pGCL for Isabelle

David Cock

May 28, 2015
# 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 ............................. 25
  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 ......................... 36
  3.2.2 Healthy Transformers ............................ 39
  3.2.3 Sublinearity .................................... 47
3.2.4 Determinism ........................................ 57
3.2.5 Modular Reasoning ................................. 60
3.2.6 Transforming Standard Expectations ............ 61
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 ................................. 72

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 ............... 126
    4.4.1 The Limit of Iterates ....................... 134
  4.5 Sublinearity ....................................... 144
    4.5.1 Nonrecursive Primitives ..................... 144
    4.5.2 Sublinearity for Loops ..................... 150
  4.6 Determinism ....................................... 154
    4.6.1 Additivity .................................. 154
    4.6.2 Maximaly .................................. 155
    4.6.3 Determinism ................................ 157
  4.7 Well-Defined Programs. ............................. 158
    4.7.1 Strict Implies Liberal ........................ 158
    4.7.2 Sub-Distributivity of Conjunction ........... 162
    4.7.3 The Well-Defined Predicate ................. 172
  4.8 The Loop Rules .................................... 175
    4.8.1 Liberal and Strict Invariants ............... 175
    4.8.2 Partial Correctness .......................... 176
    4.8.3 Total Correctness ............................ 176
    4.8.4 Unfolding ................................... 177
  4.9 The Algebra of pGCL ............................... 178
    4.9.1 Program Refinement ........................... 178
    4.9.2 Simple Identities ............................ 179
    4.9.3 Deterministic Programs are Maximal .......... 184
    4.9.4 The Algebraic Structure of Refinement ...... 186
    4.9.5 Data Refinement .............................. 189
    4.9.6 The Algebra of Data Refinement ............... 193
    4.9.7 Structural Rules for Correspondence .......... 200
### CONTENTS

- **4.9.8 Structural Rules for Data Refinement** . . . . . . . . . 201
- **4.10 Structured Reasoning** . . . . . . . . . . . . . . . . . . . 203
  - **4.10.1 Syntactic Decomposition** . . . . . . . . . . . . . . . 204
  - **4.10.2 Algebraic Decomposition** . . . . . . . . . . . . . . 209
  - **4.10.3 Hoare triples** . . . . . . . . . . . . . . . . . . . . 209
- **4.11 Loop Termination** . . . . . . . . . . . . . . . . . . . . . 209
  - **4.11.1 Trivial Termination** . . . . . . . . . . . . . . . . . 210
  - **4.11.2 Classical Termination** . . . . . . . . . . . . . . . . 210
  - **4.11.3 Probabilistic Termination** . . . . . . . . . . . . . . 213
- **4.12 Automated Reasoning** . . . . . . . . . . . . . . . . . . . 217
- **Additional Material** . . . . . . . . . . . . . . . . . . . . . 219
  - **4.13 Miscellaneous Mathematics** . . . . . . . . . . . . . . 219
    - **4.13.1 Truncated Subtraction** . . . . . . . . . . . . . . . 225
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 (λs. s (| a := Heads |)) = (a := (λs. 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 (λs. 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 $P::s \Rightarrow \text{bool}$ is embedded as $« P »::s \Rightarrow \text{real}$, such that $P \, s \rightarrow « P » \, s = 1 \land \lnot P \, s \rightarrow « P » \, s = 0$.

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 $\forall s. P \, s \rightarrow Q \, s$ is substituted with the equivalent expectation entailment $« P » \vdash « Q »$, $[ \langle \langle « ?P » \vdash « ?Q »; ?P \, ?s \rangle \Rightarrow ?Q \, ?s ]$. Thus a valid specification of $\text{Apply} (\lambda s. s((a := b \, s)))$ is:

**lemma**

$\forall x. \langle \lambda s. b \, s = x \rangle \vdash \text{wp} (a := b) \langle \lambda s. a \, s = x \rangle$

**by**($\text{pvcg, simp add:o-def}$)

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**

$\langle \lambda s::\text{real}. \, s \neq 0 \rangle \vdash \text{wp} (\text{Apply} (op \, * \, 2) \, ;; \, \text{Apply} (\lambda s. s \, / \, s)) \langle \lambda s. \, s = 1 \rangle$

**by**($\text{pvcg, simp add:o-def}$)

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 $(1::'a) - p$. The probability may depend on the state, and is therefore of
2.1. LANGUAGE PRIMITIVES

The following annotation states that the probability of heads is exactly 1/2:

**Definition**

\[ \text{flip-a} :: \text{real} \Rightarrow \text{coins prog} \]

**Where**

\[ \text{flip-a} \ p = a := (\lambda. \text{Heads}) (\lambda s. p) \oplus a := (\lambda. \text{Tails}) \]

**Lemma**

\( (\lambda s. 1/2) = \wp (\text{flip-a} (1/2)) \ «\lambda s. a = \text{Heads} » \)

**Unfolding** flip-a-def

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

by \( (\text{simp add:wp-eval o-def}) \)

### 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**

\( \lambda s. 1/3 \vdash \wp (\text{flip-a} (2/3) \sqcap \text{flip-a} (1/3)) \ «\lambda s. a = \text{Heads} » \)

**Unfolding** flip-a-def

by \( (\text{pvcg, simp add:o-def le-funI}) \)

### 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 \( (1 :: 'a) / (4 :: 'a) \).

**Definition**

\[ \text{flip-b} :: \text{real} \Rightarrow \text{coins prog} \]

**Where**

\[ \text{flip-b} \ p = b := (\lambda. \text{Heads}) (\lambda s. p) \oplus b := (\lambda. \text{Tails}) \]

**Lemma**

\( (\lambda s. 1/4) = \wp (\text{flip-a} (1/2) \sqcap \text{flip-b} (1/2)) \ «\lambda s. a = \text{Heads} \wedge b s = \text{Heads} » \)

**Unfolding** flip-a-def flip-b-def

by \( (\text{simp add:wp-eval o-def}) \)

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.
CHAPTER 2. INTRODUCTION TO PGCL

record dice =
  red :: nat
  blue :: nat

definition Puniform :: 'a set ⇒ ('a ⇒ real)
where Puniform S = (λx. 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
  (λs. 7/2) = wp (bind v at (λs. Puniform {1..6} v) in red := (λs. v)) red
  by(simp add:wp-eval supp-Puniform setsum-head-Suc Puniform-in real-eq-of-nat)

The expectations of independent variables add:

lemma
  (λs. 7) = wp ((bind v at (λs. Puniform {1..6} v) in red := (λs. v)) ;;
  (bind v at (λs. Puniform {1..6} v) in blue := (λs. v)))
  (λs. red s + blue s)
  by(simp add:wp-eval supp-Puniform setsum-head-Suc Puniform-in real-eq-of-nat)

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:
2.2. LOOPS

definition countdown :: int prog
where
  countdown = do (λx. 0 < x) −→ Apply (λs. s − 1) od

Clearly, this loop will only terminate from a state where (0::'a) ≤ x. This is, in fact, also a loop invariant.

definition inv-count :: int ⇒ bool
where
  inv-count = (λx. 0 ≤ x)

Read wp-inv G body I as: I is an invariant of the loop µx. body ;; x « G » ⊕ Skip, or « G » & & I ⊢ wp body I.

lemma wp-inv-count:
  wp-inv (λx. 0 < x) (Apply (λs. s − 1)) «inv-count»
unfolding wp-inv-def inv-count-def wp-eval o-def
proof(clarify, cases)
  fix x::int
  assume 0 ≤ x
  then show «λx. 0 < x» x * «λx. 0 ≤ x» x ≤ «λx. 0 ≤ x» (x − 1)
    by(simp add:embed-bool-def)
next
  fix x::int
  assume ¬ 0 ≤ x
  then show «λx. 0 < x» x * «λx. 0 ≤ x» x ≤ «λx. 0 ≤ 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 (λs. 1)
unfolding countdown-def
proof(intro loop-term-nat-measure[where m=λx. nat (max x 0)] wp-inv-count)
  let ?p = Apply (λx. x − 1::int)
As usual, well-definedness is trivial.

  show well-def ?p
    by(rule ud-intros)

  A measure of 0 implies termination.
  show ∀x. nat (max x 0) = 0 −→ ¬ 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 ≤ (θ::'a), 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.
\[ \forall n. \forall x. \text{nat} (\max x \ 0) = \text{Suc} n \] 

\[ \wp \ ?p \ \forall x. \text{nat} (\max x \ 0) = n \]

**show** 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 (\- . Heads) (\s. 1/2) \oplus Apply (\- . 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 (op \neq Heads) \rightarrow flip od`

Nonetheless, we can show termination .

**lemma** `wait-for-heads-term`:

`\lambda s. 1 \vdash wp \ wait-for-heads (\lambda 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.
2.3. THE MONTY HALL PROBLEM

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

show $\lambda s. 1/2 \vdash \text{wp flip \{N \mid (op \neq \text{Heads})\}}$

unfolding flip-def
by(pvcg, simp add:a-def Heads-def Tails-def)

qed

end

2.3 The Monty Hall Problem

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 contestant 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 \Rightarrow bool
where player-wins g \equiv guess g = prize g
Invariants

We prove explicitly that only valid doors are ever chosen.

**definition** \( \text{inv-prize} :: \text{game} \Rightarrow \text{bool} \)

**where** \( \text{inv-prize} g \equiv \text{prize} g \in \{1,2,3\} \)

**definition** \( \text{inv-clue} :: \text{game} \Rightarrow \text{bool} \)

**where** \( \text{inv-clue} g \equiv \text{clue} g \in \{1,2,3\} \)

**definition** \( \text{inv-guess} :: \text{game} \Rightarrow \text{bool} \)

**where** \( \text{inv-guess} g \equiv \text{guess} g \in \{1,2,3\} \)

### 2.3.2 The Game

Hide the prize behind door \( D \).

**definition** \( \text{hide-behind} :: \text{nat} \Rightarrow \text{game prog} \)

**where** \( \text{hide-behind} D \equiv \text{Apply} (\text{prize-update} (\lambda x. D)) \)

Choose door \( D \).

**definition** \( \text{guess-behind} :: \text{nat} \Rightarrow \text{game prog} \)

**where** \( \text{guess-behind} D \equiv \text{Apply} (\text{guess-update} (\lambda x. D)) \)

Open door \( D \) and reveal what’s behind.

**definition** \( \text{open-door} :: \text{nat} \Rightarrow \text{game prog} \)

**where** \( \text{open-door} D \equiv \text{Apply} (\text{clue-update} (\lambda x. D)) \)

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

**definition** \( \text{hide-prize} :: \text{game prog} \)

**where** \( \text{hide-prize} \equiv \text{hide-behind} 1 \cap \text{hide-behind} 2 \cap \text{hide-behind} 3 \)

Guess uniformly at random.

**definition** \( \text{make-guess} :: \text{game prog} \)

**where** \( \text{make-guess} \equiv \text{guess-behind} 1 \oplus (\lambda s. 1/3) \oplus \text{guess-behind} 2 \oplus (\lambda s. 1/2) \oplus \text{guess-behind} 3 \)

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

**definition** \( \text{reveal} :: \text{game prog} \)

**where** \( \text{reveal} \equiv \bigcap d \in (\lambda s. \{1,2,3\} - \{\text{prize} s, \text{guess} s\}). \text{open-door} d \)

Switch your guess to the other unopened door.

**definition** \( \text{switch-guess} :: \text{game prog} \)

**where** \( \text{switch-guess} \equiv \bigcap d \in (\lambda s. \{1,2,3\} - \{\text{clue} s, \text{guess} s\}). \text{guess-behind} d \)

The complete game, either with or without switching guesses.

**definition** \( \text{monty} :: \text{bool} \Rightarrow \text{game prog} \)
where
\[
\text{monty switch} \equiv \text{hide-prize } \;;
\text{make-guess } \;;
\text{reveal } \;;
(\text{if switch then switch-guess else Skip})
\]

### 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]`:
\[
p = g \implies \langle \text{player-wins} \rangle (s\langle\text{prize} := p, \text{guess} := g, \text{clue} := c \rangle) = 1
\]
*by*(simp add:embed-bool-def player-wins-def)

**Lemma** `eval-loss[simp]`:
\[
p \neq g \implies \langle \text{player-wins} \rangle (s\langle\text{prize} := p, \text{guess} := g, \text{clue} := c \rangle) = 0
\]
*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`:
\[
(\lambda s. 1/3) = \wp (\text{monty False}) \langle \text{player-wins} \rangle
\]
*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`:
\[
s\langle\text{prize} := p, \text{clue} := c, \text{guess} := g \rangle =
\]
\[
s\langle\text{prize} := p, \text{guess} := g, \text{clue} := c \rangle
\]
*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`:
\[
(\lambda s. 2/3) = \wp (\text{monty True}) \langle \text{player-wins} \rangle
\]
*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)

### 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.

\textbf{lemma \textit{wd-hide-prize}:}
\begin{itemize}
  \item well-def hide-prize
  \item unfolding hide-prize-def hide-behind-def
  \item by(simp add:wd-intros)
\end{itemize}

\textbf{lemma \textit{wd-make-guess}:}
\begin{itemize}
  \item well-def make-guess
  \item unfolding make-guess-def guess-behind-def
  \item by(simp add:wd-intros)
\end{itemize}

\textbf{lemma \textit{wd-reveal}:}
\begin{itemize}
  \item well-def reveal
\end{itemize}
\textbf{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.

