wpA: P ⊩ wp A Q

  note wpA
  also from uQ and prAB have wp A Q ⊩ wp B Q by(blast)
```

finally have *P* ⊩ *wp B Q* **.**
with *uP uQ drBC*
show «*G*» && (*P o φ*) ⊩ *wp C* (*Q o φ*) **by**(*blast intro*:*drefinesD*)
qed

lemma *dr-pr-trans*[*trans*]:
assumes *drAB*: *drefines φ G A B*
assumes *prBC*: *B* ⊑ *C*
shows *drefines φ G A C*
proof(*rule drefinesI*)
fix *P* **and** *Q*
assume *uP*: *unitary P*
and *uQ*: *unitary Q*
and *wpA*: *P* ⊩ *wp A Q*

with *drAB* **have** «*G*» && (*P o φ*) ⊩ *wp B* (*Q o φ*) **by**(*blast intro*:*drefinesD*)
also from *uQ prBC* **have** ... ⊩ *wp C* (*Q o φ*) **by**(*blast*)
finally show «*G*» && (*P o φ*) ⊩ ... **.**
qed

If the projection φ commutes with the transformer, then data refinement is reflexive:

lemma *dr-refl*:
assumes *wa*: *well-def a*
and *comm*: ⋀*Q*. *unitary Q* ⟹ *wp a Q o φ* ⊩ *wp a* (*Q o φ*)
shows *drefines φ G a a*
proof(*intro drefinesI2 wa*)
fix *P* **and** *Q* **and** *s*
assume *wp*: *P* ⊩ *wp a Q*
assume *uQ*: *unitary Q*

have (*P o φ*) *s* = *P* (*φ s*) **by**(*simp*)
also from *wp* **have** ... ≤ *wp a Q* (*φ s*) **by**(*blast*)
also {
from *comm uQ* **have** *wp a Q o φ* ⊩ *wp a* (*Q o φ*) **by**(*blast*)
hence (*wp a Q o φ*) *s* ≤ *wp a* (*Q o φ*) *s* **by**(*blast*)
hence *wp a Q* (*φ s*) ≤ ... **by**(*simp*)
}
finally show (*P o φ*) *s* ≤ *wp a* (*Q o φ*) *s* **.**
qed

Correspondence implies data refinement

lemma *pcorres-drefine*:
assumes *corres*: *pcorres φ G A C*
and *wC*: *well-def C*
shows *drefines φ G A C*
proof
fix *P* **and** *Q*
assume *uP*: *unitary P* **and** *uQ*: *unitary Q*

```
   and wpA: P ⊩ wp A Q
 from wpA have P o φ ⊩ wp A Q o φ by(simp add:o-def le-fun-def)
 hence «G» && (P o φ) ⊩ «G» && (wp A Q o φ)
  by(rule exp-conj-mono-right)
 also from corres uQ
 have ... = «G» && (wp C (Q o φ)) by(rule pcorresD)
 also
 have ... ⊩ wp C (Q o φ)
 proof(rule le-funI)
  fix s
  from uQ have unitary (Q o φ) by(rule unitary-intros)
  with well-def-wp-healthy[OF wC] have nn-wpC: 0 ≤ wp C (Q o φ) s by(blast)
  show («G» && wp C (Q o φ)) s ≤ wp C (Q o φ) s
  proof(cases G s)
   case True
   with nn-wpC show ?thesis by(simp add:exp-conj-def)
  next
   case False note this
   moreover {
    from uQ have unitary (Q o φ) by(simp)
    with well-def-wp-healthy[OF wC] have wp C (Q o φ) s ≤ 1 by(auto)
   }
   moreover note nn-wpC
   ultimately show ?thesis by(simp add:exp-conj-def)
  qed
 qed
 finally show «G» && (P o φ) ⊩ wp C (Q o φ) .
qed
```

Any *data* refinement of a deterministic program is correspondence. This is the analogous result to that relating program refinement and equivalence.

```
lemma drefines-determ:
 fixes a::'a prog and b::'b prog
 assumes da: determ (wp a)
   and wa: well-def a
   and wb: well-def b
   and dr: drefines φ G a b
 shows pcorres φ G a b
```

The proof follows exactly the same form as that for program refinement: Assuming that correspondence *doesn't* hold, we show that *wp b* is not feasible, and thus not healthy, contradicting the assumption.

```
proof(rule pcorresI, rule contrapos-pp)
 from wb show feasible (wp b) by(auto)

 note ha = well-def-wp-healthy[OF wa]
 note hb = well-def-wp-healthy[OF wb]

 from wb have sublinear (wp b) by(auto)
```

```
moreover from hb have feasible (wp b) by(auto)
ultimately have sab: sub-add (wp b) by(rule sublinear-subadd)

fix Q::'a ⇒ real
assume uQ: unitary Q
hence uQφ: unitary (Q o φ) by(auto)
assume ne: «G» && (wp a Q o φ) ≠ «G» && wp b (Q o φ)
hence ne': wp a Q o φ ≠ wp b (Q o φ) by(auto)
```

From refinement, « G » && $(wp\ a\ Q \circ \varphi)$ lies below « G » && $wp\ b\ (Q \circ \varphi)$.

```
from ha uQ
have gle: «G» && (wp a Q o φ) ⊩ wp b (Q o φ) by(blast intro!:drefinesD[OF dr])
have le: «G» && (wp a Q o φ) ⊩ «G» && wp b (Q o φ)
  unfolding exp-conj-def
proof(rule le-funI)
  fix s
  from gle have «G» s .& (wp a Q o φ) s ≤ wp b (Q o φ) s
    unfolding exp-conj-def by(auto)
  hence «G» s .& («G» s .& (wp a Q o φ) s) ≤ «G» s .& wp b (Q o φ) s
    by(auto intro:pconj-mono)
  moreover from uQ ha have wp a Q (φ s) ≤ 1
    by(auto dest:healthy-bounded-byD)
  moreover from uQ ha have 0 ≤ wp a Q (φ s)
    by(auto)
  ultimately
  show « G » s .& (wp a Q ∘ φ) s ≤ « G » s .& wp b (Q ∘ φ) s
    by(simp add:pconj-assoc)
qed
```

If the programs do not correspond, the terms must differ somewhere, and given the previous result, the second must be somewhere strictly larger than the first:

```
have nle: ∃s. («G» && (wp a Q o φ)) s < («G» && wp b (Q o φ)) s
proof(rule contrapos-np[OF ne], rule ext, rule antisym)
  fix s
  from le show («G» && (wp a Q o φ)) s ≤ («G» && wp b (Q o φ)) s
    by(blast)
next
  fix s
  assume ¬ (∃s. («G» && (wp a Q ∘ φ)) s < («G» && wp b (Q ∘ φ)) s)
  thus  («G» && (wp b (Q ∘ φ))) s ≤ («G» && (wp a Q ∘ φ)) s
    by(simp add:not-less)
qed
from this obtain s where less-s:
  («G» && (wp a Q ∘ φ)) s < («G» && wp b (Q ∘ φ)) s
  by(blast)
```

The transformers themselves must differ at this point:

```
hence larger: wp a Q (φ s) < wp b (Q ∘ φ) s
proof(cases G s)
```

```
  case True
  moreover from ha uQ have 0 ≤ wp a Q (φ s)
   by(blast)
  moreover from hb uQφ have 0 ≤ wp b (Q o φ) s
   by(blast)
  moreover note less-s
  ultimately show ?thesis by(simp add:exp-conj-def)
 next
  case False
  moreover from ha uQ have wp a Q (φ s) ≤ 1
   by(blast)
  moreover {
   from uQ have bounded-by 1 (Q o φ)
    by(blast)
   moreover from unitary-sound[OF uQ]
   have sound (Q o φ) by(auto)
   ultimately have wp b (Q o φ) s ≤ 1
    using hb by(auto)
  }
  moreover note less-s
  ultimately show ?thesis by(simp add:exp-conj-def)
 qed
 from less-s have («G» && (wp a Q ∘ φ)) s ≠ («G» && wp b (Q ∘ φ)) s
  by(force)
```

G must also hold, as otherwise both would be zero.

```
 hence G-s: G s
 proof(rule contrapos-np)
  assume nG: ¬ G s
  moreover from ha uQ have wp a Q (φ s) ≤ 1
   by(blast)
  moreover {
   from uQ have bounded-by 1 (Q o φ)
    by(blast)
   moreover from unitary-sound[OF uQ]
   have sound (Q o φ) by(auto)
   ultimately have wp b (Q o φ) s ≤ 1
    using hb by(auto)
  }
  ultimately
  show («G» && (wp a Q ∘ φ)) s = («G» && wp b (Q ∘ φ)) s
   by(simp add:exp-conj-def)
 qed
```

Take a carefully constructed expectation:

```
 let ?Qc = λs. bound-of Q − Q s
 have bQc: bounded-by 1 ?Qc
 proof(rule bounded-byI)
  fix s
```

```
    from uQ have bound-of Q ≤ 1 and 0 ≤ Q s by(auto)
    thus bound-of Q − Q s ≤ 1 by(auto)
  qed
  have sQc: sound ?Qc
  proof(rule soundI)
    from bQc show bounded ?Qc by(auto)

    show nneg ?Qc
    proof(rule nnegI)
      fix s
      from uQ have Q s ≤ bound-of Q by(auto)
      thus 0 ≤ bound-of Q − Q s by(auto)
    qed
  qed
```

By the maximality of *wp a*, *wp b* must violate feasibility, by mapping *s* to something strictly greater than *bound-of Q*.

```
  from uQ have 0 ≤ bound-of Q by(auto)
  with da have bound-of Q = wp a (λs. bound-of Q) (φ s)
    by(simp add:maximalD determ-maximalD)
  also have wp a (λs. bound-of Q) (φ s) = wp a (λs. Q s + ?Qc s) (φ s)
    by(simp)
  also {
    from da have additive (wp a) by(blast)
    with uQ sQc
    have wp a (λs. Q s + ?Qc s) (φ s) =
         wp a Q (φ s) + wp a ?Qc (φ s) by(subst additiveD, blast+)
  }
  also {
    from ha and sQc and bQc
    have «G» && (wp a ?Qc o φ) ⊩ wp b (?Qc o φ)
      by(blast intro!:drefinesD[OF dr])
    hence («G» && (wp a ?Qc o φ)) s ≤ wp b (?Qc o φ) s
      by(blast)
    moreover from sQc and ha
    have 0 ≤ wp a (λs. bound-of Q − Q s) (φ s)
      by(blast)
    ultimately
    have wp a ?Qc (φ s) ≤ wp b (?Qc o φ) s
      using G-s by(simp add:exp-conj-def)
    hence wp a Q (φ s) + wp a ?Qc (φ s) ≤ wp a Q (φ s) + wp b (?Qc o φ) s
      by(rule add-left-mono)
    also with larger
    have wp a Q (φ s) + wp b (?Qc o φ) s <
         wp b (Q o φ) s + wp b (?Qc o φ) s
      by(auto)
    finally
    have wp a Q (φ s) + wp a ?Qc (φ s) <
         wp b (Q o φ) s + wp b (?Qc o φ) s .
```

```
 }
 also from sab and unitary-sound[OF uQ] and sQc
 have wp b (Q o φ) s + wp b (?Qc o φ) s ≤
      wp b (λs. (Q o φ) s + (?Qc o φ) s) s
  by(blast)
 also have ... = wp b (λs. bound-of Q) s
  by(simp)
 finally
 show ¬ feasible (wp b)
 proof(rule contrapos-pn)
  assume fb: feasible (wp b)
  have bounded-by (bound-of Q) (λs. bound-of Q) by(blast)
  hence bounded-by (bound-of Q) (wp b (λs. bound-of Q))
   using uQ by(blast intro:feasible-boundedD[OF fb])
  hence wp b (λs. bound-of Q) s ≤ bound-of Q by(blast)
  thus ¬ bound-of Q < wp b (λs. bound-of Q) s by(simp)
 qed
qed
```

4.9.7 Structural Rules for Correspondence

```
lemma pcorres-Skip:
 pcorres φ G Skip Skip
 by(simp add:pcorres-def wp-eval)
```

Correspondence composes over sequential composition.

```
lemma pcorres-Seq:
 fixes A::'b prog and B::'c prog
  and C::'b prog and D::'c prog
  and φ::'c ⇒ 'b
 assumes pcAB: pcorres φ G A B
   and pcCD: pcorres φ H C D
   and wA: well-def A and wB: well-def B
   and wC: well-def C and wD: well-def D
   and p3p2: ⋀Q. unitary Q ⟹ «I» && wp B Q = wp B («H» && Q)
   and p1p3: ⋀s. G s ⟹ I s
 shows pcorres φ G (A;;C) (B;;D)
proof(rule pcorresI)
 fix Q::'b ⇒ real
 assume uQ: unitary Q
 with well-def-wp-healthy[OF wC] have uCQ: unitary (wp C Q) by(auto)
 from uQ well-def-wp-healthy[OF wD] have uDQ: unitary (wp D (Q o φ))
  by(auto dest:unitary-comp)

 have p3p1: ⋀R S. ⟦ unitary R; unitary S; «I» && R = «I» && S ⟧ ⟹
           «G» && R = «G» && S
 proof(rule ext)
  fix R::'c ⇒ real and S::'c ⇒ real and s::'c
  assume a3: «I» && R = «I» && S
```

```
    and uR: unitary R and uS: unitary S
  show («G» && R) s = («G» && S) s
  proof(simp add:exp-conj-def, cases G s)
    case False note this
    moreover from uR have R s ≤ 1 by(blast)
    moreover from uS have S s ≤ 1 by(blast)
    ultimately show «G» s .& R s = «G» s .& S s
      by(simp)
  next
    case True note p1 = this
    with p1p3 have I s by(blast)
    with fun-cong[OF a3, where x=s] have 1 .& R s = 1 .& S s
      by(simp add:exp-conj-def)
    with p1 show «G» s .& R s = «G» s .& S s
      by(simp)
  qed
qed

show «G» && (wp (A;;C) Q o φ) = «G» && wp (B;;D) (Q o φ)
proof(simp add:wp-eval)
  from uCQ pcAB have «G» && (wp A (wp C Q) ∘ φ) =
                  «G» && wp B ((wp C Q) ∘ φ)
    by(auto dest:pcorresD)
  also have «G» && wp B ((wp C Q) ∘ φ) =
          «G» && wp B (wp D (Q ∘ φ))
  proof(rule p3p1)
    from uCQ well-def-wp-healthy[OF wB] show unitary (wp B (wp C Q ∘ φ))
      by(auto intro:unitary-comp)
    from uDQ well-def-wp-healthy[OF wB] show unitary (wp B (wp D (Q ∘ φ)))
      by(auto)

    from uQ have « H » && (wp C Q ∘ φ) = « H » && wp D (Q ∘ φ)
      by(blast intro:pcorresD[OF pcCD])
    thus « I » && wp B (wp C Q ∘ φ) = « I » && wp B (wp D (Q ∘ φ))
      by(simp add:p3p2 uCQ uDQ)
  qed
  finally show «G» && (wp A (wp C Q) ∘ φ) = «G» && wp B (wp D (Q ∘ φ)) .
qed
qed
```

4.9.8 Structural Rules for Data Refinement

```
lemma dr-Skip:
  fixes φ::'c ⇒ 'b
  shows drefines φ G Skip Skip
proof(intro drefinesI2 wd-intros)
  fix P::'b ⇒ real and Q::'b ⇒ real and s::'c
  assume P ⊩ wp Skip Q
  hence (P o φ) s ≤ wp Skip Q (φ s) by(simp, blast)
```

```
  thus (P o φ) s ≤ wp Skip (Q o φ) s by(simp add:wp-eval)
qed

lemma dr-Abort:
  fixes φ::'c ⇒ 'b
  shows drefines φ G Abort Abort
proof(intro drefinesI2 wd-intros)
  fix P::'b ⇒ real and Q::'b ⇒ real and s::'c
  assume P ⊩ wp Abort Q
  hence (P o φ) s ≤ wp Abort Q (φ s) by(auto)
  thus (P o φ) s ≤ wp Abort (Q o φ) s by(simp add:wp-eval)
qed

lemma dr-Apply:
  fixes φ::'c ⇒ 'b
  assumes commutes: f o φ = φ o g
  shows drefines φ G (Apply f) (Apply g)
proof(intro drefinesI2 wd-intros)
  fix P::'b ⇒ real and Q::'b ⇒ real and s::'c

  assume wp: P ⊩ wp (Apply f) Q
  hence P ⊩ (Q o f) by(simp add:wp-eval)
  hence P (φ s) ≤ (Q o f) (φ s) by(blast)
  also have ... = Q ((f o φ) s) by(simp)
  also with commutes
  have ... = ((Q o φ) o g) s by(simp)
  also have ... = wp (Apply g) (Q o φ) s
    by(simp add:wp-eval)
  finally show (P o φ) s ≤ wp (Apply g) (Q o φ) s by(simp)
qed

lemma dr-Seq:
  assumes drAB: drefines φ P A B
      and drBC: drefines φ Q C D
      and wpB: «P» ⊩ wp B «Q»
      and wB:  well-def B
      and wC:  well-def C
      and wD:  well-def D
  shows drefines φ P (A;;C) (B;;D)
proof
  fix R and S
  assume uR: unitary R and uS: unitary S
    and wpAC: R ⊩ wp (A;;C) S

  from uR
  have «P» && (R o φ) = «P» && («P» && (R o φ))
    by(simp add:exp-conj-assoc)

  also {
```

```
    from well-def-wp-healthy[OF wC] uR uS
     and wpAC[unfolded eval-wp-Seq o-def]
    have «P» && (R o φ) ⊩ wp B (wp C S o φ)
      by(auto intro:drefinesD[OF drAB])
    with wpB well-def-wp-healthy[OF wC] uS
        sublinear-sub-conj[OF well-def-wp-sublinear, OF wB]
    have «P» && («P» && (R o φ)) ⊩ wp B («Q» && (wp C S o φ))
      by(auto intro!:entails-combine dest!:unitary-sound)
  }

  also {
    from uS well-def-wp-healthy[OF wC]
    have «Q» && (wp C S o φ) ⊩ wp D (S o φ)
      by(auto intro!:drefinesD[OF drBC])
    with well-def-wp-healthy[OF wB] well-def-wp-healthy[OF wC]
        well-def-wp-healthy[OF wD] and unitary-sound[OF uS]
    have wp B («Q» && (wp C S o φ)) ⊩ wp B (wp D (S o φ))
      by(blast intro!:mono-transD)
  }

  finally
  show «P» && (R o φ) ⊩ wp (B;;D) (S o φ)
    unfolding wp-eval o-def .
qed

lemma dr-repeat:
  fixes φ :: 'a ⇒ 'b
  assumes dr-ab: drefines φ G a b
      and Gpr:  «G» ⊩ wp b «G»
      and wa:   well-def a
      and wb:   well-def b
  shows drefines φ G (repeat n a) (repeat n b) (is ?X n)
proof(induct n)
  show ?X 0 by(simp add:dr-Skip)

  fix n
  assume IH: ?X n
  thus ?X (Suc n) by(auto intro!:dr-Seq Gpr assms wd-intros)
qed