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

\textbf{lemma \textit{wd-switch-guess}:}
\begin{itemize}
  \item well-def switch-guess
\end{itemize}
\textbf{proof –}

\begin{itemize}
  \item have $\forall s. \{1, 2, 3\} - \{\text{clue } s, \text{guess } s\} \neq \{\}$
  \item by(auto simp:insert-Diff-if)
  \item thus ?thesis
  \item unfolding switch-guess-def guess-behind-def
  \item by(intro wd-intros, auto)
\end{itemize}
\textbf{qed}

\textbf{lemmas monty-healthy =}
\begin{itemize}
  \item \textit{wd-switch-guess \textit{wd-reveal \textit{wd-make-guess \textit{wd-hide-prize}}}
\end{itemize}

Annotations

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

hide-prize chooses a valid door.

lemma wp-hide-prize:
\[(\lambda s. 1) \vdash wp \text{ hide-prize } \langle \text{inv-prize} \rangle\]

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:
\[(\lambda s. 2/3 \ast \langle \lambda g. \text{inv-prize } g \rangle s) \vdash wp \text{ make-guess } \langle \lambda g. \text{guess } g \neq \text{prize } g \land \text{inv-prize } g \land \text{inv-guess } g \rangle\]

unfolding make-guess-def guess-behind-def wp-eval o-def
by (auto simp add: embed-bool-def inv-prize-def inv-guess-def)

lemma last-one:
assumes \[a \neq b \text{ and } a \in \{1::nat,2,3\} \text{ and } b \in \{1,2,3\}\]
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:
\[\langle \lambda g. \text{guess } g \neq \text{prize } g \land \text{inv-prize } g \land \text{inv-guess } g \rangle \vdash wp \text{ reveal } \langle \lambda g. \text{guess } g \neq \text{prize } g \land \text{clue } g \neq \text{prize } g \land \text{clue } g \neq \text{guess } g \land \text{inv-prize } g \land \text{inv-guess } g \land \text{inv-clue } g \rangle\]

(is \[?X \vdash wp \text{ reveal } \langle ?Y \rangle\])
proof (rule use-premise, rule well-def-wp-healthy[OF wd-reveal], clarify)
fix \(s\)
assume guess \(s \neq \text{prize } s\)
and inv-prize \(s\)
and inv-guess \(s\)
moreover then obtain \(c\)
where singleton: \(\{\text{Suc } 0,2,3\} - \{\text{prize } s, \text{guess } s\} = \{c\}\)
and \(c \neq \text{prize } s\)
and \(c \neq \text{guess } s\)
and \(c \in \{\text{Suc } 0,2,3\}\)

unfolding inv-prize-def inv-guess-def
by (force dest: last-one elim: ex1E)
ultimately show \(1 \leq wp \text{ reveal } \langle Y \rangle s\)
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.
chapter 2. introduction to pgcl

lemma distinct-game:
\[
\begin{align*}
\text{guess } g &\neq \text{prize } g; \text{clue } g \neq \text{prize } g; \text{clue } g \neq \text{guess } g; \\
\text{inv-prize } g &\land \text{inv-guess } g; \text{inv-clue } g \implies \\
\{1, 2, 3\} &= \{\text{guess } g, \text{prize } g, \text{clue } g\}
\end{align*}
\]

unfolding \text{inv-prize-def} \text{inv-guess-def} \text{inv-clue-def}
apply rule set-eqI
apply rule iffI
apply clarify
apply (metis \text{full-types} \text{empty-iff} \text{insert-iff})
apply (metis \text{insert-iff})
done

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

lemma wp-switch-guess:
\[
\begin{align*}
\forall g. \text{guess } g &\neq \text{prize } g \land \text{clue } g \neq \text{prize } g \land \text{clue } g \neq \text{guess } g \\
\text{inv-prize } g &\land \text{inv-guess } g \land \text{inv-clue } g \implies \\
\wp \text{switch-guess}\{\text{player-wins}\}
\end{align*}
\]

proof (rule use-premise, safe)
from wd-switch-guess show healthy \{\text{wp switch-guess}\} by (auto)

fix s
assume guess s \neq \text{prize } s and clue s \neq \text{prize } s
and clue s \neq \text{guess } s and inv-prize s
and inv-guess s and inv-clue s
note state = this

hence \(1 \leq \text{Inf}\) ((\(\lambda a. \text{player-wins}\) (s\{\text{guess} \leftarrow a\})) \\
\{\text{guess } s, \text{prize } s, \text{clue } s\} - \{\text{clue } s, \text{guess } s\}))
by(auto simp:insert-Diff-if player-wins-def)
also from state have ... = \text{Inf} ((\(\lambda a. \text{player-wins}\) (s\{\text{guess} \leftarrow a\})) \\
\{1, 2, 3\} - \{\text{clue } s, \text{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 \(1 \leq \text{wp switch-guess «player-wins» s}\).

qed

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

lemma wp-monty-switch-modular:
\[
(\lambda s. 2/3) \vdash \wp \text{monty True «player-wins»}
\]

proof (rule wp-validID) — Work in probabilistic Hoare triples
note \text{wp-validI[OF wp-scale, OF wp-hide-prize, simplified]}
— Here we apply scaling to match our pre-expectation
also note \text{wp-validI[OF wp-make-guess]}
also note \text{wp-validI[OF wp-reveal]}
also note \text{wp-validI[OF wp-switch-guess]}
finally show \(\\{\lambda s. 2/3\} \text{monty True «player-wins»}\) \text{p}
  unfolding \text{monty-def}
2.3. THE MONTY HALL PROBLEM

by (simp add: wd-intros sound-intros monty-healthy)
qed

Using the VCG

lemma scaled-hide = wp-scale [OF wp-hide-prize, simplified]

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 ⇒ real

Expectations are a real-valued generalisation of boolean predicates: An expectation on state 's is a function 's ⇒ 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:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a (\rightarrow) b</td>
<td>x</td>
<td>y</td>
<td>x (\leq) y</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>0</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>0</td>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>1</td>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>1</td>
<td>1</td>
<td>T</td>
</tr>
</tbody>
</table>

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.

Figure 3.1: A probabilistic automaton
c is 3.0. The expected value from state $a$ is the weighted sum of these, or $0.7 \times 2.0 + 0.3 \times 3.0 = 2.3$.

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** :: $\text{real} \Rightarrow (\text{'a} \Rightarrow \text{real}) \Rightarrow \text{bool}$

**where**

$$\text{bounded-by } b \ P \equiv \forall x. P \ x \leq b$$

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

**lemma** **bounded-by1[intro]**:

$$[ \forall x. P \ x \leq b ] \Rightarrow \text{bounded-by } b \ P$$

**by** (simp add:bounded-by-def)

**lemma** **bounded-by2[intro]**:

$$P \leq (\lambda s. b) \Rightarrow \text{bounded-by } b \ P$$

**by** (blast dest:le-funD)

**lemma** **bounded-byD[dest]**:

$$\text{bounded-by } b \ P \Rightarrow P \ x \leq b$$

**by** (simp add:bounded-by-def)

**lemma** **bounded-byD2[dest]**:

$$\text{bounded-by } b \ P \Rightarrow P \leq (\lambda s. b)$$

**by** (blast intro:le-funI)

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

**definition** **bounded** :: $(\text{'a} \Rightarrow \text{real}) \Rightarrow \text{bool}$

**where**

$$\text{bounded } P \equiv (\exists b. \text{bounded-by } b \ P)$$

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

**definition** **bound-of** :: $(\text{'a} \Rightarrow \text{real}) \Rightarrow \text{real}$

**where**

$$\text{bound-of } P \equiv \text{Sup} \ (P \ : \ \text{UNIV})$$

**lemma** **bounded-bdd-above[intro]**:

**assumes** $bP$: bounded $P$

**shows** $\text{bdd-above} \ (\text{range } P)$

**proof**

**fix** $x$ assume $x \in \text{range } P$
3.1. EXPECTATIONS

with \( bP \) show \( x \leq \inf \{ b, \text{bounded-by } b \ P \} \)
unfolding bounded-def by (auto intro: cInf-greatest)
qed

The least upper bound has the usual properties:

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

**Lemma bounded-by-bound-of[intro]:**
fixes \( \alpha \Rightarrow \text{real} \)
assumes \( bP: \text{bounded } P \)
shows \( \text{bounded-by } (\text{bound-of } P) \ P \)
unfolding bound-of-def
using \( bP \) by (intro bounded-byI cSup-upper bounded-bdd-above, auto)

**Lemma bound-of-greater[intro]:**
bounded \( P \Rightarrow P x \leq \text{bound-of } P \)
by (blast intro: bounded-byD)

**Lemma bounded-by-mono:**
[ \[ \text{bounded-by } a \ P; a \leq b \] ] \( \Rightarrow \) bounded-by b P
unfolding bounded-by-def by (blast intro: order-trans)

**Lemma bounded-by-imp-bounded[intro]:**
bounded-by b P \( \Rightarrow \) bounded P
unfolding bounded-def by (blast)

This is occasionally easier to apply:

**Lemma bounded-by-bound-of-alt:**
[ \[ \text{bounded } P; \text{bound-of } P = a \] ] \( \Rightarrow \) bounded-by a P
by (blast)

**Lemma bounded-const[simp]:**
bounded \( (\lambda x. c) \)
by (blast)

**Lemma bounded-by-const[intro]:**
c \( \leq b \) \( \Rightarrow \) bounded-by b \( (\lambda x. c) \)
by (blast)

**Lemma bounded-by-mono-alt[intro]:**
[ \[ \text{bounded-by } b Q; P \leq Q \] ] \( \Rightarrow \) bounded-by b P
by (blast intro: order-trans dest: le-funD)

**Lemma bound-of-const[simp, intro]:**
\( \text{bound-of } (\lambda x. c) = (c::\text{real}) \)
unfolding bound-of-def
by (intro antisym cSup-least cSup-upper bounded-bdd-above bounded-const, auto)

lemma bound-of-leI:
assumes \( \forall x. P x \leq (c :: \text{real}) \)
shows bound-of \( P \leq c \)
unfolding bound-of-def
using assms by (intro cSup-least, auto)

lemma bound-of-mono[intro]:
[ \[ P \leq Q; \text{bounded} P; \text{bounded} Q \] \implies \text{bound-of} P \leq \text{bound-of} Q ]
by (blast intro:order-trans dest:le-funD)

lemma bounded-by-o[intro,simp]:
\( \forall b. \text{bounded-by} b P \implies \text{bounded-by} b (P \circ f) \)
unfolding o-def by (blast)

lemma le-bound-of[intro]:
\( \forall x. \text{bounded} f \implies f x \leq \text{bound-of} f \)
by (blast)

3.1.2 Non-Negative Functions.

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

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

lemma nnegI[intro]:
[ \[ \forall x. 0 \leq P x \] \implies \text{nneg} P ]
by (simp add:nneg-def)

lemma nnegI2[intro]:
\( (\lambda s. 0) \leq P \implies \text{nneg} P \)
by (blast dest:le-funD)

lemma nnegD[dest]:
\( \text{nneg} P \implies 0 \leq P x \)
by (simp add:nneg-def)

lemma nnegD2[dest]:
\( \text{nneg} P \implies (\lambda s. 0) \leq P \)
by (blast intro:le-funI)

lemma nneg-bdd-below[intro]:
\( \text{nneg} P \implies \text{bdd-below} (\text{range} P) \)
by (auto)
3.1. EXPECTATIONS

**lemma nneg-const[iff]:**

\( \text{nneg}\ (\lambda x. \ c) \iff 0 \leq c \)

**by (simp add:nneg-def)**

**lemma nneg-o[intro,simp]:**

\( \text{nneg} \ P \implies \text{nneg} \ (P \ o \ f) \)

**by (force)**

**lemma nneg-bound-nneg[intro]:**

\[ \text{bounded} \ P; \ nneg \ P \] \implies 0 \leq \text{bound-of} \ P

**by (blast intro:order-trans)**

**lemma nneg-bounded-by-nneg[dest]:**

\[ \text{bounded-by} \ b \ P; \ nneg \ P \] \implies 0 \leq (b::real)

**by (blast intro:order-trans)**

**lemma bounded-by-nneg[dest]:**

fixes \( \mathsf{P} :: \mathsf{\mathcal{S}} \Rightarrow \mathbb{R} \)

shows \[ \text{bounded-by} \ b \ P; \ nneg \ P \] \implies 0 \leq b

**by (blast intro:order-trans)**

### 3.1.3 Sound Expectations

**definition** sound :: \( (\mathsf{s} \Rightarrow \mathbb{R}) \Rightarrow \mathbb{B} \)

**where** sound \( P \equiv \text{bounded} \ P \land \text{nneg} \ P \)

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

**lemma soundI:**

\[ \text{bounded} \ P; \ nneg \ P \] \implies \text{sound} \ P

**by (simp add:sound-def)**

**lemma soundI2[intro]:**

\[ \text{bounded-by} \ b \ P; \ nneg \ P \] \implies \text{sound} \ P

**by(blast intro:soundI)**

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

\( \text{sound} \ P \implies \text{bounded} \ P \)

**by (simp add:sound-def)**

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

\( \text{sound} \ P \implies \text{nneg} \ P \)

**by (simp add:sound-def)**

**lemma bound-of-sound[intro]:**

assumes \( sP: \text{sound} \ P \)
CHAPTER 3. SEMANTIC STRUCTURES

shows $0 \leq \text{bound-of } P$
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\[simp,intro\] :
asumes \(sP\): sound \(P\) and \(sQ\): sound \(Q\)
shows sound \((\lambda s. P s + Q s)\)
proof
from \(sP\) have \(\forall s. P s \leq \text{bound-of } P\) by (blast)
moreover from \(sQ\) have \(\forall s. Q s \leq \text{bound-of } Q\) by (blast)
ultimately have \(\forall s. P s + Q s \leq \text{bound-of } P + \text{bound-of } Q\)
  using \(sP\) and \(sQ\) by (rule add-mono)
thus bounded-by \((\text{bound-of } P + \text{bound-of } Q)\) \((\lambda s. P s + Q s)\)
  by (blast)

from \(sP\) have \(\forall s.\ 0 \leq P s\) by (blast)
moreover from \(sQ\) have \(\forall s.\ 0 \leq Q s\) by (blast)
ultimately have \(\forall s.\ 0 \leq P s + Q s\) by (simp add: add-mono)
thus \(\text{nneg}\) \((\lambda s. P s + Q s)\) by (blast)
qed

lemma mult-sound:
asumes \(sP\): sound \(P\) and \(sQ\): sound \(Q\)
shows sound \((\lambda s. P s * Q s)\)
proof
from \(sP\) have \(\forall s. P s \leq \text{bound-of } P\) by (blast)
moreover from \(sQ\) have \(\forall s. Q s \leq \text{bound-of } Q\) by (blast)
ultimately have \(\forall s. P s * Q s \leq \text{bound-of } P * \text{bound-of } Q\)
  using \(sP\) and \(sQ\) by (blast intro: mult-mono)
thus bounded-by \((\text{bound-of } P * \text{bound-of } Q)\) \((\lambda s. P s * Q s)\)
  by (blast)

from \(sP\) and \(sQ\) show \(\text{nneg}\) \((\lambda s. P s * Q s)\)
  by (blast intro: mult-nonneg-nonneg)
qed

lemma div-sound:
asumes \(sP\): sound \(P\) and \(\text{cpos}\): \(0 < c\)
shows sound \((\lambda s. P s / c)\)
proof
from \(sP\) and \(\text{cpos}\) have \(\forall s. P s / c \leq \text{bound-of } P / c\)
  by (blast intro: divide-right-mono less-imp-le)
thus bounded-by \((\text{bound-of } P / c)\) \((\lambda s. P s / c)\)
  by (blast)

from assms show \(\text{nneg}\) \((\lambda s. P s / c)\)
  by (blast intro: divide-nonneg-pos)
qed

lemma tminus-sound:
asumes \(sP\): sound \(P\) and \(\text{nnc}\): \(0 \leq c\)
3.1. EXPECTATIONS

shows sound \((\lambda s. P s \odot c)\)
proof (rule soundI)
from \(sP\) have \(\forall s. P s \leq \text{bound-of } P\) by (blast)
with \(\text{nnc}\) have \(\forall s. P s \odot c \leq \text{bound-of } P \odot c\)
  by (blast intro: minus-left-mono)
thus bounded \((\lambda s. P s \odot c)\) by (blast)
show \(\text{nneg } (\lambda s. P s \odot c)\) by (blast)
qed