end
```

4.10 Structured Reasoning

theory *StructuredReasoning* **imports** *Algebra* **begin**

By linking the algebraic, the syntactic, and the semantic views of computation, we derive a set of rules for decomposing expectation entailment proofs, firstly over the syntactic structure of a program, and secondly over the refinement relation. These rules also form the basis for automated reasoning.

4.10.1 Syntactic Decomposition

lemma *wp-Abort*:
$(\lambda s.\ 0) \Vdash wp\ Abort\ Q$
unfolding *wp-eval* **by**(*simp*)

lemma *wlp-Abort*:
$(\lambda s.\ 1) \Vdash wlp\ Abort\ Q$
unfolding *wp-eval* **by**(*simp*)

lemma *wp-Skip*:
$P \Vdash wp\ Skip\ P$
unfolding *wp-eval* **by**(*blast*)

lemma *wlp-Skip*:
$P \Vdash wlp\ Skip\ P$
unfolding *wp-eval* **by**(*blast*)

lemma *wp-Apply*:
$Q\ o\ f \Vdash wp\ (Apply\ f)\ Q$
unfolding *wp-eval* **by**(*simp*)

lemma *wlp-Apply*:
$Q\ o\ f \Vdash wlp\ (Apply\ f)\ Q$
unfolding *wp-eval* **by**(*simp*)

lemma *wp-Seq*:
assumes *ent-a*: $P \Vdash wp\ a\ Q$
and *ent-b*: $Q \Vdash wp\ b\ R$
and *wa*: *well-def a*
and *wb*: *well-def b*
and *s-Q*: *sound Q*
and *s-R*: *sound R*
shows $P \Vdash wp\ (a\ ;;\ b)\ R$
proof −
note *ha* = *well-def-wp-healthy*[*OF wa*]
note *hb* = *well-def-wp-healthy*[*OF wb*]
note *ent-a*
also from *ent-b ha hb s-Q s-R* **have** $wp\ a\ Q \Vdash wp\ a\ (wp\ b\ R)$
by(*blast intro*:*healthy-monoD2*)
finally show *?thesis* **by**(*simp add*:*wp-eval*)
qed

lemma *wlp-Seq*:
assumes *ent-a*: $P \Vdash wlp\ a\ Q$
and *ent-b*: $Q \Vdash wlp\ b\ R$
and *wa*: *well-def a*

    and *wb*: *well-def b*
    and *u-Q*: *unitary Q*
    and *u-R*: *unitary R*
  shows $P \Vdash wlp\ (a\ ;;\ b)\ R$
proof −
  note *ha* = *well-def-wlp-nearly-healthy*[*OF wa*]
  note *hb* = *well-def-wlp-nearly-healthy*[*OF wb*]
  note *ent-a*
  also from *ent-b ha hb u-Q u-R* **have** $wlp\ a\ Q \Vdash wlp\ a\ (wlp\ b\ R)$
    by(*blast intro*:*nearly-healthy-monoD*[*OF ha*])
  finally show *?thesis* **by**(*simp add*:*wp-eval*)
qed

lemma *wp-PC*:
  $(\lambda s.\ P\ s * wp\ a\ Q\ s + (1 - P\ s) * wp\ b\ Q\ s) \Vdash wp\ (a\ {}_{P}\oplus\ b)\ Q$
  by(*simp add*:*wp-eval*)

lemma *wlp-PC*:
  $(\lambda s.\ P\ s * wlp\ a\ Q\ s + (1 - P\ s) * wlp\ b\ Q\ s) \Vdash wlp\ (a\ {}_{P}\oplus\ b)\ Q$
  by(*simp add*:*wp-eval*)

A simpler rule for when the probability does not depend on the state.

lemma *PC-fixed*:
  assumes *wpa*: $P \Vdash a\ ab\ R$
    and *wpb*: $Q \Vdash b\ ab\ R$
    and *np*: $0 \le p$ **and** *bp*: $p \le 1$
  shows $(\lambda s.\ p * P\ s + (1 - p) * Q\ s) \Vdash (a\ {}_{(\lambda s.\ p)}\oplus\ b)\ ab\ R$
  unfolding *PC-def*
proof(*rule le-funI*)
  fix *s*
  from *wpa* **and** *np* **have** $p * P\ s \le p * a\ ab\ R\ s$
    by(*auto intro*:*mult-left-mono*)
  moreover {
    from *bp* **have** $0 \le 1 - p$ **by**(*simp*)
    with *wpb* **have** $(1 - p) * Q\ s \le (1 - p) * b\ ab\ R\ s$
      by(*auto intro*:*mult-left-mono*)
  }
  ultimately show $p * P\ s + (1 - p) * Q\ s \le$
        $p * a\ ab\ R\ s + (1 - p) * b\ ab\ R\ s$
    by(*rule add-mono*)
qed

lemma *wp-PC-fixed*:
  $[\![\ P \Vdash wp\ a\ R;\ Q \Vdash wp\ b\ R;\ 0 \le p;\ p \le 1\ ]\!] \Longrightarrow$
  $(\lambda s.\ p * P\ s + (1 - p) * Q\ s) \Vdash wp\ (a\ {}_{(\lambda s.\ p)}\oplus\ b)\ R$
  by(*simp add*:*wp-def PC-fixed*)

lemma *wlp-PC-fixed*:
  $[\![\ P \Vdash wlp\ a\ R;\ Q \Vdash wlp\ b\ R;\ 0 \le p;\ p \le 1\ ]\!] \Longrightarrow$

$(\lambda s.\ p * P\ s + (1 - p) * Q\ s) \Vdash wlp\ (a\ {}_{(\lambda s.\ p)}\!\oplus b)\ R$
by(*simp add*:*wlp-def PC-fixed*)

lemma *wp-DC*:
$(\lambda s.\ min\ (wp\ a\ Q\ s)\ (wp\ b\ Q\ s)) \Vdash wp\ (a \bigsqcap b)\ Q$
unfolding *wp-eval* **by**(*simp*)

lemma *wlp-DC*:
$(\lambda s.\ min\ (wlp\ a\ Q\ s)\ (wlp\ b\ Q\ s)) \Vdash wlp\ (a \bigsqcap b)\ Q$
unfolding *wp-eval* **by**(*simp*)

Combining annotations for both branches:

lemma *DC-split*:
fixes $a::'s\ prog$ **and** b
assumes *wpa*: $P \Vdash a\ ab\ R$
and *wpb*: $Q \Vdash b\ ab\ R$
shows $(\lambda s.\ min\ (P\ s)\ (Q\ s)) \Vdash (a \bigsqcap b)\ ab\ R$
unfolding *DC-def*
proof(*rule le-funI*)
fix *s*
from *wpa wpb*
have $P\ s \le a\ ab\ R\ s$ **and** $Q\ s \le b\ ab\ R\ s$ **by**(*auto*)
thus $min\ (P\ s)\ (Q\ s) \le min\ (a\ ab\ R\ s)\ (b\ ab\ R\ s)$ **by**(*auto*)
qed

lemma *wp-DC-split*:
$[\![P \Vdash wp\ prog\ R;\ Q \Vdash wp\ prog'\ R]\!] \Longrightarrow$
$(\lambda s.\ min\ (P\ s)\ (Q\ s)) \Vdash wp\ (prog \bigsqcap prog')\ R$
by(*simp add*:*wp-def DC-split*)

lemma *wlp-DC-split*:
$[\![P \Vdash wlp\ prog\ R;\ Q \Vdash wlp\ prog'\ R]\!] \Longrightarrow$
$(\lambda s.\ min\ (P\ s)\ (Q\ s)) \Vdash wlp\ (prog \bigsqcap prog')\ R$
by(*simp add*:*wlp-def DC-split*)

lemma *wp-DC-split-same*:
$[\![P \Vdash wp\ prog\ Q;\ P \Vdash wp\ prog'\ Q]\!] \Longrightarrow P \Vdash wp\ (prog \bigsqcap prog')\ Q$
unfolding *wp-eval* **by**(*blast intro*:*min.boundedI*)

lemma *wlp-DC-split-same*:
$[\![P \Vdash wlp\ prog\ Q;\ P \Vdash wlp\ prog'\ Q]\!] \Longrightarrow P \Vdash wlp\ (prog \bigsqcap prog')\ Q$
unfolding *wp-eval* **by**(*blast intro*:*min.boundedI*)

lemma *SetPC-split*:
fixes $f::'x \Rightarrow 'y\ prog$
and $p::'y \Rightarrow 'x \Rightarrow real$
assumes *rec*: $\bigwedge x\ s.\ x \in supp\ (p\ s) \Longrightarrow P\ x \Vdash f\ x\ ab\ Q$
and *nnp*: $\bigwedge s.\ nneg\ (p\ s)$
shows $(\lambda s.\ \sum x \in supp\ (p\ s).\ p\ s\ x * P\ x\ s) \Vdash SetPC\ f\ p\ ab\ Q$

unfolding *SetPC-def*
proof(*rule le-funI*)
fix *s*
from *rec* **have** $\bigwedge x.\ x \in supp\ (p\ s) \Longrightarrow P\ x\ s \le f\ x\ ab\ Q\ s$ **by**(*blast*)
moreover from *nnp* **have** $\bigwedge x.\ 0 \le p\ s\ x$ **by**(*blast*)
ultimately have $\bigwedge x.\ x \in supp\ (p\ s) \Longrightarrow p\ s\ x * P\ x\ s \le p\ s\ x * f\ x\ ab\ Q\ s$
by(*blast intro*:*mult-left-mono*)
thus $(\sum x \in supp\ (p\ s).\ p\ s\ x * P\ x\ s) \le (\sum x \in supp\ (p\ s).\ p\ s\ x * f\ x\ ab\ Q\ s)$
by(*rule sum-mono*)
qed

lemma *wp-SetPC-split*:
$[\![\bigwedge x\ s.\ x \in supp\ (p\ s) \Longrightarrow P\ x \Vdash wp\ (f\ x)\ Q;\ \bigwedge s.\ nneg\ (p\ s)\ ]\!] \Longrightarrow$
$(\lambda s.\ \sum x \in supp\ (p\ s).\ p\ s\ x * P\ x\ s) \Vdash wp\ (SetPC\ f\ p)\ Q$
by(*simp add*:*wp-def SetPC-split*)

lemma *wlp-SetPC-split*:
$[\![\bigwedge x\ s.\ x \in supp\ (p\ s) \Longrightarrow P\ x \Vdash wlp\ (f\ x)\ Q;\ \bigwedge s.\ nneg\ (p\ s)\ ]\!] \Longrightarrow$
$(\lambda s.\ \sum x \in supp\ (p\ s).\ p\ s\ x * P\ x\ s) \Vdash wlp\ (SetPC\ f\ p)\ Q$
by(*simp add*:*wlp-def SetPC-split*)

lemma *wp-SetDC-split*:
$[\![\bigwedge s\ x.\ x \in S\ s \Longrightarrow P \Vdash wp\ (f\ x)\ Q;\ \bigwedge s.\ S\ s \neq \{\}\ ]\!] \Longrightarrow$
$P \Vdash wp\ (SetDC\ f\ S)\ Q$
by(*rule le-funI*, *unfold wp-eval*, *blast intro*!:*cInf-greatest*)

lemma *wlp-SetDC-split*:
$[\![\bigwedge s\ x.\ x \in S\ s \Longrightarrow P \Vdash wlp\ (f\ x)\ Q;\ \bigwedge s.\ S\ s \neq \{\}\ ]\!] \Longrightarrow$
$P \Vdash wlp\ (SetDC\ f\ S)\ Q$
by(*rule le-funI*, *unfold wp-eval*, *blast intro*!:*cInf-greatest*)

lemma *wp-SetDC*:
assumes *wp*: $\bigwedge s\ x.\ x \in S\ s \Longrightarrow P\ x \Vdash wp\ (f\ x)\ Q$
and *ne*: $\bigwedge s.\ S\ s \neq \{\}$
and *sP*: $\bigwedge x.\ sound\ (P\ x)$
shows $(\lambda s.\ Inf\ ((\lambda x.\ P\ x\ s)\ `\ S\ s)) \Vdash wp\ (SetDC\ f\ S)\ Q$
using *assms* **by**(*intro le-funI*, *simp add*:*wp-eval*, *blast intro*!:*cInf-mono*)

lemma *wlp-SetDC*:
assumes *wp*: $\bigwedge s\ x.\ x \in S\ s \Longrightarrow P\ x \Vdash wlp\ (f\ x)\ Q$
and *ne*: $\bigwedge s.\ S\ s \neq \{\}$
and *sP*: $\bigwedge x.\ sound\ (P\ x)$
shows $(\lambda s.\ Inf\ ((\lambda x.\ P\ x\ s)\ `\ S\ s)) \Vdash wlp\ (SetDC\ f\ S)\ Q$
using *assms* **by**(*intro le-funI*, *simp add*:*wp-eval*, *blast intro*!:*cInf-mono*)

lemma *wp-Embed*:
$P \Vdash t\ Q \Longrightarrow P \Vdash wp\ (Embed\ t)\ Q$
by(*simp add*:*wp-def Embed-def*)

lemma *wlp-Embed*:
$P \Vdash t\ Q \Longrightarrow P \Vdash wlp\ (Embed\ t)\ Q$
by(*simp add*:*wlp-def Embed-def*)

lemma *wp-Bind*:
$[\![\bigwedge s.\ P\ s \leq wp\ (a\ (f\ s))\ Q\ s]\!] \Longrightarrow P \Vdash wp\ (Bind\ f\ a)\ Q$
by(*auto simp*:*wp-def Bind-def*)

lemma *wlp-Bind*:
$[\![\bigwedge s.\ P\ s \leq wlp\ (a\ (f\ s))\ Q\ s]\!] \Longrightarrow P \Vdash wlp\ (Bind\ f\ a)\ Q$
by(*auto simp*:*wlp-def Bind-def*)

lemma *wp-repeat*:
$[\![P \Vdash wp\ a\ Q;\ Q \Vdash wp\ (repeat\ n\ a)\ R;$
$well\text{-}def\ a;\ sound\ Q;\ sound\ R]\!] \Longrightarrow P \Vdash wp\ (repeat\ (Suc\ n)\ a)\ R$
by(*auto intro*!:*wp-Seq wd-intros*)

lemma *wlp-repeat*:
$[\![P \Vdash wlp\ a\ Q;\ Q \Vdash wlp\ (repeat\ n\ a)\ R;$
$well\text{-}def\ a;\ unitary\ Q;\ unitary\ R]\!] \Longrightarrow P \Vdash wlp\ (repeat\ (Suc\ n)\ a)\ R$
by(*auto intro*!:*wlp-Seq wd-intros*)

Note that the loop rules presented in section Section 4.8 are of the same form, and would belong here, had they not already been stated.

The following rules are specialisations of those for general transformers, and are easier for the unifier to match.

lemmas *wp-strengthen-post*=
entails-strengthen-post[**where** *t*=*wp a* **for** *a*]

lemma *wlp-strengthen-post*:
$P \Vdash wlp\ a\ Q \Longrightarrow nearly\text{-}healthy\ (wlp\ a) \Longrightarrow unitary\ R \Longrightarrow Q \Vdash R \Longrightarrow unitary\ Q \Longrightarrow$
$P \Vdash wlp\ a\ R$
by(*blast intro*:*entails-trans*)

lemmas *wp-weaken-pre*=
entails-weaken-pre[**where** *t*=*wp a* **for** *a*]
lemmas *wlp-weaken-pre*=
entails-weaken-pre[**where** *t*=*wlp a* **for** *a*]

lemmas *wp-scale*=
entails-scale[**where** *t*=*wp a* **for** *a*, *OF - well-def-wp-healthy*]

4.10.2 Algebraic Decomposition

Refinement is a powerful tool for decomposition, belied by the simplicity of the rule. This is an *axiomatic* formulation of refinement (all annotations of the *a* are annotations of *b*), rather than an operational version (all traces of *b* are traces of *a*.

lemma *wp-refines*:
$\llbracket\, a \sqsubseteq b;\ P \Vdash wp\ a\ Q;\ sound\ Q \,\rrbracket \Longrightarrow P \Vdash wp\ b\ Q$
by(*auto intro*:*entails-trans*)

lemmas *wp-drefines* = *drefinesD*

4.10.3 Hoare triples

The Hoare triple, or validity predicate, is logically equivalent to the weakest-precondition entailment form. The benefit is that it allows us to define transitivity rules for computational (also/finally) reasoning.

definition
wp-valid :: $('a \Rightarrow real) \Rightarrow 'a\ prog \Rightarrow ('a \Rightarrow real) \Rightarrow bool$ (‹⦃-⦄ - ⦃-⦄*p*›)
where
$wp\text{-}valid\ P\ prog\ Q \equiv P \Vdash wp\ prog\ Q$

lemma *wp-validI*:
$P \Vdash wp\ prog\ Q \Longrightarrow \{\!|P|\!\}\ prog\ \{\!|Q|\!\}p$
unfolding *wp-valid-def* **by**(*assumption*)

lemma *wp-validD*:
$\{\!|P|\!\}\ prog\ \{\!|Q|\!\}p \Longrightarrow P \Vdash wp\ prog\ Q$
unfolding *wp-valid-def* **by**(*assumption*)

lemma *valid-Seq*:
$\llbracket\, \{\!|P|\!\}\ a\ \{\!|Q|\!\}p;\ \{\!|Q|\!\}\ b\ \{\!|R|\!\}p;\ well\text{-}def\ a;\ well\text{-}def\ b;\ sound\ Q;\ sound\ R \,\rrbracket \Longrightarrow$
$\{\!|P|\!\}\ a\ ;;\ b\ \{\!|R|\!\}p$
unfolding *wp-valid-def* **by**(*rule wp-Seq*)

We make it available to the computational reasoner:

declare *valid-Seq*[*trans*]

end

4.11 Loop Termination

theory *Termination* **imports** *Embedding StructuredReasoning Loops* **begin**

Termination for loops can be shown by classical means (using a variant, or a measure function), or by probabilistic means: We only need that the loop terminates *with probability one*.

4.11.1 Trivial Termination

A maximal transformer (program) doesn't affect termination. This is essentially saying that such a program doesn't abort (or diverge).