lemma const-sound:
\(0 \leq c \implies \text{sound } (\lambda s. c)\)
by (blast)

lemma sound-o[intro,simp]:
\(\text{sound } P \implies \text{sound } (P o f)\)
unfolding o-def by (blast)

lemma sc-bounded-by[intro,simp]:
\([ \text{sound } P; 0 \leq c ] \implies \text{bounded-by } (c \ast \text{bound-of } P) (\lambda x. c \ast P x)\)
by (blast intro: mult-left-mono)

lemma sc-bounded[intro,simp]:
assumes \(sP:\ \text{sound } P \text{ and } \text{pos}: 0 \leq c\)
shows \(\text{bounded } (\lambda x. c \ast P x)\)
using assms by (blast)

lemma sc-bound[simp]:
assumes \(sP:\ \text{sound } P \text{ and } \text{cnn}: 0 \leq c\)
shows \(c \ast \text{bound-of } P = \text{bound-of } (\lambda x. c \ast P x)\)
proof (cases \(c = 0\))
case True then show \(?thesis\) by (simp)
next
case False with \(\text{cnn}\) have \(c < c\) by (auto)
show \(?thesis\)
proof (rule antisym)
  from \(sP\) and \(\text{cnn}\) have \(\text{bounded } (\lambda x. c \ast P x)\) by (simp)
hence \(\forall x. c \ast P x \leq \text{bound-of } (\lambda x. c \ast P x)\)
  by (rule le-bound-of)
with \(\text{cpos}\) have \(\forall x. P x \leq \text{inverse } c \ast \text{bound-of } (\lambda x. c \ast P x)\)
  by (force intro: mult-div-mono-right)
hence \(\text{bound-of } P \leq \text{inverse } c \ast \text{bound-of } (\lambda x. c \ast P x)\)
  by (blast)
with \(\text{cpos}\) show \(c \ast \text{bound-of } P \leq \text{bound-of } (\lambda x. c \ast P x)\)
  by (force intro: mult-div-mono-left)
next
from \(sP\) and \(\text{cpos}\) have \(\forall x. c \ast P x \leq c \ast \text{bound-of } P\)
  by (blast intro: mult-left-mono less-imp-le)
thus \(\text{bound-of } (\lambda x. c \ast P x) \leq c \ast \text{bound-of } P\)
\textbf{lemma sc-sound:}\n\[\text{sound } P; \ 0 \leq c \implies \text{sound } (\lambda s. c \cdot P s)\]
by (blast intro:mult-nonneg-nonneg)

\textbf{lemma bounded-by-mult:}\n\text{assumes } sP: \text{sound } P \text{ and } bP: \text{bounded-by } a \ P
\text{and } sQ: \text{sound } Q \text{ and } bQ: \text{bounded-by } b \ Q
\text{shows } \text{bounded-by } (a \cdot b) \ (\lambda s. P s \cdot Q s)
\text{using } \text{assms by (intro bounded-byI, auto intro:mult-mono)}

\textbf{lemma bounded-by-add:}\n\text{fixes } P::'s \Rightarrow \text{real and } Q
\text{assumes } bP: \text{bounded-by } a \ P
\text{and } bQ: \text{bounded-by } b \ Q
\text{shows } \text{bounded-by } (a + b) \ (\lambda s. P s + Q s)
\text{using } \text{assms by (intro bounded-byI, auto intro:add-mono)}

\textbf{lemma sound-unit[intro!,simp]:}\n\text{sound } (\lambda s. 1)
by (auto)

\textbf{lemma unit-mult[intro]:}\n\text{assumes } sP: \text{sound } P \text{ and } bP: \text{bounded-by } 1 \ P
\text{and } sQ: \text{sound } Q \text{ and } bQ: \text{bounded-by } 1 \ Q
\text{shows } \text{bounded-by } 1 \ (\lambda s. P s \cdot Q s)
\text{proof (rule bounded-byI)}
\text{fix } s
\text{have } P s \cdot Q s \leq 1 \cdot 1
\text{using } \text{assms by (blast dest:bounded-by-mult)}
\text{thus } P s \cdot Q s \leq 1 \text{ by (simp)}
qed

\textbf{lemma setsum-sound:}\n\text{assumes } sP: \forall x \in S. \text{sound } (P x)
\text{shows } \text{sound } (\lambda s. \sum x \in S. P x s)
\text{proof (rule soundI2)}
\text{from } sP \text{ show } \text{bounded-by } (\sum x \in S. \text{bound-of } (P x)) \ (\lambda s. \sum x \in S. P x s)
\text{by (auto intro!:setsum-mono)}
\text{from } sP \text{ show } \text{nneg } (\lambda s. \sum x \in S. P x s)
\text{by (auto intro!:setsum-nonneg)}
qed
3.1. EXPECTATIONS

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** unitary[intro]:
[ sound P; bounded-by 1 P ] ⇒ unitary P
by(simp add:unitary-def)

**lemma** unitary2:
[ nneg P; bounded-by 1 P ] ⇒ unitary P
by(auto)

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

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

3.1.5 Standard Expectations

**definition** embed-bool :: ('s ⇒ bool) ⇒ 's ⇒ real(" - " 1000)
**where** «P» ≡ (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.
\textbf{lemma embed-bool-idem:}
\[
\langle P \rangle s \ast \langle P \rangle s = \langle P \rangle s
\]
by (simp add:embed-bool-def)

\textbf{lemma eval-embed-true[simp]:}
\[
P \; s \quad \Rightarrow \quad \langle P \rangle s = 1
\]
by (simp add:embed-bool-def)

\textbf{lemma eval-embed-false[simp]:}
\[
\neg P \; s \quad \Rightarrow \quad \langle P \rangle s = 0
\]
by (simp add:embed-bool-def)

\textbf{lemma embed-ge-0[simp,intro]:}
\[
0 \leq \langle G \rangle s
\]
by (simp add:embed-bool-def)

\textbf{lemma embed-le-1[simp,intro]:}
\[
\langle G \rangle s \leq 1
\]
by (simp add:embed-bool-def)

\textbf{lemma embed-le-1-alt[simp,intro]:}
\[
0 \leq 1 - \langle G \rangle s
\]
by (subst add-le-cancel-right[where c=\langle G \rangle s, symmetric], simp)

\textbf{lemma expect-1-I:}
\[
P \; x \quad \Rightarrow \quad 1 \leq \langle P \rangle x
\]
by (simp)

\textbf{lemma standard-sound[intro,simp]:}
\[
\text{sound } \langle P \rangle
\]
by (blast)

\textbf{lemma embed-o[simp]:}
\[
\langle P \rangle \circ f = \langle P \circ f \rangle
\]
\textbf{unfolding embed-bool-def o-def by(simp)}

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

\textbf{definition negate :: ('}s ⇒ bool') ⇒ 's ⇒ bool (N)}
\textbf{where} \quad \text{negate } P = (\lambda s. \neg P \; s)

\textbf{lemma negate1:}
\[
\neg P \; s \quad \Rightarrow \quad N \; P \; s
\]
by (simp add:negate-def)

\textbf{lemma embed-split:}
\[
f \; s = \langle P \rangle \; s \ast f \; s + \langle N \; P \rangle \; s \ast f \; s
\]
by (simp add:negate-def embed-bool-def)
3.1. EXPECTATIONS

**Lemma** negate-embed:

\[ \langle \neg P \rangle s = 1 - \langle P \rangle s \]

by (simp add: embed-bool-def negate-def)

**Lemma** eval-nembed-true[simp]:

\[ P s \implies \langle \neg P \rangle s = 0 \]

by (simp add: embed-bool-def negate-def)

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

\[ \neg P s \implies \langle \neg P \rangle s = 1 \]

by (simp add: embed-bool-def negate-def)

**Lemma** negate-Not[simp]:

\[ \langle \neg \text{Not} \rangle = (\lambda x. x) \]

by (simp add: negate-def)

**Lemma** negate-negate[simp]:

\[ \langle \neg \langle \neg P \rangle \rangle = P \]

by (simp add: negate-def)

**Lemma** embed-bool-cancel:

\[ \langle G \rangle s * \langle \neg G \rangle s = 0 \]

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 ⇒ real) ⇒ (′s ⇒ real) ⇒ bool (- ⊢ - 50)

**Where** \( P \vdash Q \equiv P \leq Q \)

**Lemma** entailsI[intro]:

\[ \Gamma s. P s \leq Q s \implies P \vdash Q \]

by (simp add: le-funI)

**Lemma** entailsD[dest]:

\[ P \vdash Q \implies P s \leq Q s \]

by (simp add: le-funD)

**Lemma** eq-entails[intro]:

\[ P = Q \implies P \vdash Q \]

by (blast)

**Lemma** entails-trans[trans]:

\[ [ P \vdash Q; Q \vdash R ] \implies P \vdash R \]

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:

\[ \bigwedge s. \; P \; s \implies Q \; s \implies \langle P \rangle \vdash \langle Q \rangle \]
by (rule entailsI, case-tac P s, simp-all)

lemma entails-implies:

\[ \bigwedge s. \; \langle P \rangle \vdash \langle Q \rangle ; \; P \; s \implies Q \; s \]
by (rule ccontr, drule-tac s = s in entailsD, simp)

3.1.7 Expectation Conjunction

definition pconj :: real ⇒ real ⇒ real (infixl & 71)
where
\[ p \cdot & q \equiv p + q \ominus 1 \]

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]:
\[ b \leq 1 \implies 0 \cdot & b = 0 \]
by (simp add: pconj-def tminus-def)

lemma pconj-rzero[intro,simp]:
\[ b \leq 1 \implies b \cdot & 0 = 0 \]
by (simp add: pconj-def tminus-def)

lemma pconj-lone[intro,simp]:
\[ 0 \leq b \implies 1 \cdot & b = b \]
by (simp add: pconj-def tminus-def)

lemma pconj-rone[intro,simp]:
\[ 0 \leq b \implies b \cdot & 1 = b \]
by (simp add: pconj-def tminus-def)

lemma pconj-bconj:
\[ \langle a \rangle \; s \cdot & \langle b \rangle \; s = \langle \lambda s. \; a \; s \land \; b \; s \rangle \; s \]
unfolding embed-bool-def pconj-def tminus-def by (force)

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

lemma pconj-assoc:
\[ 0 \leq a; \; a \leq 1; \; 0 \leq b; \; b \leq 1; \; 0 \leq c; \; c \leq 1 \]
3.1. EXPECTATIONS

\[ a \land (b \land c) = (a \land b) \land c \]
\textbf{unfolding} pconj-def tminus-def \textbf{by}(simp)

\textbf{lemma} pconj-mono:
\[ \[(a \leq b) \land (c \leq d) \implies (a \land c \leq b \land d)\]
\textbf{unfolding} pconj-def tminus-def \textbf{by}(simp)

\textbf{lemma} pconj-nneg[intro,simp]:
\[ 0 \leq a \land b \]
\textbf{unfolding} pconj-def tminus-def \textbf{by}(auto)

\textbf{lemma} min-pconj:
\[ (\min a b) \land (\min c d) \leq \min (a \land c) (b \land d) \]
\textbf{by}(cases a \leq b,
(cases c \leq d,  
\text{simp-all add:} \min.\text{absorb1} \min.\text{absorb2} \text{pconj-mono}]],
(cases c \leq d,  
\text{simp-all add:} \min.\text{absorb1} \min.\text{absorb2} \text{pconj-mono})

\textbf{lemma} pconj-less-one[simp]:
\[ a + b < 1 \implies a \land b = 0 \]
\textbf{unfolding} pconj-def \textbf{by}(simp)

\textbf{lemma} pconj-ge-one[simp]:
\[ 1 \leq a + b \implies a \land b = a + b - 1 \]
\textbf{unfolding} pconj-def \textbf{by}(simp)

\textbf{lemma} pconj-idem[simp]:
\[ «P» s \land «P» s = «P» s \]
\textbf{unfolding} pconj-def \textbf{by}(cases P s, simp-all)

\textbf{3.1.8 Rules Involving Conjunction.}

\textbf{lemma} exp-conj-mono-left:
\[ P \vdash Q \implies P \land R \vdash Q \land R \]
\textbf{unfolding} exp-conj-def pconj-def
\textbf{by}(auto intro:tminus-left-mono add-right-mono)

\textbf{lemma} exp-conj-mono-right:
\[ Q \vdash R \implies P \land Q \vdash P \land R \]
\textbf{unfolding} exp-conj-def pconj-def
\textbf{by}(auto intro:tminus-left-mono add-left-mono)

\textbf{lemma} exp-conj-comm[ac-simps]:
\[ a \land b = b \land a \]
\textbf{by}(simp add:exp-conj-def ac-simps)

\textbf{lemma} exp-conj-bounded-by[intro,simp]:
\[ \text{ assumes } bP: \text{ bounded-by } 1 \]
\[ P \]
and \( bQ \) \( \text{bounded-by} \ 1 \ Q \)

shows \( \text{bounded-by} \ 1 \ (P \ & \& \ Q) \)

proof (rule \text{bounded-byI}, unfold \text{exp-conj-def} \text{ 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 \text{tminus-def} by (simp)

qed

lemma \text{exp-conj-o-distrib} [simp]:

\((P \ & \& \ Q) \ o f = (P \ o f) \ & \& (Q \ o f)\)

unfolding \text{exp-conj-def} \text{ o-def} by (simp)

lemma \text{exp-conj-assoc}:

assumes \text{unitary} \( P \) and \text{unitary} \( Q \) and \text{unitary} \( R \)

shows \( P \ & \& (Q \ & \& R) = (P \ & \& Q) \ & \& R \)

unfolding \text{exp-conj-def}

proof (rule \text{ext})

fix \( s \)

from \text{assms} have \( 0 \leq P \ s \) by (blast)

moreover from \text{assms} have \( 0 \leq Q \ s \) by (blast)

moreover from \text{assms} have \( 0 \leq R \ s \) by (blast)

moreover from \text{assms} have \( P \ s \leq 1 \) by (blast)

moreover from \text{assms} have \( Q \ s \leq 1 \) by (blast)

moreover from \text{assms} have \( R \ s \leq 1 \) by (blast)

ultimately show \( P \ s \ & (Q \ s \ & R \ s) = (P \ s \ & Q \ s) \ & R \ s \)

by (simp add: pconj-assoc)

qed

lemma \text{exp-conj-top-left} [simp]:

sound \( P \implies \ «\lambda . \ True\) \ & \& \( P \) = \( P \)

unfolding \text{exp-conj-def} by (force)

lemma \text{exp-conj-top-right} [simp]:

sound \( P \implies P \ & \ «\lambda . \ True\) = \( P \)

unfolding \text{exp-conj-def} by (force)

lemma \text{exp-conj-idem} [simp]:

\( «P\) \ & \& \ «P\) = \( P\)

unfolding \text{exp-conj-def}

by (rule \text{ext}, cases \( P \ s \), simp-all)

lemma \text{exp-conj-nneg} [intro, simp]:

\((\lambda s . \ 0) \leq P \ & \& \ Q\)

unfolding \text{exp-conj-def}

by (blast intro: \text{le-funI})
lemma \text{exp-conj-sound}[\text{intro, simp}];
\begin{itemize}
  \item \textbf{assumes} \( s\text{-}P \): \text{sound} \( P \)
  \item \textbf{and} \( s\text{-}Q \): \text{sound} \( Q \)
\end{itemize}
\textbf{shows} \text{sound} \(( P \&\& Q )\)
\textbf{unfolding} \text{exp-conj-def}
\textbf{proof}(\text{rule soundI})
\begin{itemize}
  \item \textbf{from} \( s\text{-}P \) \textbf{and} \( s\text{-}Q \) \textbf{have} \( \forall s. \ 0 \leq P s + Q s \) \textbf{by}(blast intro:add-nonneg-nonneg)
  \item \textbf{hence} \( \forall s. \ P s \& Q s \leq P s + Q s \)
  \item \textbf{unfolding} \text{pconj-def} \textbf{by}(force intro:tminus-less)
  \item \textbf{also from} \text{assms} \textbf{have} \( \forall s. \ s \leq \text{bound-of} \ P + \text{bound-of} \ Q \)
    \textbf{by}(blast intro:add-mono)
  \item \textbf{finally have} \text{bounded-by} \ ((\text{bound-of} \ P + \text{bound-of} \ Q) \ (\lambda s. \ P s \& Q s))
    \textbf{by}(blast)
\end{itemize}
\textbf{thus} \text{bounded} \ ((\lambda s. \ P s \& Q s)) \textbf{by}(blast)
\textbf{show} \text{nneg} \ ((\lambda s. \ P s \& Q s))
\textbf{unfolding} \text{pconj-def tminus-def} \textbf{by}(force)
\textbf{qed}

lemma \text{exp-conj-rzero}[\text{simp}]:
\begin{itemize}
  \item \text{bounded-by} \ 1 \ P \Rightarrow \ P \& \ (\lambda s. \ 0) = (\lambda s. \ 0)
\end{itemize}
\textbf{unfolding} \text{exp-conj-def} \textbf{by}(force)

lemma \text{exp-conj-1-right}[\text{simp}]:
\begin{itemize}
  \item \textbf{assumes} \text{n}: \text{nneg} \ A
  \item \textbf{shows} \( A \& \ (\lambda -. \ 1) = A \)
\end{itemize}
\textbf{unfolding} \text{exp-conj-def pconj-def tminus-def}
\textbf{proof}(\text{rule ext }, \text{simp})
\begin{itemize}
  \item \textbf{fix} \( s \)
  \item \textbf{from} \text{n} \textbf{have} \( 0 \leq A s \) \textbf{by}(blast)
  \item \textbf{thus} \text{max} \ (A s) \ 0 = A s \textbf{by}(force)
\end{itemize}
\textbf{qed}

lemma \text{exp-conj-std-split}:
\begin{itemize}
  \item \text{«} \lambda s. \ P s \& Q s » = \text{«} P » \& \ & \text{«} Q »
\end{itemize}
\textbf{unfolding} \text{exp-conj-def embed-bool-def pconj-def}
\textbf{by}(auto)

3.1.9 Rules Involving Entailment and Conjunction Togetherness

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

lemma \text{entails-frame}:
\begin{itemize}
  \item \textbf{assumes} ePR: \( P \vdash R \)
  \item \textbf{and} eQS: \( Q \vdash S \)
\end{itemize}
\textbf{shows} \( P \& \& Q \vdash R \& \& S \)
\textbf{proof}(\text{rule le-funI})
\begin{itemize}
  \item \textbf{fix} \( s \)
from ePR have \( P \ s \leq R \ s \) by (blast)
moreover from eQS have \( Q \ s \leq S \ s \) by (blast)
ultimately have \( P \ s + Q \ s \leq R \ s + S \ s \) by (rule add-mono)
thus \( (P \ & \& Q) \ s \leq (R \ & \& S) \ s \)

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 \( P Q e \) 
and exhaust: \( \forall x . P \ x \vdash Q \ x \)
and framed: \( \forall x . P \ x \ & \& R \vdash Q \ x \ & \& S \)
and sR: sound \( R \)
and sS: sound \( S \)
and bQ: \( \forall x . \text{bounded-by } 1 \ (Q \ x) \)

shows \( R \vdash S \)

proof (rule le-funI)
fix \( s \)
from exhaust obtain \( x \) where \( P-x:s \) \( P \ x \ s = 1 \) by (blast)
moreover 

\[
\begin{aligned}
& \text{hence } 1 = P \ x \ s \text{ by (simp)} \\
& \text{also from } P Q e \text{ have } P \ x \ s \leq Q \ x \ s \text{ by (blast dest:le-funD)} \\
& \text{finally have } Q \ x \ s = 1 \\
& \text{using } bQ \text{ by (blast intro:antisym)}
\end{aligned}
\]

moreover note le-funD[OF framed[where \( x=x \), where \( x=s \]]
moreover from sR have \( 0 \leq R \ s \) by (blast)
moreover from sS have \( 0 \leq S \ s \) by (blast)
ultimately show \( R \ s \leq S \ s \) by (simp add:exp-conj-def)

qed

lemma unitary-bot[iff]:
unitary (\( \lambda s . 0 :: \text{real} \))
by (auto)

lemma unitary-top[iff]:
unitary (\( \lambda s . 1 :: \text{real} \))
by (auto)

lemma unitary-embed[iff]:
unitary ‘\( P \)’
by (auto)

lemma unitary-const[iff]:
\( \lfloor 0 \leq c ; c \leq 1 \rfloor \implies \text{unitary } (\lambda s . c) \)
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 setsum-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: $P_{\text{prior}}(a) = 0.7 \times P_{\text{post}}(b) + 0.3 \times P_{\text{post}}(c)$, 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. 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: $P_{\text{prior}}(e) = 0.5 \times P_{\text{post}}(b) + 0.3 \times P_{\text{post}}(c)$.

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 probability 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.
3.2. EXPECTATION TRANSFORMERS

![Figure 3.3: A diverging automaton.]

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 $\land$ and $\lor$, 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**  
\[ \texttt{le-trans} :: \ 's \text{ trans} \Rightarrow 's \text{ trans} \Rightarrow \text{bool} \]

**Where**  
\[ \texttt{le-trans} \ t \ u \equiv \forall \ P. \ \text{sound} \ P \Rightarrow t \ P \leq u \ P \]

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

**Definition**  
\[ \texttt{le-utrans} :: \ 's \text{ trans} \Rightarrow 's \text{ trans} \Rightarrow \text{bool} \]

**Where**  
\[ \texttt{le-utrans} \ t \ u \leftarrow \rightarrow (\forall \ P. \ \text{unitary} \ P \Rightarrow t \ P \leq u \ P) \]

**Lemma** \[ \texttt{le-transI}[intro]: \]
\[ [ \forall P. \ \text{sound} \ P = \Rightarrow t \ P \leq u \ P ] \Rightarrow \texttt{le-trans} \ t \ u \]
\[ \text{by (simp add:le-trans-def)} \]

**Lemma** \[ \texttt{le-utransI}[intro]: \]
\[ [ \forall P. \ \text{unitary} \ P = \Rightarrow t \ P \leq u \ P ] \Rightarrow \texttt{le-utrans} \ t \ u \]
\[ \text{by (simp add:le-utrans-def)} \]

**Lemma** \[ \texttt{le-transD}[dest]: \]
\[ [ \texttt{le-trans} \ t \ u; \ \text{sound} \ P ] \Rightarrow t \ P \leq u \ P \]
\[ \text{by (simp add:le-trans-def)} \]

**Lemma** \[ \texttt{le-utransD}[dest]: \]
\[ [ \texttt{le-utrans} \ t \ u; \ \text{unitary} \ P ] \Rightarrow t \ P \leq u \ P \]
\[ \text{by (simp add:le-utrans-def)} \]

**Lemma** \[ \texttt{le-trans-trans}[trans]: \]
\[ [ \texttt{le-trans} \ x \ y; \ \texttt{le-trans} \ y \ z ] \Rightarrow \texttt{le-trans} \ x \ z \]
\[ \text{unfolding} \ \texttt{le-trans-def} \ \text{by (blast dest:order-trans)} \]

**Lemma** \[ \texttt{le-utrans-trans}[trans]: \]
\[ [ \texttt{le-utrans} \ x \ y; \ \texttt{le-utrans} \ y \ z ] \Rightarrow \texttt{le-utrans} \ x \ z \]
\[ \text{unfolding} \ \texttt{le-utrans-def} \ \text{by (blast dest:order-trans)} \]

**Lemma** \[ \texttt{le-trans-refl}[iff]: \]
\[ \texttt{le-trans} \ x \ x \]
\[ \text{by (simp add:le-trans-def)} \]

**Lemma** \[ \texttt{le-utrans-refl}[iff]: \]
\[ \texttt{le-utrans} \ x \ x \]
\[ \text{by (simp add:le-utrans-def)} \]
3.2. EXPECTATION TRANSFORMERS

\textbf{lemma le-trans-le-utrans[dest]}:
\begin{align*}
\text{le-trans } t\ u & \implies \text{le-utrans } t\ u \\
\text{unfolding le-trans-def le-utrans-def by(auto)}
\end{align*}

\textbf{definition}
\begin{align*}
\text{l-trans :: } & 's\ \text{trans } \Rightarrow \ 's\ \text{trans } \Rightarrow \ \text{bool} \\
\text{where} & \\
\text{l-trans } t\ u & \iff \text{le-trans } t\ u \land \neg \text{le-trans } u\ t
\end{align*}

Transformer equivalence is induced by comparison:

\textbf{definition}
\begin{align*}
\text{equiv-trans :: } & 's\ \text{trans } \Rightarrow \ 's\ \text{trans } \Rightarrow \ \text{bool} \\
\text{where} & \\
\text{equiv-trans } t\ u & \iff \text{le-trans } t\ u \land \text{le-trans } u\ t
\end{align*}

\textbf{definition}
\begin{align*}
\text{equiv-utrans :: } & 's\ \text{trans } \Rightarrow \ 's\ \text{trans } \Rightarrow \ \text{bool} \\
\text{where} & \\
\text{equiv-utrans } t\ u & \iff \text{le-utrans } t\ u \land \text{le-utrans } u\ t
\end{align*}

\textbf{lemma equiv-transI[dest]}:
\begin{align*}
[ \forall P. \text{sound } P \implies t\ P = u\ P ] & \implies \text{equiv-trans } t\ u \\
\text{unfolding equiv-trans-def by(force)}
\end{align*}

\textbf{lemma equiv-utransI[dest]}:
\begin{align*}
[ \forall P. \text{sound } P \implies t\ P = u\ P ] & \implies \text{equiv-utrans } t\ u \\
\text{unfolding equiv-utrans-def by(force)}
\end{align*}

\textbf{lemma equiv-transD[dest]}:
\begin{align*}
[ \text{equiv-trans } t\ u; \text{sound } P ] & \implies t\ P = u\ P \\
\text{unfolding equiv-trans-def by(blast intro:antisym)}
\end{align*}

\textbf{lemma equiv-utransD[dest]}:
\begin{align*}
[ \text{equiv-utrans } t\ u; \text{unitary } P ] & \implies t\ P = u\ P \\
\text{unfolding equiv-utrans-def by(blast intro:antisym)}
\end{align*}

\textbf{lemma equiv-trans-refl[iff]}:
\begin{align*}
\text{equiv-trans } t\ t \\
\text{by(blast)}
\end{align*}

\textbf{lemma equiv-utrans-refl[iff]}:
\begin{align*}
\text{equiv-utrans } t\ t \\
\text{by(blast)}
\end{align*}

\textbf{lemma le-trans-antisym}:
\begin{align*}
[ \text{le-trans } x\ y; \text{le-trans } y\ x ] & \implies \text{equiv-trans } x\ y \\
\text{unfolding equiv-trans-def by(simp)}
\end{align*}

\textbf{lemma le-utrans-antisym}:
\[ \text{le-utrans } x \; y; \text{le-utrans } y \; x \implies \text{equiv-utrans } x \; y \]

**unfolding** `equiv-utrans-def` by `simp`

**lemma** `equiv-trans-comm[ac-simps]`:

`equiv-trans` \( t \; u \iff equiv-trans \; u \; t \)

**unfolding** `equiv-trans-def` by `blast`

**lemma** `equiv-utrans-comm[ac-simps]`:

`equiv-utrans` \( t \; u \iff equiv-utrans \; u \; t \)

**unfolding** `equiv-utrans-def` by `blast`

**lemma** `equiv-imp-le[intro]`:

`equiv-trans` \( t \; u \implies \text{le-trans } t \; u \)

**unfolding** `equiv-trans-def` by `clarify`

**lemma** `equiv-imp-le-alt`:

`equiv-utrans` \( t \; u \implies \text{le-utrans } u \; t \)

by `force simp: ac-simps`

**lemma** `equiv-uimp-le-alt`:

`equiv-utrans` \( t \; u \implies \text{le-utrans } u \; t \)

by `force simp: ac-simps`

**lemma** `le-trans-equiv-rsp[simp]`:

`equiv-trans` \( t \; u \implies \text{le-trans } t \; v \iff \text{le-trans } u \; v \)

**unfolding** `equiv-trans-def` by `blast intro: le-trans-trans`

**lemma** `le-utrans-equiv-rsp[simp]`:

`equiv-utrans` \( t \; u \implies \text{le-utrans } t \; v \iff \text{le-utrans } u \; v \)

**unfolding** `equiv-utrans-def` by `blast intro: le-utrans-trans`

**lemma** `equiv-trans-le-trans[trans]`:

\[ \begin{array}{c}
\text{equiv-trans } t \; u; \text{le-trans } u \; v \\
\implies \text{le-trans } t \; v
\end{array} \]

by `simp`

**lemma** `equiv-utrans-le-utrans[trans]`:

\[ \begin{array}{c}
\text{equiv-utrans } t \; u; \text{le-utrans } u \; v \\
\implies \text{le-utrans } t \; v
\end{array} \]

by `simp`

**lemma** `le-trans-equiv-rsp-right[simp]`:

`equiv-trans` \( t \; u \implies \text{le-trans } v \; t \iff \text{le-trans } v \; u \)

**unfolding** `equiv-trans-def` by `blast intro: le-trans-trans`

**lemma** `le-utrans-equiv-rsp-right[simp]`:

`equiv-utrans` \( t \; u \implies \text{le-utrans } v \; t \iff \text{le-utrans } v \; u \)
3.2. EXPECTATION TRANSFORMERS

unfolding equiv-utrans-def by (blast intro: le-utrans-trans)

lemma le-trans-equiv-trans[trans]:
\[ \text{le-trans } t u; \text{ equiv-trans } u v \implies \text{le-trans } t v \]
by (simp)

lemma le-utrans-equiv-utrans[trans]:
\[ \text{le-utrans } t u; \text{ equiv-utrans } u v \implies \text{le-utrans } t v \]
by (simp)

lemma equiv-trans-trans:
assumes xy: \text{equiv-trans } x y
and yz: \text{equiv-trans } y z
shows \text{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: \text{equiv-utrans } x y
and yz: \text{equiv-utrans } y z
shows \text{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 \implies equiv-utrans t u
by (auto)