```
lemma maximal-Seq-term:
  fixes r::'s prog and s::'s prog
  assumes mr: maximal (wp r)
    and ws: well-def s
    and ts: (λs. 1) ⊩ wp s (λs. 1)
  shows (λs. 1) ⊩ wp (r ;; s) (λs. 1)
proof –
  note hs = well-def-wp-healthy[OF ws]
  have wp s (λs. 1) = (λs. 1)
  proof(rule antisym)
    show (λs. 1) ⊩ wp s (λs. 1) by(rule ts)
    have bounded-by 1 (wp s (λs. 1))
      by(auto intro!:healthy-bounded-byD[OF hs])
    thus wp s (λs. 1) ⊩ (λs. 1) by(auto intro!:le-funI)
  qed
  with mr show ?thesis
    by(simp add:wp-eval embed-bool-def maximalD)
qed
```

From any state where the guard does not hold, a loop terminates in a single step.

```
lemma term-onestep:
  assumes wb: well-def body
  shows «𝒩 G» ⊩ wp do G ⟶ body od (λs. 1)
proof(rule le-funI)
  note hb = well-def-wp-healthy[OF wb]
  fix s
  show «𝒩 G» s ≤ wp do G ⟶ body od (λs. 1) s
  proof(cases G s, simp-all add:wp-loop-nguard hb)
    from hb have sound (wp do G ⟶ body od (λs. 1))
      by(auto intro:healthy-sound[OF healthy-wp-loop])
    thus 0 ≤ wp do G ⟶ body od (λs. 1) s by(auto)
  qed
qed
```

4.11.2 Classical Termination

The first non-trivial termination result is quite standard: If we can provide a natural-number-valued measure, that decreases on every iteration, and implies termination on reaching zero, the loop terminates.