3.2.2 Healthy Transformers

Feasibility

definition feasible :: \((\forall a \Rightarrow \text{real}) \Rightarrow (\forall a \Rightarrow \text{real}) \Rightarrow \text{bool}\)
where feasible t \longleftrightarrow (\forall P b. \text{bounded-by } b P \land \text{nneg } P \rightarrow \\
\text{bounded-by } b (t P) \land \text{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
lemmas. 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 feasible[\text{intro}]:**
\[
\prod b \prod t \prod P \prod \text{bounded-by } b \prod P \prod \neg\neg P \quad \Rightarrow \quad \text{bounded-by } b \prod t \prod P
\]
by (force simp: feasible-def)

**lemma feasible-boundedD[dest]:**
\[
\prod t \prod \text{bounded-by } b \prod P \prod \neg\neg P \quad \Rightarrow \quad \text{bounded-by } b \prod t \prod P
\]
by (simp add: feasible-def)

**lemma feasible-nnegD[dest]:**
\[
\prod t \prod \text{bounded-by } b \prod P \prod \neg\neg P \quad \Rightarrow \quad \neg\neg \prod t \prod P
\]
by (simp add: feasible-def)

**lemma feasible-sound[dest]:**
\[
\prod t \prod \text{sound } P \quad \Rightarrow \quad \text{sound } \prod t \prod P
\]
by (rule soundI, unfold sound_def, (blast)+)

**lemma feasible-pr-0[simp]:**
\[
\text{fixes } t : (\lambda s. \text{real}) \Rightarrow \text{real}
\]
assumes \( \text{ft} \quad \text{feasible } t \)
shows \( t (\lambda x. 0) = (\lambda x. 0) \)
proof (rule ext, rule antisym)
fix \( s \)

have \( \text{bounded-by } 0 \quad (\lambda::'s. 0::\text{real}) \quad \text{by (blast)} \)
with \( \text{ft} \quad \text{have } \text{bounded-by } 0 \quad (t (\lambda s. 0)) \quad \text{by (blast)} \)
thus \( t (\lambda s. 0) \leq 0 \quad \text{by (blast)} \)

have \( \neg\neg (\lambda::'s. 0::\text{real}) \quad \text{by (blast)} \)
with \( \text{ft} \quad \text{have } \neg\neg (t (\lambda s. 0)) \quad \text{by (blast)} \)
thus \( 0 \leq t (\lambda s. 0) \quad \text{by (blast)} \)
qed

**lemma feasible-id:**
\( \text{feasible } (\lambda x. x) \)
unfolding feasible-def by (blast)

**lemma feasible-bounded-by[dest]:**
\[
\prod t \prod \text{sound } P \prod \text{bounded-by } b \prod P \quad \Rightarrow \quad \text{bounded-by } b \prod t \prod P
\]
by (auto)

**lemma feasible-fixes-top:**
\( \text{feasible } t \quad \Rightarrow \quad t (\lambda s. t) \leq (\lambda s. (1::\text{real})) \)
by (drule bounded-byD2[OF feasible-bounded-by], auto)

**lemma feasible-fixes-bot:**
3.2. EXPECTATION TRANSFORMERS

assumes ft: feasible $t$
shows $t\ (\lambda s. 0) = (\lambda s. 0)$
proof (rule antisym)
  have sb: sound ($\lambda s. 0$) by (auto)
  with ft show ($\lambda s. 0$) $\leq t\ (\lambda s. 0)$ by (auto)
  thm bound-of-const
from sb have bounded-by (bound-of ($\lambda s. 0 :: \text{real}$)) ($\lambda s. 0$) by (auto)
  hence bounded-by 0 ($\lambda s. 0 :: \text{real}$) by (simp add: bound-of-const)
  with ft show bounded-by 0 ($t\ (\lambda s. 0)$) by (auto)
thus $t\ (\lambda s. 0) \leq (\lambda s. 0)$ 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-trans :: (('s $\Rightarrow$ real) $\Rightarrow$ ('s $\Rightarrow$ real)) $\Rightarrow$ bool
where
  mono-trans $t \equiv \forall P\ Q.\ (\text{sound } P \land \text{sound } Q \land P \leq Q) \rightarrow t\ P \leq t\ Q$

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 $P \not\vdash u\ Q$, then by transitivity we have $P \vdash u\ (t\ R)$. We thus derive a probabilistic form of the standard rule for sequential composition: $[\text{mono-trans } t;\ P \not\vdash u\ Q;\ Q \vdash t\ R] \implies P \vdash u\ (t\ R)$.

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

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

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

**definition**

\[ \text{scaling} :: (('s \Rightarrow \text{real}) \Rightarrow ('s \Rightarrow \text{real})) \Rightarrow \text{bool} \]

**where**

\[ \text{scaling } t \equiv \forall P c x. \text{sound } P \land 0 \leq c \rightarrow c \ast t \ P \ x = t (\lambda x. \ c \ast P \ x) \ x \]

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): 

\[ t \ P \ s = \text{bound-of } P \ast t (\lambda s. \ P \ s / \text{bound-of } P) \ s \]

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]`:

\[ [ \ [\forall P c x. [\text{sound } P; 0 \leq c ] \rightarrow c \ast t \ P \ x = t (\lambda x. \ c \ast P \ x) \ x ] ] \Rightarrow \text{scaling } t \]

by `(simp add:scaling-def)`

**lemma** `scalingD[dest]`:

\[ [ \text{scaling } t; \text{sound } P; 0 \leq c ] \Rightarrow c \ast t \ P \ x = t (\lambda x. \ c \ast P \ x) \ x \]

by `(simp add:scaling-def)`

**lemma** `right-scalingD`:

assumes `st`: `scaling t`  
and `sP`: `sound P`  
and `nnc`: `0 \leq c`  
shows `t \ P \ s \ast c = t (\lambda s. \ P \ s \ast c) \ s`  

**proof** –

have `t \ P \ s \ast c = c \ast t \ P \ s` by `(simp add:algebra-simps)`  
also from `assms` have `... = t (\lambda s. \ c \ast P \ s) \ s` by `(rule scalingD)`  
also have `... = t (\lambda s. \ P \ s \ast c) \ s` by `(simp add:algebra-simps)`  
finally show `?thesis` .

qed

Healthiness

Healthy transformers are feasible and monotonic, and respect scaling

**definition**

\[ \text{healthy} :: (('s \Rightarrow \text{real}) \Rightarrow ('s \Rightarrow \text{real})) \Rightarrow \text{bool} \]

**where**

\[ \text{healthy } t \leftarrow \text{feasible } t \land \text{mono-trans } t \land \text{scaling } t \]

**lemma** `healthyI[intro]`:

\[ [ \text{feasible } t; \text{mono-trans } t; \text{scaling } t ] \Rightarrow \text{healthy } t \]

by `(simp add:healthy-def)`
3.2. EXPECTATION TRANSFORMERS

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

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

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

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

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

**lemma** healthy-bounded-byD[intro]:
  \[ \text{healthy t; bounded-by b P; nneg P} \] \implies bounded-by b (t P)
  by(blast)

**lemma** healthy-bounded-byD2:
  \[ \text{healthy t; bounded-by b P; sound P} \] \implies bounded-by b (t P)
  by(blast)

**lemma** healthy-boundedD[dest,simp]:
  \[ \text{healthy t; sound P} \] \implies bounded (t P)
  by(blast)

**lemma** healthy-nnegD[dest,simp]:
  \[ \text{healthy t; sound P} \] \implies nneg (t P)
  by(blast intro:feasible-nnegD)

**lemma** healthy-nnegD2[dest,simp]:
  \[ \text{healthy t; bounded-by b P; nneg P} \] \implies nneg (t P)
  by(blast)

**lemma** healthy-sound[intro]:
  \[ \text{healthy t; sound P} \] \implies sound (t P)
  by(rule soundI, blast, blast intro:feasible-nnegD)

**lemma** healthy-unitary[intro]:
  \[ \text{healthy t; unitary P} \] \implies 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 \textit{healthy-fixes-bot} = \textit{feasible-fixes-bot}[OF \textit{healthy-feasibleD}]

Some additional results on \textit{le-trans}, specific to \textit{healthy} transformers.

\textbf{lemma \textit{le-trans-bot}}[\textit{intro},\textit{simp}]:
\begin{align*}
\text{healthy } t & \Rightarrow \text{le-trans } (\lambda P\ s\ .\ 0)\ t \\
\text{by & (blast intro:le-funI)}
\end{align*}

\textbf{lemma \textit{le-trans-top}}[\textit{intro},\textit{simp}]:
\begin{align*}
\text{healthy } t & \Rightarrow \text{le-trans } t\ (\lambda P\ s\ .\ \text{bound-of } P) \\
\text{by & (blast intro:le-transI[OF le-funI])}
\end{align*}

\textbf{lemma \textit{healthy-pr-bot}}[\textit{simp}]:
\begin{align*}
\text{healthy } t & \Rightarrow t\ (\lambda s\ .\ 0) = (\lambda s\ .\ 0) \\
\text{by & (blast intro:feasible-pr-0)}
\end{align*}

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

\textbf{lemma \textit{healthy-equivI}}:
\begin{align*}
\text{fixes } & t::(\text{'}s\Rightarrow\text{real}) \Rightarrow \text{'}s\Rightarrow\text{real and } u \\
\text{assumes equiv: & equiv-trans } t\ u \\
\text{and healthy: & healthy } t \\
\text{shows & healthy } u
\end{align*}

\textbf{proof}
\begin{align*}
\text{have & le-t-u: le-trans } t\ u\ \text{by(blast intro:equiv)}
\end{align*}

\textbf{proof}
\begin{align*}
\text{have & le-u-t: le-trans } u\ t\ \text{by(simp add:equiv-imp-le ac-simps equiv)}
\end{align*}

\textbf{from}
\begin{align*}
\text{equiv have & eq-u-t: equiv-trans } u\ t\ \text{by(simp add:ac-simps)}
\end{align*}

\textbf{show}
\begin{align*}
\text{feasible } u
\end{align*}

\textbf{proof}
\begin{align*}
\text{fix } & b\ \text{and } P::\text{'}s\Rightarrow\text{real} \\
\text{assume } & bP: \text{bounded-by } b\ P\ and\ nP: \text{nneg } P \\
\text{hence } & sp: \text{sound } P\ \text{by(blast)} \\
\text{with healthy have } & \forall s.\ 0 \leq t\ P\ s\ \text{by(blast)} \\
\text{also from } & sp\ and\ le-t-u\ have\ \forall s.\ ...\ s \leq u\ P\ s\ \text{by(blast)} \\
\text{finally show } & \text{nneg } (u\ P)\ \text{by(blast)}
\end{align*}

\textbf{from}
\begin{align*}
\text{sp\ and\ le-u-t\ have } & \forall s.\ u\ P\ s \leq t\ P\ s\ \text{by(blast)} \\
\text{also from } & \text{healthy\ and } sp\ and\ bP\ have\ \forall s.\ t\ P\ s \leq b\ \text{by(blast)} \\
\text{finally show } & \text{bounded-by } b\ (u\ P)\ \text{by(blast)}
\end{align*}

\textbf{qed}

\textbf{show}
\begin{align*}
\text{mono-trans } u
\end{align*}

\textbf{proof}
\begin{align*}
\text{fix } & P::\text{'}s\Rightarrow\text{real\ and } Q::\text{'}s\Rightarrow\text{real} \\
\text{assume } & sp: \text{sound } P\ and\ sQ: \text{sound } Q \\
\text{and } & le: P \vdash Q \\
\text{from } & sp\ and\ le-u-t\ have\ u\ P \vdash t\ P\ \text{by(blast)} \\
\text{also from } & sp\ and\ sQ\ and\ le\ and\ healthy\ have\ t\ P \vdash t\ Q\ \text{by(blast)} \\
\text{also from } & sQ\ and\ le-t-u\ have\ t\ Q \vdash u\ Q\ \text{by(blast)} \\
\text{finally show } & u\ P \vdash u\ Q
\end{align*}. 
3.2. EXPECTATION TRANSFORMERS

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

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)
CHAPTER 3. SEMANTIC STRUCTURES

with \( nc \) have \( \forall s. c \cdot t P s \leq c \cdot b \) by(blast intro:mult-left-mono)
also {
from \( nnP \) and \( bP \) have \( \theta \leq b \) by(auto)
with \( bc \) have \( c \cdot b \leq 1 \cdot b \) by(blast intro:mult-right-mono)
  hence \( c \cdot b \leq b \) by(simp)
}
finally show bounded-by \( b \) (\( \lambda s. c \cdot t P s \)) by(blast)
from \( ht \) \( nnP \) \( bP \) have \( \forall s. \theta \leq t P s \) by(blast)
with \( nc \) have \( \forall s. 0 \leq c \cdot t P s \) by(rule mult-nonneg-nonneg)
thus \( nneg \) (\( \lambda s. c \cdot t P s \)) by(blast)
qed

show mono-trans (\( \lambda P s. c \cdot t P s \))
proof
  fix \( P::'s \Rightarrow \text{real} \) and \( Q \)
  assume \( sP: \text{sound} \ P \) and \( sQ: \text{sound} \ Q \) and \( le: P \vdash \vdash Q \)
  with \( ht \) have \( \forall s. t P s \leq t Q s \) by(auto intro:le-funD)
  with \( nc \) have \( \forall s. c \cdot t P s \leq c \cdot t Q s \)
    by(blast intro:mult-left-mono)
  thus \( \forall s. c \cdot t P s \vdash \vdash c \cdot t Q s \) by(blast)
qed

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

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

lemma healthy-bot[iff]:
  healthy (\( \lambda P s. \theta \))
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 \text{real}) \Rightarrow ('s \Rightarrow \text{real})) \Rightarrow \text{bool}
where
  nearly-healthy \( t \) \( \sim\rightarrow \) (\( \forall P. \text{unitary} \ P \Rightarrow \text{unitary} \ (t P) \)) \( \land \)
    (\( \forall P. \text{unitary} \ P \Rightarrow \text{unitary} \ Q \Rightarrow P \vdash Q \Rightarrow t P \vdash t Q \))

lemma nearly-healthy[intro]:
[ \( \forall P. \text{unitary} \ P \Rightarrow \text{unitary} \ (t P) \); \( \forall P Q. \text{unitary} \ P; \text{unitary} \ Q; P \vdash Q \) \( \Rightarrow t P \vdash t Q \) ] \( \Rightarrow \) nearly-healthy \( t \)
by(simp add:nearly-healthy-def)

lemma nearly-healthy-mono[dest]:
3.2. EXPECTATION TRANSFORMERS

\[ \text{nearly-healthy } t; \quad P \vdash Q; \quad \text{unitary } P; \quad \text{unitary } Q \implies t P \vdash t Q \]

by (simp add: nearly-healthy-def)

lemma nearly-healthy-unitaryD[dest]:
\[ \text{nearly-healthy } t; \quad \text{unitary } P \implies \text{unitary } (t P) \]
by (simp add: nearly-healthy-def)

lemma healthy-nearly-healthy[dest]:
assumes ht: \text{healthy } t
shows \text{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. \( \max(x - y) \, (\theta::\theta) \) (see Section 4.13.1).