```
lemma loop-term-nat-measure-noinv:
  fixes m :: 's ⇒ nat and body :: 's prog
  assumes wb: well-def body
  and guard: ⋀s. m s = 0 ⟶ ¬ G s
  and variant: ⋀n. «λs. m s = Suc n» ⊩ wp body «λs. m s = n»
  shows λs. 1 ⊩ wp do G ⟶ body od (λs. 1)
proof –
  note hb = well-def-wp-healthy[OF wb]
  have ⋀n. (∀ s. m s = n ⟶ 1 ≤ wp do G ⟶ body od (λs. 1) s)
```

```
 proof(induct-tac n)
  fix n
  show ∀ s. m s = 0 ⟶ 1 ≤ wp do G ⟶ body od (λs. 1) s
  proof(clarify)
   fix s
   assume m s = 0
   with guard have ¬ G s by(blast)
   with hb show 1 ≤ wp do G ⟶ body od (λs. 1) s
    by(simp add:wp-loop-nguard)
  qed
  assume IH: ∀ s. m s = n ⟶ 1 ≤ wp do G ⟶ body od (λs. 1) s
  hence IH′: ∀ s. m s = n ⟶ 1 ≤ wp do G ⟶ body od «λs. True» s
   by(simp add:embed-bool-def)
  have ∀ s. m s = Suc n ⟶ 1 ≤ wp do G ⟶ body od «λs. True» s
  proof(intro fold-premise healthy-intros hb, rule le-funI)
   fix s
   show «λs. m s = Suc n» s ≤ wp do G ⟶ body od «λs. True» s
   proof(cases G s)
    case False
    hence 1 = «𝒩 G» s by(auto)
    also from wb have ... ≤ wp do G ⟶ body od (λs. 1) s
     by(rule le-funD[OF term-onestep])
    finally show ?thesis by(simp add:embed-bool-def)
   next
    case True note G = this
    from IH′ have «λs. m s = n» ⊩ wp do G ⟶ body od «λs. True»
     by(blast intro:use-premise healthy-intros hb)
    with variant wb
    have «λs. m s = Suc n» ⊩ wp (body ;; do G ⟶ body od) «λs. True»
     by(blast intro:wp-Seq wd-intros)
    hence «λs. m s = Suc n» s ≤ wp (body ;; do G ⟶ body od) «λs. True» s
     by(auto)
    also from hb G have ... = wp do G ⟶ body od «λs. True» s
     by(simp add:wp-loop-guard)
    finally show ?thesis .
   qed
  qed
  thus ∀ s. m s = Suc n ⟶ 1 ≤ wp do G ⟶ body od (λs. 1) s
   by(simp add:embed-bool-def)
 qed
 thus ?thesis by(auto)
qed
```

This version allows progress to depend on an invariant. Termination is then determined by the invariant's value in the initial state.

```
lemma loop-term-nat-measure:
 fixes m :: ′s ⇒ nat and body :: ′s prog
 assumes wb: well-def body
 and guard: ⋀s. m s = 0 ⟶ ¬ G s
```

```
  and variant: ⋀n. «λs. m s = Suc n» && «I» ⊩ wp body «λs. m s = n»
  and inv:     wp-inv G body «I»
  shows «I» ⊩ wp do G ⟶ body od (λs. 1)
proof −
  note hb = well-def-wp-healthy[OF wb]
  note scb = sublinear-sub-conj[OF well-def-wp-sublinear, OF wb]
  have «I» ⊩ wp do G ⟶ body od «λs. True»
  proof(rule use-premise, intro healthy-intros hb)
    fix s
    have ⋀n. (∀ s. m s = n ∧ I s ⟶ 1 ≤ wp do G ⟶ body od «λs. True» s)
    proof(induct-tac n)
      fix n
      show ∀ s. m s = 0 ∧ I s ⟶ 1 ≤ wp do G ⟶ body od «λs. True» s
      proof(clarify)
        fix s
        assume m s = 0
        with guard have ¬ G s by(blast)
        with hb show 1 ≤ wp do G ⟶ body od «λs. True» s
          by(simp add:wp-loop-nguard)
      qed
      assume IH: ∀ s. m s = n ∧ I s ⟶ 1 ≤ wp do G ⟶ body od «λs. True» s
      show ∀ s. m s = Suc n ∧ I s ⟶ 1 ≤ wp do G ⟶ body od «λs. True» s
      proof(intro fold-premise healthy-intros hb le-funI)
        fix s
        show «λs. m s = Suc n ∧ I s» s ≤ wp do G ⟶ body od «λs. True» s
        proof(cases G s)
          case False with hb show ?thesis
            by(simp add:wp-loop-nguard)
        next
          case True note G = this
          have «λs. m s = Suc n» && «I» && «G» =
              «λs. m s = Suc n» && («I» && «I») && «G»
            by(simp)
          also have ... = («λs. m s = Suc n» && «I») && («I» && «G»)
            by(simp add:exp-conj-assoc exp-conj-unitary del:exp-conj-idem)
          also have ... = («λs. m s = Suc n» && «I») && («G» && «I»)
            by(simp only:exp-conj-comm)
          also {
            from inv hb have «G» && «I» ⊩ wp body «I»
              by(rule wp-inv-stdD)
            with variant
            have («λs. m s = Suc n» && «I») && («G» && «I») ⊩
                wp body «λs. m s = n» && wp body «I»
              by(rule entails-frame)
          }
          also from scb
          have wp body «λs. m s = n» && wp body «I» ⊩
              wp body («λs. m s = n» && «I»)
            by(blast)
```

finally have «λ*s*. *m s* = *Suc n* » && « *I* » && « *G* » ⊩
wp body (« λ*s*. *m s* = *n* » && « *I* ») **.**
moreover {
from *IH* **have** «λ*s*. *m s* = *n* ∧ *I s*» ⊩ *wp do G* ⟶ *body od* «λ*s*. *True*»
by(*blast intro*:*use-premise healthy-intros hb*)
hence «λ*s*. *m s* = *n*» && «*I*» ⊩ *wp do G* ⟶ *body od* «λ*s*. *True*»
by(*simp add*:*exp-conj-std-split*)
}
ultimately
have «λ*s*. *m s* = *Suc n* » && « *I* » && « *G* » ⊩
wp (*body* ;; *do G* ⟶ *body od*) «λ*s*. *True*»
using *wb* **by**(*blast intro*:*wp-Seq wd-intros*)
hence («λ*s*. *m s* = *Suc n* ∧ *I s*» && « *G* ») *s* ≤
wp (*body* ;; *do G* ⟶ *body od*) «λ*s*. *True*» *s*
by(*auto simp*:*exp-conj-std-split*)
with *G* **have** «λ*s*. *m s* = *Suc n* ∧ *I s*» *s* ≤
wp (*body* ;; *do G* ⟶ *body od*) «λ*s*. *True*» *s*
by(*simp add*:*exp-conj-def*)
also from *hb G* **have** ... = *wp do G* ⟶ *body od* «λ*s*. *True*» *s*
by(*simp add*:*wp-loop-guard*)
finally show *?thesis* **.**
qed
qed
qed
moreover assume *I s*
ultimately show *1* ≤ *wp do G* ⟶ *body od* «λ*s*. *True*» *s*
by(*auto*)
qed
thus *?thesis* **by**(*simp add*:*embed-bool-def*)
qed

4.11.3 Probabilistic Termination

Any loop that has a non-zero chance of terminating after each step terminates with probability 1.

lemma *termination-0-1*:
fixes *body* :: *'s prog*
assumes *wb*: *well-def body*
— The loop terminates in one step with nonzero probability
and *onestep*: (λ*s*. *p*) ⊩ *wp body* «𝒩 *G*»
and *nzp*: *0* < *p*
— The body is maximal i.e. it terminates absolutely.
and *mb*: *maximal* (*wp body*)
shows λ*s*. *1* ⊩ *wp do G* ⟶ *body od* (λ*s*. *1*)
proof −
note *hb* = *well-def-wp-healthy*[*OF wb*]
note *sb* = *healthy-scalingD*[*OF hb*]
note *sab* = *sublinear-subadd*[*OF well-def-wp-sublinear*, *OF wb*, *OF healthy-feasibleD*, *OF hb*]

from *hb* **have** *hloop*: *healthy* (*wp do G* $\longrightarrow$ *body od*)
  by(*rule healthy-intros*)
hence *swp*: *sound* (*wp do G* $\longrightarrow$ *body od* ($\lambda s.\ 1$)) **by**(*blast*)

p is no greater than 1, by feasibility.

from *onestep* **have** *onestep′*: $\bigwedge s.\ p \leq$ *wp body* «$\mathcal{N}$ *G*» *s* **by**(*auto*)
also {
  from *hb* **have** *unitary* (*wp body* «$\mathcal{N}$ *G*») **by**(*auto*)
  hence $\bigwedge s.$ *wp body* «$\mathcal{N}$ *G*» $s \leq 1$ **by**(*auto*)
}
finally have *p1*: $p \leq 1$ **.**

This is the crux of the proof: that given a lower bound below 1, we can find another, higher one.

have *new-bound*: $\bigwedge k.\ 0 \leq k \Longrightarrow k \leq 1 \Longrightarrow (\lambda s.\ k) \Vdash$ *wp do G* $\longrightarrow$ *body od* ($\lambda s.\ 1$) $\Longrightarrow$
    ($\lambda s.\ p * (1-k) + k$) $\Vdash$ *wp do G* $\longrightarrow$ *body od* ($\lambda s.\ 1$)
proof(*rule le-funI*)
  fix *k s*
  assume *X*: $\lambda s.\ k \Vdash$ *wp do G* $\longrightarrow$ *body od* ($\lambda s.\ 1$)
    and *k0*: $0 \leq k$ **and** *k1*: $k \leq 1$

  from *k1* **have** *nz1k*: $0 \leq 1 - k$ **by**(*auto*)
  with *p1* **have** $p * (1-k) + k \leq 1 * (1-k) + k$
   by(*blast intro*:*mult-right-mono add-mono*)
  hence $p * (1 - k) + k \leq 1$
   by(*simp*)

The new bound is $p * (1 - k) + k$.

  hence $p * (1-k) + k \leq$ «$\mathcal{N}$ *G*» *s* + «*G*» *s* $*$ ($p * (1-k) + k$)
   by(*cases G s*, *simp-all*)

By the one-step termination assumption:

  also from *onestep′ nz1k*
  have ... $\leq$ «$\mathcal{N}$ *G*» *s* + «*G*» *s* $*$ (*wp body* «$\mathcal{N}$ *G*» *s* $* (1-k) + k$)
   by (*simp add*: *mult-right-mono ordered-comm-semiring-class.comm-mult-left-mono*)

By scaling:

  also from *nz1k*
  have ... = «$\mathcal{N}$ *G*» *s* + «*G*» *s* $*$ (*wp body* ($\lambda s.$ «$\mathcal{N}$ *G*» *s* $* (1-k)$) *s* + *k*)
   by(*simp add*:*right-scalingD*[*OF sb*])

By the maximality (termination) of the loop body:

  also from *mb k0*
  have ... = «$\mathcal{N}$ *G*» *s* + «*G*» *s* $*$ (*wp body* ($\lambda s.$ «$\mathcal{N}$ *G*» *s* $* (1-k)$) *s* + *wp body* ($\lambda s.\ k$) *s*)
   by(*simp add*:*maximalD*)

By sub-additivity of the loop body:

  also from *k0 nz1k*

have ... $\leq$ «$\mathcal{N}$ G» s + «G» s ∗ (wp body (λs. «$\mathcal{N}$ G» s ∗ (1−k) + k) s)
 by(auto intro!:add-left-mono mult-left-mono sub-addD[OF sab] sound-intros)
also
have ... = «$\mathcal{N}$ G» s + «G» s ∗ (wp body (λs. «$\mathcal{N}$ G» s + «G» s ∗ k) s)
 by(simp add:negate-embed algebra-simps)

By monotonicity of the loop body, and that *k* is a lower bound:

also from k0 hloop le-funD[OF X]
have ... $\leq$ «$\mathcal{N}$ G» s +
 «G» s ∗ (wp body (λs. «$\mathcal{N}$ G» s + «G» s ∗ wp do G ⟶ body od (λs. 1) s) s)
 by(iprover intro:add-left-mono mult-left-mono le-funI embed-ge-0
 le-funD[OF mono-transD, OF healthy-monoD, OF hb]
 sound-sum standard-sound sound-intros swp)

Unrolling the loop once and simplifying:

also {
 have ⋀s. «$\mathcal{N}$ G» s + «G» s ∗ wp body (wp do G ⟶ body od (λs. 1)) s =
 «$\mathcal{N}$ G» s + «G» s ∗ («$\mathcal{N}$ G» s + «G» s ∗ wp body (wp do G ⟶ body od (λs. 1)) s)
 by(simp only:distrib-left mult.assoc[symmetric] embed-bool-idem embed-bool-cancel)
 also have ⋀s. ... s = «$\mathcal{N}$ G» s + «G» s ∗ wp do G ⟶ body od (λs. 1) s
 by(simp add:fun-cong[OF wp-loop-unfold[symmetric, **where** P=λs. 1, simplified, OF hb]])
 finally have X: ⋀s. «$\mathcal{N}$ G» s + «G» s ∗ wp body (wp do G ⟶ body od (λs. 1)) s =
 «$\mathcal{N}$ G» s + «G» s ∗ wp do G ⟶ body od (λs. 1) s **.**
 have «$\mathcal{N}$ G» s + «G» s ∗ (wp body (λs. «$\mathcal{N}$ G» s + «G» s ∗
 wp do G ⟶ body od (λs. 1) s) s) =
 «$\mathcal{N}$ G» s + «G» s ∗ (wp body (λs. «$\mathcal{N}$ G» s + «G» s ∗
 wp body (wp do G ⟶ body od (λs. 1)) s) s)
 by(simp only:X)
}

Lastly, by folding two loop iterations:

also
have «$\mathcal{N}$ G» s + «G» s ∗ (wp body (λs. «$\mathcal{N}$ G» s + «G» s ∗
 wp body (wp do G ⟶ body od (λs. 1)) s) s) =
 wp do G ⟶ body od (λs. 1) s
 by(simp add:wp-loop-unfold[OF - hb, **where** P=λs. 1, simplified, symmetric]
 fun-cong[OF wp-loop-unfold[OF - hb, **where** P=λs. 1, simplified, symmetric]])
finally show p ∗ (1−k) + k $\leq$ wp do G ⟶ body od (λs. 1) s **.**
qed

If the previous bound lay in $[0,1)$, the new bound is strictly greater. This is where we appeal to the fact that *p* is nonzero.

from nzp **have** inc: ⋀k. 0 $\leq$ k ⟹ k < 1 ⟹ k < p ∗ (1 − k) + k
 by(auto intro:mult-pos-pos)

The result follows by contradiction.

show ?thesis
proof(rule ccontr)

If the loop does not terminate everywhere, then there must exist some state from which the probability of termination is strictly less than one.

assume $\neg$ *?thesis*
hence $\neg$ $(\forall s.\ 1 \le wp\ do\ G \longrightarrow body\ od\ (\lambda s.\ 1)\ s)$ **by**(*auto*)
then obtain *s* **where** *point*: $\neg\ 1 \le wp\ do\ G \longrightarrow body\ od\ (\lambda s.\ 1)\ s$ **by**(*auto*)

let $?k = Inf\ (range\ (wp\ do\ G \longrightarrow body\ od\ (\lambda s.\ 1)))$

from *hloop*
have *Inflb*: $\bigwedge s.\ ?k \le wp\ do\ G \longrightarrow body\ od\ (\lambda s.\ 1)\ s$
by(*intro cInf-lower bdd-belowI*, *auto*)
also from *point* **have** $wp\ do\ G \longrightarrow body\ od\ (\lambda s.\ 1)\ s < 1$ **by**(*auto*)

Thus the least (infimum) probabilty of termination is strictly less than one.

finally have *k1*: $?k < 1$ **.**
hence $?k \le 1$ **by**(*auto*)
moreover from *hloop* **have** *k0*: $0 \le ?k$
by(*intro cInf-greatest*, *auto*)

The infimum is, naturally, a lower bound.

moreover from *Inflb* **have** $(\lambda s.\ ?k) \Vdash wp\ do\ G \longrightarrow body\ od\ (\lambda s.\ 1)$ **by**(*auto*)
ultimately

We can therefore use the previous result to find a new bound, . . .

have $\bigwedge s.\ p * (1 - ?k) + ?k \le wp\ do\ G \longrightarrow body\ od\ (\lambda s.\ 1)\ s$
by(*blast intro*:*le-funD*[*OF new-bound*])

. . . which is lower than the infimum, by minimality, . . .

hence $p * (1 - ?k) + ?k \le ?k$
by(*blast intro*:*cInf-greatest*)

. . . yet also strictly greater than it.

moreover from *k0 k1* **have** $?k < p * (1 - ?k) + ?k$ **by**(*rule inc*)

We thus have a contradiction.

ultimately show *False* **by**(*simp*)
qed
qed

end

4.12 Automated Reasoning

theory *Automation* **imports** *StructuredReasoning*
begin

This theory serves as a container for automated reasoning tactics for pGCL, implemented in ML. At present, there is a basic verification condition generator (VCG).

named-theorems *wd*
theorems to automatically establish well−definedness
named-theorems *pwp-core*
core probabilistic wp rules, *for evaluating primitive terms*
named-theorems *pwp*
user−supplied probabilistic wp rules
named-theorems *pwlp*
user−supplied probabilistic wlp rules

ML-file ‹*pVCG.ML*›

method-setup *pvcg* =
‹*Scan.succeed* (*fn ctxt* => *SIMPLE-METHOD′* (*pVCG.pVCG-tac ctxt*))›
Probabilistic weakest preexpectation tactic

declare *wd-intros*[*wd*]

lemmas *core-wp-rules* =
wp-Skip *wlp-Skip*
wp-Abort *wlp-Abort*
wp-Apply *wlp-Apply*
wp-Seq *wlp-Seq*
wp-DC-split *wlp-DC-split*
wp-PC-fixed *wlp-PC-fixed*
wp-SetDC *wlp-SetDC*
wp-SetPC-split *wlp-SetPC-split*

declare *core-wp-rules*[*pwp-core*]

end

Additional Material

4.13 Miscellaneous Mathematics

theory *Misc*
imports
  HOL−Analysis.Multivariate-Analysis
begin lemma *sum-UNIV*:
  fixes *S*::*′a*::*finite set*
  assumes *complete*: $\bigwedge x.\ x \notin S \Longrightarrow f\,x = 0$
  shows *sum f S* = *sum f UNIV*
proof −
  from *complete* **have** *sum f S* = *sum f* (*UNIV* − *S*) + *sum f S* **by**(*simp*)
  also have ... = *sum f UNIV*
    by(*auto intro*: *sum.subset-diff*[*symmetric*])
  finally show *?thesis* **.**
qed

lemma *cInf-mono*:
  fixes *A*::*′a*::*conditionally-complete-lattice set*
  assumes *lower*: $\bigwedge b.\ b \in B \Longrightarrow \exists\, a{\in}A.\ a \leq b$
    and *bounded*: $\bigwedge a.\ a \in A \Longrightarrow c \leq a$
    and *ne*: $B \neq \{\}$
  shows *Inf A* ≤ *Inf B*
proof(*rule cInf-greatest*[*OF ne*])
  fix *b* **assume** *bin*: $b \in B$
  with *lower* **obtain** *a* **where** *ain*: $a \in A$ **and** *le*: $a \leq b$ **by**(*auto*)
  from *ain bounded* **have** *Inf A* ≤ *a* **by**(*intro cInf-lower bdd-belowI*, *auto*)
  also note *le*
  finally show *Inf A* ≤ *b* **.**
qed

lemma *max-distrib*:
  fixes *c*::*real*
  assumes *nn*: $0 \leq c$
  shows $c * max\ a\ b = max\ (c * a)\ (c * b)$
proof(*cases* $a \leq b$)
  case *True*
  moreover with *nn* **have** $c * a \leq c * b$ **by**(*auto intro*:*mult-left-mono*)
  ultimately show *?thesis* **by**(*simp add*:*max.absorb2*)