definition sublinear ::
\((('s \Rightarrow \text{real}) \Rightarrow ('s \Rightarrow \text{real})) \Rightarrow \text{bool}\)
where
\[ \text{sublinear } t \leftarrow \forall a \ b \ c \ P \ Q \ s. \ (\text{sound } P \land \text{sound } Q \land 0 \leq a \land 0 \leq b \land 0 \leq c) \]
\[ \implies a \ast t P s + b \ast t Q s \ominus c \leq t (\lambda s'. a \ast P s' + b \ast Q s' \ominus c) s \]

lemma sublinearI[intro]:
\[ \forall a \ b \ c \ P \ Q \ s. \ [ \text{sound } P; \text{sound } Q; \ 0 \leq a; \ 0 \leq b; \ 0 \leq c ] \implies a \ast t P s + b \ast t Q s \ominus c \leq t (\lambda s'. a \ast P s' + b \ast Q s' \ominus c) s \]
by (simp add: sublinear-def)

lemma sublinearD[dest]:
\[ \text{sublinear } t; \quad \text{sound } P; \quad \text{sound } Q; \ 0 \leq a; \ 0 \leq b; \ 0 \leq c \implies a \ast t P s + b \ast t Q s \ominus c \leq t (\lambda s'. a \ast P s' + b \ast Q s' \ominus c) s \]
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

\textbf{definition} \textit{sub-add} ::
\[
(\forall s \Rightarrow \text{real}) \Rightarrow (\forall s \Rightarrow \text{real}) \Rightarrow \text{bool}
\]
\textbf{where}
\[
\text{sub-add } t \leftrightarrow (\forall P Q s. (\text{sound } P \land \text{sound } Q) \rightarrow \\
P s + t Q s \leq t (\lambda s'. P s' + Q s') s)
\]

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

Here \(P\) is an affine function (expectation) \(\text{real} \Rightarrow \text{real}\), 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 \(b\).

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.
3.2. EXPECTATION TRANSFORMERS

that respect scaling. Note the form of the definition of convexity:

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

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

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

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]:

\[ \forall P, Q, s. \ [ \text{sound } P; \text{sound } Q ] \Rightarrow t \ P s + t \ Q s \leq t \ (\lambda s'. P \ s' + Q \ s') s \] \Rightarrow sub-add t

by(simp add:sub-add-def)

**lemma** sub-addI2:

\[ \forall P, Q. \ [ \text{sound } P; \text{sound } Q ] \Rightarrow \ lambda t \ P s + t \ Q s \Rightarrow t \ (\lambda s. P \ s + Q \ s) s \] \Rightarrow sub-add t

by(auto)

**lemma** sub-addD[dest]:

\[ \text{sub-add } t; \text{sound } P; \text{sound } Q \Rightarrow t \ P s + t \ Q s \leq t \ (\lambda s'. P \ s' + Q \ s') s \]

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 $t \ P s + t \ Q s \leq t \ (\lambda s. P \ s + Q \ s) s$

by(auto)

also {

from $sP \ sQ$ have sound $(\lambda s. P \ s + Q \ s)$ by(auto)
with \( \text{eq have} \ t \ (\lambda s. \ P \ s + Q \ s) \ s = u \ (\lambda s. \ P \ s + Q \ s) \ s \)
by(simp add:equiv-transD)
}
finally show \( u \ P \ s + u \ Q \ s \leq u \ (\lambda s. \ P \ s + Q \ s) \ s \).
qed

Sublinearity and feasibility imply sub-additivity.

\textbf{lemma sublinear-subadd:}
fixes \( t::(\prime s \Rightarrow \text{real}) \Rightarrow \prime s \Rightarrow \text{real} \)
assumes slt: \( \text{sublinear} \ t \)
and ft: \( \text{feasible} \ t \)
shows \( \text{sub-add} \ t \)
proof
fix \( P::\prime s \Rightarrow \text{real} \) and \( Q::\prime s \Rightarrow \text{real} \) and \( s::\prime s \)
assume sP: \( \text{sound} \ P \) and sQ: \( \text{sound} \ Q \)
with ft have \( \text{sound} \ (t \ P) \) sound \( (t \ Q) \) by(auto)
hence \( 0 \leq t \ P \ s \) and \( 0 \leq t \ Q \ s \) by(auto)
hence \( 0 \leq t \ P \ s + t \ Q \ s \) by(auto)
hence ... = ...\( \ominus 0 \) by(simp)
also from \( sP \ sQ \)
have ... \( \leq t \ (\lambda s. \ P \ s + Q \ s \ominus 0) \ s \)
by(rule sublinearD[OF slt, \( \text{where} \ a=1 \) and \( b=1 \) and \( c=0 \), \( \text{simplified} \)])
also {
from \( sP \ sQ \) have \( \exists s. \ 0 \leq P \ s \) and \( \exists s. \ 0 \leq Q \ s \) by(auto)
hence \( \exists s. \ 0 \leq P \ s + Q \ s \) by(auto)
hence \( t \ (\lambda s. \ P \ s + Q \ s \ominus 0) \ s = t \ (\lambda s. \ P \ s + Q \ s) \ s \)
by(simp)
}
finally show \( t \ P \ s + t \ Q \ s \leq t \ (\lambda s. \ P \ s + Q \ s) \ s \).
qed

A few properties following from sub-additivity:

\textbf{lemma standard-negate:}
assumes ht: \( \text{healthy} \ t \)
and sat: \( \text{sub-add} \ t \)
shows \( t \ « \ P » \ s + t \ « \N P » \ s \leq 1 \)
proof –
from sat have \( t \ « \ P » \ s + t \ « \N P » \ s \leq t \ (\lambda s. \ « \ P » \ s + « \N P » \ s) \ s \) by(auto)
also have ... = \( t \ (\lambda s. \ 1) \ s \) by(simp add:negate-embed)
also {
from ht have \( \text{bounded-by} \ 1 \ (t \ (\lambda s. \ 1)) \) by(auto)
hence \( t \ (\lambda s. \ 1) \ s \leq 1 \) by(auto)
}
finally show \( ?\text{thesis} \).
qed
3.2. EXPECTATION TRANSFORMERS

lemma sub-add-setsum:
  fixes t::'s trans and S::'a set
  assumes sat: sub-add t
  and ht: healthy t
  and sP: \forall x. sound (P x)
  shows (\lambda x. \sum y\in S. t (P y) x) \leq t (\lambda x. \sum y\in 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 (\lambda s. 0)) by(auto)
    thus 0 \leq t (\lambda s. 0) s by(auto)
  next
    fix F::'a set and x::'a
    assume IH: \lambda a. \sum y\in F. t (P y) a \vdash t (\lambda x. \sum y\in F. P y x)
    hence t (P x) s + (\sum y\in F. t (P y) s) \leq t (P x) s + t (\lambda x. \sum y\in F. P y x) s
      by(auto intro:add-left-mono)
    also from sat sP
    have ... \leq t (\lambda xa. P x xa + (\sum y\in F. P y xa)) s
      by(auto intro!:sub-addD[OF sat] setsum-sound)
    finally
    show t (P x) s + (\sum y\in F. t (P y) s) \leq t (\lambda xa. P x xa + (\sum y\in 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 (\sum y\in\{s. G s\} \cdot P y \cdot t \langle \lambda z. z = y \rangle s) +
    (\sum y\in\{s. \neg G s\} \cdot P y \cdot t \langle \lambda z. z = y \rangle s) \leq t P s
proof
  have \{s. G s\} \cap \{s. \neg G s\} = \{\} by(blast)
  hence (\sum y\in\{s. G s\} \cdot P y \cdot t \langle \lambda z. z = y \rangle s) +
    (\sum y\in\{s. \neg G s\} \cdot P y \cdot t \langle \lambda z. z = y \rangle s) =
    (\sum y\in\{s. G s\} \cup \{s. \neg G s\} \cdot P y \cdot t \langle \lambda z. z = y \rangle s)
      by(auto intro: setsum.union_disjoint[symmetric])
  also
  have \{s. G s\} \cup \{s. \neg G s\} = UNIV by (blast)
  hence (\sum y\in\{s. G s\} \cup \{s. \neg G s\} \cdot P y \cdot t \langle \lambda z. z = y \rangle s) =
    (\lambda x. \sum y\in UNIV. P y \cdot t (\lambda x. \langle \lambda z. z = y \rangle x) s)
      by(simp)
  } also

from sP have \( \forall y. 0 \leq P y \) by(auto)
with healthy-scalingD[OF ht]
have \( \lambda x. \sum y \in \text{UNIV}. P y \ast t (\lambda x. (\lambda z. z = y) x) x \) s = 
\( \lambda x. \sum y \in \text{UNIV}. t (\lambda x. P y \ast (\lambda z. z = y) x) x \) s
by(simp add: scalingD)
}
also { from sat ht sP 
have \( \lambda x. \sum y \in \text{UNIV}. t (\lambda x. P y \ast (\lambda z. z = y) x) \) \leq 
\( \lambda x. \sum y \in \text{UNIV}. P y \ast (\lambda z. z = y) x \) 
by(intro sub-add-setsum sound-intros, auto)

hence \( \lambda x. \sum y \in \text{UNIV}. t (\lambda x. P y \ast (\lambda z. z = y) x) \) s \leq 
\( \lambda x. \sum y \in \text{UNIV}. P y \ast (\lambda z. z = y) x \) s
by(auto)
}
also { have rw1: \( \lambda x. \sum y \in \text{UNIV}. P y \ast (\lambda z. z = y) x \) = 
\( \lambda x. \sum y \in \text{UNIV}. \text{if } y = x \text{ then } P y \text{ else } 0 \) 
by(auto intro!: setsum.cong)
also from sP have \( \vdash P \)
by(cases finite (UNIV::'s set), auto simp: setsum.delta)
finally have leP: \( \lambda x. \sum y \in \text{UNIV}. P y \ast (\lambda z. z = y) x \) \leq P .
moreover have sound \( \lambda x. \sum y \in \text{UNIV}. P y \ast (\lambda z. z = y) x \)
proof(intro soundI2 bounded-byI nnegI setsum-nonneg ballI)
fix x 
from leP have \( \sum y \in \text{UNIV}. P y \ast (\lambda z. z = y) x \) \leq P x by(auto)
also from sP have \( \leq \) bound-of P by(auto)
finally show \( \sum y \in \text{UNIV}. P y \ast (\lambda z. z = y) x \) \leq bound-of P .
fix y
from sP show \( 0 \leq P y \ast (\lambda z. z = y) x \)
by(auto intro!: mult-nonneg-nonneg)
qed

ultimately have \( t (\lambda x. \sum y \in \text{UNIV}. P y \ast (\lambda z. z = y) x) \) s \leq 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 :: 
\( (('s \Rightarrow \text{real}) \Rightarrow ('s \Rightarrow \text{real})) \Rightarrow \text{bool} \)
where
sub-distrib t \( \iff \) \( \forall P s. \text{sound} P \rightarrow t P s \odot 1 \leq t (\lambda s'. P s' \odot 1) s \)

lemma sub-distribI[intro]:
\[ \forall P s. \text{sound} P \Rightarrow t P s \odot 1 \leq t (\lambda s'. P s' \odot 1) s \] \( \Rightarrow \) sub-distrib t
by(simp add: sub-distrib-def)
3.2. EXPECTATION TRANSFORMERS

**lemma** sub-distribI2:
\[ \bigwedge P. \text{sound } P \implies \lambda s. t P s \oplus 1 \vdash t (\lambda s. P s \oplus 1) \] \implies \text{sub-distrib } t
by(auto)

**lemma** sub-distribD[dest]:
\[ [ \text{sub-distrib } t; \text{sound } P ] \implies t P s \oplus 1 \leq t (\lambda s'. P s' \oplus 1) s \]
by(simp add:sub-distrib-def)

**lemma** equiv-sub-distrib:
fixes \( t :: (\prime s \Rightarrow \text{real}) \Rightarrow \prime s \Rightarrow \text{real} \)
assumes eq: equiv-trans \( t u \)
and sd: sub-distrib \( t \)
shows sub-distrib \( u \)
proof
fix \( P :: \prime s \Rightarrow \text{real} \) and \( s :: \prime s \)
assume sP: sound \( P \)
moreover have \( \text{sound } (\lambda s. 0) \) by(auto)
ultimately show \( t P s \oplus 1 \leq t (\lambda s. P s \oplus 1) s \)
by(rule sublinearD[OF sd, where a=1 and b=0 and c=1, simplified])
qed

Sublinearity implies sub-distributivity:

**lemma** sublinear-sub-distrib:
fixes \( t :: (\prime s \Rightarrow \text{real}) \Rightarrow \prime s \Rightarrow \text{real} \)
assumes slt: sublinear \( t \)
shows sub-distrib \( t \)
proof
fix \( P :: \prime s \Rightarrow \text{real} \) and \( s :: \prime s \)
assume sP: sound \( P \)
moreover have sound \( (\lambda s. 0) \) by(auto)
ultimately show \( t P s \oplus 1 \leq t (\lambda s. P s \oplus 1) s \)
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 \( t :: (\prime s \Rightarrow \text{real}) \Rightarrow \prime s \Rightarrow \text{real} \)
assumes sdt: sub-distrib \( t \) and sat: sub-add \( t \) and ht: healthy \( t \)
shows sublinear \( t \)
proof
fix \( P :: \prime s \Rightarrow \text{real} \) and \( Q :: \prime s \Rightarrow \text{real} \) and \( s :: \prime s \)
and a::real and b::real and c::real
assume sP: sound \( P \) and sQ: sound \( Q \)
and nna: \( 0 \leq a \) and nnb: \( 0 \leq b \) and nnc: \( 0 \leq c \)
from \(ht \, sP \, sQ \, nna \, nmb\)

have \(saP\): sound \((\lambda s. a * P \, s)\) and \(staP\): sound \((\lambda s. a * t \, P \, s)\)
and \(shQ\): sound \((\lambda s. b * Q \, s)\) and \(stbQ\): sound \((\lambda s. b * t \, Q \, s)\)
by\((auto \, intro:sc\text{-}sound)\)

hence \(sabPQ\): sound \((\lambda s. a * P \, s + b * Q \, s)\)
and \(stabPQ\): sound \((\lambda s. a * t \, P \, s + b * t \, Q \, s)\)
by\((auto \, intro:\text{sound}-\text{sum})\)

from \(ht \, sP \, sQ \, nna \, nmb\)

have \(a * t \, P \, s + b * t \, Q \, s = t \, (\lambda s. a * P \, s) + t \, (\lambda s. b * Q \, s)\)
by\((simp \, add:\text{scaling}D \, \text{healthy-scaling}D)\)
also from \(saP \, sbQ \, sat\)

have \(t \, (\lambda s. a * P \, s) + t \, (\lambda s. b * Q \, s)\) \leq
\(t \, (\lambda s. a * P \, s + b * Q \, s)\)
by\((\text{blast})\)
finally

have \(notm: a * t \, P \, s + b * t \, Q \, s \leq t \, (\lambda s. a * P \, s + b * Q \, s)\)

show \(a * t \, P \, s + b * t \, Q \, s \otimes c \leq t \, (\lambda s'. a * P \, s' + b * Q \, s' \otimes c)\)
proof\((\text{cases c = 0})\)

  case \(True\) note \(z = \text{this}\)
  from \(stabPQ\) have \(\exists s. 0 \leq a * t \, P \, s + b * t \, Q \, s\)
  by\((auto)\)
  moreover from \(sabPQ\)
  have \(\exists s. 0 \leq a * P \, s + b * Q \, s\)
  by\((auto)\)
  ultimately show \(\exists\text{thesis}\)
  by\((simp \, add:\text{z notm})\)

next

  case \(False\) note \(nz = \text{this}\)
  from \(nz\) and \(nni\) have \(nni: 0 \leq inverse \, c\)
  by\((auto)\)

  have \(\exists s. \, (inverse \, c * a) * P \, s + (inverse \, c * b) * Q \, s =
  inverse \, c * (a * P \, s + b * Q \, s)\)
  by\((simp \, add:\text{divide-simps})\)
  with \(sabPQ\) and \(nni\)
  have \(si: \, \text{sound} \, (\lambda s. (inverse \, c * a) * P \, s + (inverse \, c * b) * Q \, s)\)
  by\((\text{auto intro:sc\text{-}sound})\)
  hence \(sim: \, \text{sound} \, (\lambda s. (inverse \, c * a) * P \, s + (inverse \, c * b) * Q \, s \otimes 1)\)
  by\((\text{auto intro!:\text{tminus\text{-}sound}})\)

from \(nz\)

have \(a * t \, P \, s + b * t \, Q \, s \otimes c =
(c * inverse \, c) * a * t \, P \, s +
(c * inverse \, c) * b * t \, Q \, s \otimes c\)
by\((simp)\)
also

have \(... = c * (inverse \, c * a * t \, P \, s) +
c * (inverse \, c * b * t \, Q \, s) \otimes c\)
by\((simp \, add:\text{field\text{-}simps})\)
also from \(nni\)

have \(... = c * (inverse \, c * a * t \, P \, s + inverse \, c * b * t \, Q \, s \otimes 1)\)
by\((simp \, add:\text{distrib\text{-}left \text{tminus\text{-}left\text{-}distrib}})\)
3.2. EXPECTATION TRANSFORMERS

also {  
  have X: \( \bigwedge s. \text{(inverse } c * a) * t \ P s + \text{(inverse } c * b) * t \ Q s = \text{inverse } c * (a * t \ P s + b * t \ Q s) \) by(simp add: divide-simps)

also from nni and notm
  have inverse c * (a * t \ P s + b * t \ Q s) \leq
      inverse c * (t (\lambda s. a * P s + b * Q s) s)
    by(blast intro:mult-left-mono)

finally
  have (inverse c * a) * t \ P s + (inverse c * b) * t \ Q s \ominus 1 \leq
      t (\lambda s. (inverse c * a) * P s + (inverse c * b) * Q s) s \ominus 1
    by(rule tminus-left-mono)

also {  
    from sdt si
    have t (\lambda s. (inverse c * a) * P s + (inverse c * b) * Q s) s \ominus 1 \leq
      t (\lambda s. (inverse c * a) * P s + (inverse c * b) * Q s \ominus 1) s
    by(blast)
  }

finally
  have c * (inverse c * a * t \ P s + inverse c * b * t \ Q s \ominus 1) \leq
      c * t (\lambda s. inverse c * a * P s + inverse c * b * Q s \ominus 1) s
    using nnc by(blast intro:mult-left-mono)

}  
also from nnc ht sim
  have c * t (\lambda s. inverse c * a * P s + inverse c * b * Q s \ominus 1) s
    = t (\lambda s. c * (inverse c * a * P s + inverse c * b * Q s \ominus 1)) s
    by(simp add:scalingD healthy-scalingD)

also from nnc
  have ... = t (\lambda s. (c * inverse c) * a * P s +
      (c * inverse c) * b * Q s \ominus 1) s
    by(simp add:distrib-left tminus-left-distrib)

also have ... = t (\lambda s. (c * inverse c) * a * P s +
      (c * inverse c) * b * Q s \ominus 1) s
    by(simp add:field-simps)

finally
  show a * t \ P s + b * t \ Q s \ominus c \leq t (\lambda s'. a * P s' + b * Q s' \ominus 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]:
\[ \forall P, Q. [\text{sound } P; \text{sound } Q] \Rightarrow t P \land t Q \vdash t (P \land Q) \]\nunfolding sub-conj-def by(simp)

lemma sub-conjD[dest]:
\[ [\text{sub-conj } t; \text{sound } P; \text{sound } Q] \Rightarrow t P \land t Q \vdash t (P \land Q) \]\nunfolding 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)
proof(rule sub-conjI, rule le-funI, unfold exp-conj-def pconj-def)
fix P ‚ s ⇒ real and Q ‚ s ⇒ real and s
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 ⇒ real) ⇒ ’s ⇒ 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 ⇒ real and Q ‚ s ⇒ real and s
assume sP: sound P and sQ: sound Q
thus t P s + t Q s ⊙ 1 ≤ t (λs. P s + Q s ⊙ 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:
\[ [\text{equiv-trans } t w; \text{sublinear } t; \text{healthy } t] \Rightarrow \text{sublinear } u \]
by(prover 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 \((\text{sub-})\)probabilistic transformers however.

**definition**

\[
\text{additive} :: (\forall a \Rightarrow \text{real}) \Rightarrow (\forall a \Rightarrow \text{real}) \Rightarrow \text{bool}
\]

**where**

\[
\text{additive } t \equiv \forall P \ Q. \ (\text{sound } P \land \text{sound } Q) \implies t \ (\lambda s. \ P s + Q s) = (\lambda s. \ t P s + t Q s)
\]

**lemma** *additiveD*:

\[
[ [ \text{additive } t; \text{sound } P; \text{sound } Q ] ] \implies t \ (\lambda s. \ P s + Q s) = (\lambda s. \ t P s + t Q s)
\]

by \((\text{simp add: additive-def})\)

**lemma** *additiveI[intro]*:

\[
[ [ \forall P \ Q s. \ [ [ \text{sound } P; \text{sound } Q ] ] \implies t \ (\lambda s. \ P s + Q s) s = t P s + t Q s ] ] \implies \text{additive } t
\]

unfolding \text{additive-def} by \((\text{blast})\)

Additivity is strictly stronger than sub-additivity.

**lemma** *additive-sub-add*:

\[
\text{additive } t = \implies \text{sub-add } t
\]

by \((\text{simp add: sub-addI additiveD})\)

The additivity property extends to finite summation.

**lemma** *additive-setsum*:

fixes \(S::\text{'s set}\)

assumes \(\text{additive}: \text{additive } t\)

and \(\text{healthy}: \text{healthy } t\)

and \(\text{finite}: \text{finite } S\)

and \(sPz: \ \forall z. \text{sound } (P z)\)

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 \(z::\text{'s set}\) and \(T::\text{'s set}\)

assume \(\text{finT}: \text{finite } T\)

and \(IH: t \ (\lambda x. \ \sum y \in T. \ P y x) = (\lambda x. \ \sum y \in T. \ t (P y) x)\)

from \(\text{additive } sPz\)

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 \((\text{auto intro!: setsum-sound additiveD})\)

also from \(IH\)
have ... = (λx. t (P z) x + (∑y∈T. t (P y) x))
  by(simp)

finally show t (λx. P z x + (∑y∈T. P y x)) =
  (λx. t (P z) x + (∑y∈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 ⇒ real) ⇒ 's ⇒ real
  assumes additive: additive t
       and ht: healthy t
       and sP: sound P
  shows t P x = (∑y∈UNIV. P y * t «λz. z = y» x)
proof –
  have (∑y∈UNIV. P y * «λz. z = y» x) =
        (∑y∈UNIV. if y = x then P y else 0)
    by(auto intro:setsum.cong)
  also have (∑y∈UNIV. y) = P x
    by(simp add:setsum.delta)
finally
  have t P x = t (λx. ∑y∈UNIV. P y * «λz. z = y» x) x
    by(simp)
  also { from sP have (∑y∈UNIV. t (λx. P y * «λz. z = y» x)) =
                      (∑y∈UNIV. P y * t «λz. z = y» x)
                    by(subst additive-setsum, simp-all add:assms)
  }
  also from sP
  have (∑y∈UNIV. t (λx. P y * «λz. z = y» 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 –
3.2. EXPECTATION TRANSFORMERS

from assms
have \( t P x = (\sum_{y \in \text{UNIV}} P y * t \langle \lambda z. z = y \rangle x ) \)
  by (rule additive-delta-split)
also 
  have \( \text{UNIV} = \{ s. G s \} \cup \{ s. \neg G s \} \)
    by (auto)
  hence \( (\sum_{y \in \text{UNIV}} P y * t \langle \lambda z. z = y \rangle x ) = \)
    \( (\sum_{y \in \{ s. G s \}} P y * t \langle \lambda z. z = y \rangle x ) + \)
    \( (\sum_{y \in \{ s. \neg G s \}} P y * t \langle \lambda z. z = y \rangle x ) \)
    by (auto intro setsum.union-disjoint)
finally show \(?\)thesis .
qed

Maximality

**definition**

maximal :: (('a ⇒ real) ⇒ 'a ⇒ real) ⇒ bool

**where**

\( \text{maximal} \ t \equiv \forall c. \ 0 \leq c \rightarrow t \langle \lambda -. c \rangle = (\lambda -. c) \)

**lemma maximalI[ intro]:**

\[ \forall c. \ 0 \leq c \Rightarrow t \langle \lambda -. c \rangle = (\lambda -. c) \] \( \Rightarrow \) maximal \( t \)

by (simp add: maximal-def)

**lemma maximalD[ dest]:**

\[ \text{maximal} \ t; \ 0 \leq c \] \( \Rightarrow \) \( t \langle \lambda -. c \rangle = (\lambda -. c) \)

by (simp add: maximal-def)

A transformer that is both additive and maximal is deterministic:

**definition**

determ :: (('a ⇒ real) ⇒ 'a ⇒ real) ⇒ bool

**where**

\( \text{determ} \ t \equiv \text{additive} \ t \wedge \text{maximal} \ t \)

**lemma determI[ intro]:**

\[ \text{additive} \ t; \ \text{maximal} \ t \] \( \Rightarrow \) determ \( t \)

by (simp add: determ-def)

**lemma determ-additiveD[ intro]:**

determ \( t \) \( \Rightarrow \) additive \( t \)

by (simp add: determ-def)

**lemma determ-maximalD[ intro]:**

determ \( t \) \( \Rightarrow \) 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 «N P» s = 1`

  **Proof**
  - `have t «P» s + t «N P» s = t (λs. «P» s + «N P» s) s`
  - `by(simp add: additiveD determ determ-additiveD)`
  - `also`
    - `have P s. (λs. «P» s + «N P» s) R s = 1`
    - `by(case-tac P s, simp-all)`
    - `hence t (λs. «P» s + «N P» s) = t (λs. 1)`
    - `by(simp)`
  - `also have t (λs. 1) = (λs. 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 ⊢ t R`
  - and `wp2`: `Q ⊢ t S`
  - and `sc`: `sub-conj t`
  - and `sR`: `sound R`
  - and `sS`: `sound S`
- **Shows** `P & Q ⊢ t (R & S)`

  **Proof**
  - `from wp1 and wp2 have P & Q ⊢ t R & t S`
    - `by(blast intro: entails-frame)`
  - `also from sc and sR and sS have ... ≤ t (R & S)`
    - `by(rule sub-conjD)`
- **Finally show** `thesis`.

**QED**

These allow mismatched results to be composed

**Lemma** `entails-strengthen-post`:

- `[ P ⊢ t Q; healthy t; sound R; Q ⊢ R; sound Q ] → P ⊢ t R`
  - `by(blast intro: entails-trans)`
3.2. EXPECTATION TRANSFORMERS

**lemma** entails-weaken-pre:
\[
\left[\ P \vdash t \ R; \ P \vdash Q \right] \Longrightarrow P \vdash t \ R
\]
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: \( P \vdash Q \) and h: healthy t
and \( sQ \): sound Q and pos: \( 0 \leq c \)
shows \((\lambda s. c \ast P s) \vdash t (\lambda s. c \ast Q s)\)

**proof** (rule le-funI)
  fix s
  from pos and wp have \( c \ast P s \leq c \ast t Q s \)
  by (auto intro: mult-left-mono)
  with \( sQ \) pos h show \( c \ast P s \leq t (\lambda s. c \ast Q s) s \)
  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: \( \forall s. P s \Longrightarrow 1 \leq t \ Q s \)
shows \( P \vdash t \ Q \)

**proof** (rule entailsI)
  fix s show \( P s \leq t \ Q s \)
  proof (cases P s)
  case True with wp show \?thesis by (auto)
next
  case False with h show \?thesis by (auto)
qed

The other direction works too.

**lemma** fold-premise:
assumes ht: healthy t
and wp: \( P \vdash Q \)
shows \( \forall s. P s \longrightarrow 1 \leq t \ Q s \)

**proof** (clarify)
  fix s assume P s
  hence \( 1 = P s \) by (simp)
  also from wp have \( ... \leq t \ Q s \) by (auto)
  finally show \( 1 \leq t \ Q s \).
qed
Predicate conjunction behaves as expected:

**Lemma** *conj-post*:

\[
\begin{aligned}
P &\vdash t \langle \lambda s. Q s \land R s \rangle; \text{healthy } t \implies P \vdash t \langle Q \rangle \\
\text{by (blast intro: entails-strengthen-post implies-entails)}
\end{aligned}
\]

Similar to \[
\begin{aligned}
\text{healthy } ?t; \bigwedge s. ?P s \implies 1 \leq ?t \langle ?Q \rangle s \implies \langle ?P \rangle ?t \langle ?Q \rangle,
\end{aligned}
\]

but more general.