```
next
  case False then have b ≤ a by(auto)
  moreover with nn have c ∗ b ≤ c ∗ a by(auto intro:mult-left-mono)
  ultimately show ?thesis by(simp add:max.absorb1)
qed

lemma mult-div-mono-left:
  fixes c::real
  assumes nnc: 0 ≤ c and nzc: c ≠ 0
    and inv: a ≤ inverse c ∗ b
  shows c ∗ a ≤ b
proof −
  from nnc inv have c ∗ a ≤ (c ∗ inverse c) ∗ b
   by(auto simp:mult.assoc intro:mult-left-mono)
  also from nzc have ... = b by(simp)
  finally show c ∗ a ≤ b .
qed

lemma mult-div-mono-right:
  fixes c::real
  assumes nnc: 0 ≤ c and nzc: c ≠ 0
    and inv: inverse c ∗ a ≤ b
  shows a ≤ c ∗ b
proof −
  from nzc have a = (c ∗ inverse c) ∗ a  by(simp)
  also from nnc inv have (c ∗ inverse c) ∗ a ≤ c ∗ b
   by(auto simp:mult.assoc intro:mult-left-mono)
  finally show a ≤ c ∗ b .
qed

lemma min-distrib:
  fixes c::real
  assumes nnc: 0 ≤ c
  shows c ∗ min a b = min (c ∗ a) (c ∗ b)
proof(cases a ≤ b)
  case True moreover with nnc have c ∗ a ≤ c ∗ b
   by(blast intro:mult-left-mono)
  ultimately show ?thesis by(auto)
next
  case False hence b ≤ a by(auto)
  moreover with nnc have c ∗ b ≤ c ∗ a
   by(blast intro:mult-left-mono)
  ultimately show ?thesis by(simp add:min.absorb2)
qed

lemma finite-set-least:
  fixes S::'a::linorder set
  assumes finite: finite S
    and ne: S ≠ {}
```

```
  shows ∃x∈S. ∀y∈S. x ≤ y
proof −
  have S = {} ∨ (∃x∈S. ∀y∈S. x ≤ y)
  proof(rule finite-induct, simp-all add:assms)
    fix x::'a and S::'a set
    assume IH: S = {} ∨ (∃x∈S. ∀y∈S. x ≤ y)
    show (∀y∈S. x ≤ y) ∨ (∃x'∈S. x' ≤ x ∧ (∀y∈S. x' ≤ y))
    proof(cases S={})
      case True then show ?thesis by(auto)
    next
      case False with IH have ∃x∈S. ∀y∈S. x ≤ y by(auto)
      then obtain z where zin: z ∈ S and zmin: ∀y∈S. z ≤ y by(auto)
      thus ?thesis by(cases z ≤ x, auto)
    qed
  qed
  with ne show ?thesis by(auto)
qed

lemma cSup-add:
  fixes c::real
  assumes ne: S ≠ {}
      and bS: ⋀x. x∈S ⟹ x ≤ b
  shows Sup S + c = Sup {x + c |x. x ∈ S}
proof(rule antisym)
  from ne bS show Sup {x + c |x. x ∈ S} ≤ Sup S + c
    by(auto intro!:cSup-least add-right-mono cSup-upper bdd-aboveI)

  have Sup S ≤ Sup {x + c |x. x ∈ S} − c
  proof(intro cSup-least ne)
    fix x assume xin: x ∈ S
    from bS have ⋀x. x∈S ⟹ x + c ≤ b + c by(auto intro:add-right-mono)
    hence bdd-above {x + c |x. x ∈ S} by(intro bdd-aboveI, blast)
    with xin have x + c ≤ Sup {x + c |x. x ∈ S} by(auto intro:cSup-upper)
    thus x ≤ Sup {x + c |x. x ∈ S} − c by(auto)
  qed
  thus Sup S + c ≤ Sup {x + c |x. x ∈ S} by(auto)
qed

lemma cSup-mult:
  fixes c::real
  assumes ne: S ≠ {}
      and bS: ⋀x. x∈S ⟹ x ≤ b
      and nnc: 0 ≤ c
  shows c ∗ Sup S = Sup {c ∗ x |x. x ∈ S}
proof(cases)
  assume c = 0
  moreover from ne have ∃x. x ∈ S by(auto)
  ultimately show ?thesis by(simp)
next
```

```
  assume cnz: c ≠ 0
  show ?thesis
  proof(rule antisym)
    from bS have baS: bdd-above S by(intro bdd-aboveI, auto)
    with ne nnc show Sup {c ∗ x |x. x ∈ S} ≤ c ∗ Sup S
      by(blast intro!:cSup-least mult-left-mono[OF cSup-upper])
    have Sup S ≤ inverse c ∗ Sup {c ∗ x |x. x ∈ S}
    proof(intro cSup-least ne)
      fix x assume xin: x∈S
      moreover from bS nnc have ⋀x. x∈S ⟹ c ∗ x ≤ c ∗ b by(auto intro:mult-left-mono)
      ultimately have c ∗ x ≤ Sup {c ∗ x |x. x ∈ S}
        by(auto intro!:cSup-upper bdd-aboveI)
      moreover from nnc have 0 ≤ inverse c by(auto)
      ultimately have inverse c ∗ (c ∗ x) ≤ inverse c ∗ Sup {c ∗ x |x. x ∈ S}
        by(auto intro:mult-left-mono)
      with cnz show x ≤ inverse c ∗ Sup {c ∗ x |x. x ∈ S}
        by(simp add:mult.assoc[symmetric])
    qed
    with nnc have c ∗ Sup S ≤ c ∗ (inverse c ∗ Sup {c ∗ x |x. x ∈ S})
      by(auto intro:mult-left-mono)
    with cnz show c ∗ Sup S ≤ Sup {c ∗ x |x. x ∈ S}
      by(simp add:mult.assoc[symmetric])
  qed
qed

lemma closure-contains-Sup:
  fixes S :: real set
  assumes neS: S ≠ {} and bS: ∀ x∈S. x ≤ B
  shows Sup S ∈ closure S
proof −
  let ?T = uminus ‘ S
  from neS have neT: ?T ≠ {} by(auto)
  from bS have bT: ∀ x∈?T. −B ≤ x by(auto)
  hence bbT: bdd-below ?T by(intro bdd-belowI, blast)

  have Sup S = − Inf ?T
  proof(rule antisym)
    from neT bbT
    have ⋀x. x∈S ⟹ Inf (uminus ‘ S) ≤ −x
      by(blast intro:cInf-lower)
    hence ⋀x. x∈S ⟹ −1 ∗ −x ≤ −1 ∗ Inf (uminus ‘ S)
      by(rule mult-left-mono-neg, auto)
    hence lenInf: ⋀x. x∈S ⟹ x ≤ − Inf (uminus ‘ S)
      by(simp)
    with neS bS show Sup S ≤ − Inf ?T
      by(blast intro:cSup-least)

    have − Sup S ≤ Inf ?T
    proof(rule cInf-greatest[OF neT])
```

```
      fix x assume x ∈ uminus ‘ S
      then obtain y where yin: y ∈ S and rwx: x = −y by(auto)
      from yin bS have y ≤ Sup S
        by(intro cSup-upper bdd-belowI, auto)
      hence −1 ∗ Sup S ≤ −1 ∗ y
        by(simp add:mult-left-mono-neg)
      with rwx show − Sup S ≤ x by(simp)
    qed
    hence −1 ∗ Inf ?T ≤ −1 ∗ (− Sup S)
      by(simp add:mult-left-mono-neg)
    thus − Inf ?T ≤ Sup S by(simp)
  qed
  also {
    from neT bbT have Inf ?T ∈ closure ?T by(rule closure-contains-Inf)
    hence − Inf ?T ∈ uminus ‘ closure ?T by(auto)
  }
  also {
    have linear uminus by(auto intro:linearI)
    hence uminus ‘ closure ?T ⊆ closure (uminus ‘ ?T)
      by(rule closure-linear-image-subset)
  }
  also {
    have uminus ‘ ?T ⊆ S by(auto)
    hence closure (uminus ‘ ?T) ⊆ closure S by(rule closure-mono)
  }
  finally show Sup S ∈ closure S .
qed

lemma tendsto-min:
  fixes x y::real
  assumes ta: a ⟶ x
    and tb: b ⟶ y
  shows (λi. min (a i) (b i)) ⟶ min x y
proof(rule LIMSEQ-I, simp)
  fix e::real assume pe: 0 < e

  from ta pe obtain noa where balla: ∀ n≥noa. abs (a n − x) < e
    by(auto dest:LIMSEQ-D)
  from tb pe obtain nob where ballb: ∀ n≥nob. abs (b n − y) < e
    by(auto dest:LIMSEQ-D)

  {
    fix n
    assume ge: max noa nob ≤ n
    hence gea: noa ≤ n and geb: nob ≤ n by(auto)
    have abs (min (a n) (b n) − min x y) < e
    proof cases
      assume le: min (a n) (b n) ≤ min x y
      show ?thesis
```

```
   proof cases
     assume a n ≤ b n
     hence rwmin: min (a n) (b n) = a n by(auto)
     with le have a n ≤ min x y by(simp)
     moreover from gea balla have abs (a n − x) < e by(simp)
     moreover have min x y ≤ x by(auto)
     ultimately have abs (a n − min x y) < e by(auto)
     with rwmin show abs (min (a n) (b n) − min x y) < e by(simp)
   next
     assume ¬ a n ≤ b n
     hence b n ≤ a n by(auto)
     hence rwmin: min (a n) (b n) = b n by(auto)
     with le have b n ≤ min x y by(simp)
     moreover from geb ballb have abs (b n − y) < e by(simp)
     moreover have min x y ≤ y by(auto)
     ultimately have abs (b n − min x y) < e by(auto)
     with rwmin show abs (min (a n) (b n) − min x y) < e by(simp)
   qed
  next
   assume ¬ min (a n) (b n) ≤ min x y
   hence le: min x y ≤ min (a n) (b n) by(auto)
   show ?thesis
   proof cases
     assume x ≤ y
     hence rwmin: min x y = x by(auto)
     with le have x ≤ min (a n) (b n) by(simp)
     moreover from gea balla have abs (a n − x) < e by(simp)
     moreover have min (a n) (b n) ≤ a n by(auto)
     ultimately have abs (min (a n) (b n) − x) < e by(auto)
     with rwmin show abs (min (a n) (b n) − min x y) < e by(simp)
   next
     assume ¬ x ≤ y
     hence y ≤ x by(auto)
     hence rwmin: min x y = y by(auto)
     with le have y ≤ min (a n) (b n) by(simp)
     moreover from geb ballb have abs (b n − y) < e by(simp)
     moreover have min (a n) (b n) ≤ b n by(auto)
     ultimately have abs (min (a n) (b n) − y) < e by(auto)
     with rwmin show abs (min (a n) (b n) − min x y) < e by(simp)
   qed
  qed
 }
 thus ∃ no. ∀ n≥no. |min (a n) (b n) − min x y| < e by(blast)
qed

definition supp :: ('s ⇒ real) ⇒ 's set
where supp f = {x. f x ≠ 0}

definition dist-remove :: ('s ⇒ real) ⇒ 's ⇒ 's ⇒ real
```

where *dist-remove p x* = (λy. *if y=x then 0 else p y* / (*1* − *p x*))

lemma *supp-dist-remove*:
p x ≠ *0* ⟹ *p x* ≠ *1* ⟹ *supp* (*dist-remove p x*) = *supp p* − {*x*}
by(*auto simp*:*dist-remove-def supp-def*)

lemma *supp-empty*:
supp f = {} ⟹ *f x* = *0*
by(*simp add*:*supp-def*)

lemma *nsupp-zero*:
x ∉ *supp f* ⟹ *f x* = *0*
by(*simp add*:*supp-def*)

lemma *sum-supp*:
fixes *f*::′*a*::*finite* ⇒ *real*
shows *sum f* (*supp f*) = *sum f UNIV*
proof −
have *sum f* (*UNIV* − *supp f*) = *0*
by(*simp add*:*supp-def*)
hence *sum f* (*supp f*) = *sum f* (*UNIV* − *supp f*) + *sum f* (*supp f*)
by(*simp*)
also have ... = *sum f UNIV*
by(*simp add*:*sum.subset-diff*[*symmetric*])
finally show *?thesis* **.**
qed

4.13.1 Truncated Subtraction

definition
tminus :: *real* ⇒ *real* ⇒ *real* (**infixl** ‹⊖› *60*)
where
x ⊖ *y* = *max* (*x* − *y*) *0*

lemma *minus-le-tminus*[*intro*!,*simp*]:
a − *b* ≤ *a* ⊖ *b*
unfolding *tminus-def* **by**(*auto*)

lemma *tminus-cancel-1*:
0 ≤ *a* ⟹ *a* + *1* ⊖ *1* = *a*
unfolding *tminus-def* **by**(*simp*)

lemma *tminus-zero-imp-le*:
x ⊖ *y* ≤ *0* ⟹ *x* ≤ *y*
by(*simp add*:*tminus-def*)

lemma *tminus-zero*[*simp*]:
0 ≤ *x* ⟹ *x* ⊖ *0* = *x*
by(*simp add*:*tminus-def*)

```
lemma tminus-left-mono:
  a ≤ b ⟹ a ⊖ c ≤ b ⊖ c
  unfolding tminus-def
  by(case-tac a ≤ c, simp-all)

lemma tminus-less:
  ⟦ 0 ≤ a; 0 ≤ b ⟧ ⟹ a ⊖ b ≤ a
  unfolding tminus-def by(force)

lemma tminus-left-distrib:
  assumes nna: 0 ≤ a
  shows a ∗ (b ⊖ c) = a ∗ b ⊖ a ∗ c
proof(cases b ≤ c)
  case True note le = this
  hence a ∗ max (b − c) 0 = 0 by(simp add:max.absorb2)
  also {
    from nna le have a ∗ b ≤ a ∗ c by(blast intro:mult-left-mono)
    hence 0 = max (a ∗ b − a ∗ c) 0 by(simp add:max.absorb1)
  }
  finally show ?thesis by(simp add:tminus-def)
next
  case False hence le: c ≤ b by(auto)
  hence a ∗ max (b − c) 0 = a ∗ (b − c) by(simp only:max.absorb1)
  also {
    from nna le have a ∗ c ≤ a ∗ b by(blast intro:mult-left-mono)
    hence a ∗ (b − c) = max (a ∗ b − a ∗ c) 0 by(simp add:max.absorb1 field-simps)
  }
  finally show ?thesis by(simp add:tminus-def)
qed

lemma tminus-le[simp]:
  b ≤ a ⟹ a ⊖ b = a − b
  unfolding tminus-def by(simp)

lemma tminus-le-alt[simp]:
  a ≤ b ⟹ a ⊖ b = 0
  by(simp add:tminus-def)

lemma tminus-nle[simp]:
  ¬b ≤ a ⟹ a ⊖ b = 0
  unfolding tminus-def by(simp)

lemma tminus-add-mono:
  (a+b) ⊖ (c+d) ≤ (a⊖c) + (b⊖d)
proof(cases 0 ≤ a − c)
  case True note pac = this
  show ?thesis
  proof(cases 0 ≤ b − d)
```

case *True* **note** *pbd* = *this*
from *pac* **and** *pbd* **have** $(c + d) \le (a + b)$ **by**(*simp*)
with *pac* **and** *pbd* **show** *?thesis* **by**(*simp*)
next
case *False* **with** *pac* **show** *?thesis*
by(*cases* $c + d \le a + b$, *auto*)
qed
next
case *False* **note** *nac* = *this*
show *?thesis*
proof(*cases* $0 \le b - d$)
case *True* **with** *nac* **show** *?thesis*
by(*cases* $c + d \le a + b$, *auto*)
next
case *False* **note** *nbd* = *this*
with *nac* **have** $\neg(c + d) \le (a + b)$ **by**(*simp*)
with *nac* **and** *nbd* **show** *?thesis* **by**(*simp*)
qed
qed

lemma *tminus-sum-mono*:
assumes *fS*: *finite S*
shows $sum\ f\ S \ominus sum\ g\ S \le sum\ (\lambda x.\ f\ x \ominus g\ x)\ S$
(**is** *?X S*)
proof(*rule finite-induct*)
from *fS* **show** *finite S* **.**

show *?X* {} **by**(*simp*)

fix *x* **and** *F*
assume *fF*: *finite F* **and** *xniF*: $x \notin F$
and *IH*: *?X F*
have $f\ x + sum\ f\ F \ominus g\ x + sum\ g\ F \le$
$(f\ x \ominus g\ x) + (sum\ f\ F \ominus sum\ g\ F)$
by(*rule tminus-add-mono*)
also from *IH* **have** $\ldots \le (f\ x \ominus g\ x) + (\sum x{\in}F.\ f\ x \ominus g\ x)$
by(*rule add-left-mono*)
finally show *?X* (*insert x F*)
by(*simp add*:*sum.insert*[*OF fF xniF*])
qed

lemma *tminus-nneg*[*simp*,*intro*]:
$0 \le a \ominus b$
by(*cases* $b \le a$, *auto*)

lemma *tminus-right-antimono*:
assumes *clb*: $c \le b$
shows $a \ominus b \le a \ominus c$
proof(*cases* $b \le a$)

case *True*
moreover with *clb* **have** $c \leq a$ **by**(*auto*)
moreover note *clb*
ultimately show *?thesis* **by**(*simp*)
next
case *False* **then show** *?thesis* **by**(*simp*)
qed

lemma *min-tminus-distrib*:
min a b $\ominus$ *c* = *min* (*a* $\ominus$ *c*) (*b* $\ominus$ *c*)
unfolding *tminus-def* **by**(*auto*)

end

Bibliography

David Cock. Verifying probabilistic correctness in Isabelle with pGCL. In *Proceedings of the 7th Systems Software Verification*, pages 1–10, Sydney, Australia, November 2012. doi: 10.4204/EPTCS.102.15.

David Cock. Practical probability: Applying pGCL to lattice scheduling. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, pages 1–16, Rennes, France, July 2013. doi: 10.1007/978-3-642-39634-2_23.

David Cock. From probabilistic operational semantics to information theory - side channels with pGCL in isabelle. In *Proceedings of the 5th International Conference on Interactive Theorem Proving*, pages 1–15, Vienna, Austria, July 2014a. Springer.

David Cock. *Leakage in Trustworthy Systems*. PhD thesis, University of New South Wales, 2014b.

Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975. ISSN 0001-0782. doi: 10.1145/360933.360975.

Joe Hurd, Annabelle McIver, and Carroll Morgan. Probabilistic guarded commands mechanized in hol. *Theoretical Computer Science*, 346(1):96 – 112, 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2005.08.005. URL http://www.sciencedirect.com/science/article/pii/S0304397505004767.

Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer, 2004.

Steve Selvin. A problem in probability (letter to the editor). *American Statistician*, 29(1):67, Feb 1975.