**Lemma** *entails-pconj-assumption*:

- **assumes** \( f: \text{feasible } t \) and \( wP: \bigwedge s. P s \implies Q s \leq t R s \)
- **and** \( uQ: \text{unitary } Q \) and \( uR: \text{unitary } R \)
- **shows** \( \langle P \rangle \land \langle Q \rangle \vdash t R \)
- **unfolding** exp-conj-def

**Proof** (rule entailsI)

- **fix** \( s \) and show \( \langle P \rangle s \land \langle Q \rangle s \leq t R s \)
- **proof** (cases \( P s \))
  - **case** True
    - moreover from \( uQ \) have \( 0 \leq Q s \) by (auto)
    - ultimately show \( \Theta \) by (simp add: conj-lone wP)
  - **next**
  - **case** False
    - moreover from \( uQ \) have \( Q s \leq 1 \) by (auto)
    - ultimately show \( \Theta \) using assms by (simp, blast)
- qed
- qed

### 3.3 Induction

**theory** *Induction*

- **imports** *Expectations Transformers Conditionally-Complete-Lattices*
  - **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 `/src/HOL/Inductive.thy`), is that we do not have a complete lattice.

Finding a lower bound is easy (it’s \( \lambda s. 0 \cdot \cdot \cdot b \)), 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 \( \forall P. \ sound \ P \rightarrow \ healthy \ (t \ P) \) and \( \forall Q. \ sound \ Q \rightarrow \ healthy \ (\lambda P. \ t \ P \ Q) \). Thus by feasibility, \( t \ P \ Q \) must be bounded by
bound-of \( P \). Thus, as by definition \( x \leq t \ P \ x \) for any fixed point, all must
lie in the set of sound expectations bounded above by \( \lambda \cdot \ bound-of \ P \).

**definition** Inf-exp :: 's expect set ⇒ 's expect
**where** Inf-exp \( S = (\lambda s. \ Inf \ \{ f s \mid f \in S \}) \)

**lemma** Inf-exp-lower:
\[
[ \ P \in S; \ \forall P \in S. \ nneg P \ ] \Rightarrow \ Inf-exp S \leq P
\]
**unfolding** Inf-exp-def
by(intro le-funI cInf-lower bdd-belowI[where \( m=0 \)], auto)

**lemma** Inf-exp-greatest:
\[
[ \ S \neq \{\}; \ \forall P \in S. \ Q \leq P \ ] \Rightarrow \ Q \leq Inf-exp S
\]
**unfolding** Inf-exp-def by(auto intro le-funI cInf-greatest)

**definition** Sup-exp :: 's expect set ⇒ 's expect
**where** Sup-exp \( S = (if \ S = \{} \ then \ \lambda s. \ 0 \ else \ (\lambda s. \ Sup \ \{ f s \mid f \in S \})) \)

**lemma** Sup-exp-upper:
\[
[ \ P \in S; \ \forall P \in S. \ bounded-by b P \ ] \Rightarrow \ P \leq Sup-exp S
\]
**unfolding** Sup-exp-def
by(cases \( S=\{\} \), simp-all, intro le-funI cSup-upper bdd-aboveI[where \( M=b \)], auto)

**lemma** Sup-exp-sound:
\[
\ assumes \ sS: \ \forall P. \ P \in S \Rightarrow \ sound \ P
\quad \text{and} \ bS: \ \forall P. \ P \in S \Rightarrow \ bounded-by b P
\quad \text{shows} \ \ sound \ (Sup-exp S)
\]
**proof**(cases \( S=\{\} \), simp add: Sup-exp-def, blast,
intro sound12 bounded-by12 nneg12)
assume neS: \( S \neq \{\} \)
then obtain \( P \) where Pin: \( P \in S \) by(auto)
with \( sS \ bS \) have \( nP: \ nneg P \ bounded-by b P \) by(auto)
hence nb: \( 0 \leq b \) by(auto)

from \( bS \ nb \) show Sup-exp \( S \vdash \lambda s. \ b \)
by(auto intro: Sup-exp-least)

from \( nP \) have \( \lambda s. \ 0 \vdash P \) by(auto)
also from Pin bS have P ⊨ \text{Sup-exp } S
by(auto intro; Sup-exp-upper)
finally show λs. 0 ⊨ \text{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-bound:
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-unitary:
(\⋀ P. unitary P ⇒ unitary (t P)) ⇒ bounded-by 1 (lfp-exp t)
by(auto intro:!: lfp-exp-lowerbound)

lemma lfp-exp-unitary1:
(\⋀ 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)
3.3. INDUCTION

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:
\( \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) \Rightarrow \text{bounded-by } 1 \) (gfp-exp t)
unfolding gfp-exp-def
by (rule bounded-by12[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 \( \lambda s. 0 \vdash \text{Sup-exp} \{ P. \text{unitary } P \land P \vdash t P \} \) 
by(simp only:empty Sup-exp-def, auto)

next

assume \( \{ P. \text{unitary } P \land P \vdash t P \} \neq \{ \} \) 
then obtain \( Q \vdash Q \in \{ P. \text{unitary } P \land P \vdash t P \} \) by(auto)

hence \( \lambda s. 0 \vdash Q \) by(auto)

also from \( Q \in \{ P. \text{unitary } P \land P \vdash t P \} \)
by(auto intro: Sup-exp-upper)

finally show \( \lambda s. 0 \vdash \text{Sup-exp} \{ P. \text{unitary } P \land P \vdash t P \} \).

qed

lemma gfp-exp-unitary:
\( (\forall P. \text{unitary } P \Rightarrow \text{unitary } (t P)) \Rightarrow \text{unitary } (\text{gfp-exp } t) \)
by(iprover intro: gfp-exp-nneg gfp-exp-bound unitaryI2)

lemma gfp-exp-lemma2:

assumes \( \text{ft: } \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) \)
and \( \text{mt: } \forall P Q. \left[ \begin{array}{c}
\text{unitary } P; \text{unitary } Q; P \vdash Q \end{array} \right] \Rightarrow t P \vdash t Q \)

shows \( \text{gfp-exp } t \leq t (\text{gfp-exp } t) \)

proof(rule gfp-exp-least)

show unitary \( (t (\text{gfp-exp } t)) \) by(auto intro: gfp-exp-unitary ft)
fix \( P \)
assume \( \text{fp: } P \leq t P \) and \( uP; \text{unitary } P \)
with \( \text{ft} \) have \( P \leq \text{gfp-exp } t \) by(auto intro: gfp-exp-upperbound)
with \( uP \) gfp-exp-unitary \( \text{ft} \)
have \( t P \leq t (\text{gfp-exp } t) \) by(blast intro: mt)
with \( \text{fp} \) show \( P \leq t (\text{gfp-exp } t) \) by(auto)
qed

lemma gfp-exp-lemma3:

assumes \( \text{ft: } \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) \)
and \( \text{mt: } \forall P Q. \left[ \begin{array}{c}
\text{unitary } P; \text{unitary } Q; P \vdash Q \end{array} \right] \Rightarrow t P \vdash t Q \)

shows \( t (\text{gfp-exp } t) \leq \text{gfp-exp } t \)
by(iprover intro: gfp-exp-upperbound unitary-sound
  gfp-exp-unitary gfp-exp-lemma2 assms)

lemma gfp-exp-unfold:
\( (\forall P. \text{unitary } P \Rightarrow \text{unitary } (t P)) \Rightarrow (\forall P Q. \left[ \begin{array}{c}
\text{unitary } P; \text{unitary } Q; P \vdash Q \end{array} \right] \Rightarrow t P \vdash t Q) \Rightarrow \text{gfp-exp } t = t (\text{gfp-exp } t) \)
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, \textit{healthy} transformers do form a complete lattice, where the bottom element is \( \lambda s. 0::'c \), and the
3.3. INDUCTION

top element is the greatest allowed by feasibility: $\lambda P \cdot \text{bound-of } P$.

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

**lemma** Inf-trans-lower:  
\[
\begin{array}{c}
\forall t \in S; \forall u \in S. \forall P. \text{sound } P \rightarrow \text{sound } (u P) \\
\Rightarrow \text{le-trans } (\text{Inf-trans } S) \ t
\end{array}
\]
**unfolding** Inf-trans-def  
by (rule le-transI[of Inf-exp-lower], blast+)

**lemma** Inf-trans-greatest:  
\[
\begin{array}{c}
\forall t \in S; \forall u \in S. \forall P. \text{unitary } P \rightarrow \text{unitary } (u P) \\
\Rightarrow \text{le-utrans } t \ (\text{Inf-trans } S)
\end{array}
\]
**unfolding** Inf-trans-def  
by (auto intro: le-utransI[of Inf-exp-greatest])

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

**lemma** Sup-trans-upper:  
\[
\begin{array}{c}
\forall t \in S; \forall u \in S. \forall P. \text{nneg } P \rightarrow \text{nneg } (u P) \\
\Rightarrow \text{le-utrans } t \ (\text{Sup-trans } S)
\end{array}
\]
**unfolding** Sup-trans-def  
by (intro le-utransI[of Sup-exp-upper], auto intro:unitary-bound)

**lemma** Sup-trans-upper2:  
\[
\begin{array}{c}
\forall t \in S; \forall u \in S. \forall P. \text{nneg } P \rightarrow \text{nneg } (u P) \\
\Rightarrow \text{le-utrans } t \ (\text{Sup-trans } S)
\end{array}
\]
**unfolding** Sup-trans-def  
by (blast intro:Sup-exp-upper)

**lemma** Sup-trans-least:  
\[
\begin{array}{c}
\forall t \in S; \forall u \in S. \forall P. \text{nneg } P \rightarrow \text{nneg } (u P) \\
\Rightarrow \text{le-utrans } t \ (\text{Sup-trans } S)
\end{array}
\]
**unfolding** Sup-trans-def  
by (auto intro:Sup-exp-least)

**lemma** feasible-Sup-trans:  
**fixes** $S$: 's trans set  
**assumes** fS: $\forall t \in S. \text{feasible } t$  
**shows** feasible (Sup-trans $S$)  
**proof** (cases $S=\{\}$, simp add: Sup-trans-def Sup-exp-def, blast, intro feasibleI)  
**fix** $b: \text{real and } P$: 's expect  
**assume** $bP: \text{bounded-by } b \text{ } P$ and $nP: \text{nneg } P$  
and $neS: S \neq \{\}$
from ncS obtain t where \( \text{in}: t \in S \) by(auto)
with \( fS \) have \( f: \text{feasible} \ t \) by(auto)
with \( bP \ nP \) have \( \lambda s. \emptyset \vdash t \ P \) by(auto)
also {
from \( bP \ nP \) have \( \text{sound} \ P \) by(auto)
with \( \text{in} fS \) have \( t \ P \vdash \text{Sup-trans} \ S \ P \)
  by(auto intro!:Sup-trans-upper2)
}
finally show \( \text{nneq} \ (\text{Sup-trans} \ S \ P) \) by(auto)

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

definition \( \text{lfp-trans} \ :: \ (\text{‘s trans} \
\Rightarrow \text{‘s trans}) \Rightarrow \text{‘s trans} \)
where \( \text{lfp-trans} \ T = \text{Inf-trans} \ \{ t. \ (\forall P. \ \text{sound} \ P \Rightarrow \text{sound} \ (t \ P)) \wedge \text{le-trans} \ (T \ t) \} \)

lemma \( \text{lfp-trans-lowerbound} \):
  \[ \text{le-trans} \ (T \ t) \ t; \bigwedge P. \text{sound} \ P \Rightarrow \text{sound} \ (t \ P) \big] \Rightarrow \text{le-trans} \ (\text{lfp-trans} \ T) \ t \nunfolding lfp-trans-def
  by(auto intro:Inf-trans-lower)

lemma \( \text{lfp-trans-greatest} \):
  \[ \bigwedge t P. \big[ \text{le-trans} \ (T \ t) \ t; \bigwedge P. \text{sound} \ P \Rightarrow \text{sound} \ (t \ P) \big] \big] \Rightarrow \text{le-trans} \ u \ t ; \bigwedge P. \text{sound} \ P \Rightarrow \text{sound} \ (v \ P); \text{le-trans} \ (T \ v) \ v \big] \Rightarrow \text{le-trans} \ u \ (\text{lfp-trans} \ T) \nunfolding lfp-trans-def by(rule Inf-trans-greatest, auto)

lemma \( \text{lfp-trans-sound} \):
  fixes \( P \ Q::\text{‘s expect} \)
  assumes \( sP::\text{sound} \ P \)
    and \( fP::\text{le-trans} \ (T \ v) \ v \)
    and \( sP::\text{sound} \ P \Rightarrow \text{sound} \ (v \ P) \)
  shows \( \text{sound} \ (\text{lfp-trans} \ T \ P) \)
proof(intro soundI2 bounded-byI2 nneqI2)
from \( fP \ sP \) have \( \text{le-trans} \ (\text{lfp-trans} \ T) \ v \)
  by(iprover intro:lfp-trans-lowerbound)
with \( sP \) have \( \text{lfp-trans} \ T \ P \vdash v \ P \) by(auto)
also {
from \( sP \) have \( \text{sound} \ (v \ P) \) by(iprover)
  hence \( v \ P \vdash \lambda s. \text{bound-of} \ (v \ P) \) by(auto)
}
finally show \( \text{lfp-trans} \ T \ P \vdash \lambda s. \text{bound-of} \ (v \ P) \)

have \( \text{le-trans} \ (\lambda P \ s. \emptyset) \ (\text{lfp-trans} \ T) \)
proof(intro lfp-trans-greatest)
3.3. INDUCTION

\[\text{fix } t::'s \text{ trans} \]
\[\text{assume } \bigwedge P. \text{ sound } P \implies \text{ sound } (t P) \]
\[\text{hence } \bigwedge P. \text{ sound } P \implies \lambda s. 0 + t P \text{ by(auto)} \]
\[\text{thus } le-trans (\lambda P s. 0) t \text{ by(auto)} \]
\[\text{next} \]
\[\text{fix } P::'s \text{ expect} \]
\[\text{assume } \text{ sound } P \text{ thus } \text{ sound } (v P) \text{ by(rule } sv) \]
\[\text{next} \]
\[\text{show } le-trans (T v) v \text{ by(rule } fv) \]
\[\text{qed} \]
\[\text{with } sP \text{ show } \lambda s. 0 + \text{ lfp-trans } T P \text{ by(auto)} \]
\[\text{qed} \]

**lemma lfp-trans-unitary:**
\[\text{fixes } P Q::'s \text{ expect} \]
\[\text{assumes } uP: \text{ unitary } P \]
\[\text{and } fv: \text{ le-trans } (T v) v \]
\[\text{and } sv: \bigwedge P. \text{ sound } P \implies \text{ sound } (v P) \]
\[\text{and } T: \text{ le-trans } (T (\lambda P s. \text{ bound-of } P)) (\lambda P s. \text{ bound-of } P) \]
\[\text{shows } \text{ unitary } (\text{lfp-trans } T P) \]
\[\text{proof(rule unitaryI)} \]
\[\text{from } \text{ unitary-sound[OF } uP \text{] } fv \text{ } sv \text{ show } \text{ sound } (\text{lfp-trans } T P) \]
\[\text{by(rule lfp-trans-sound)} \]
\[\text{show } \text{ bounded-by } 1 (\text{lfp-trans } T P) \]
\[\text{proof(rule bounded-byI2)} \]
\[\text{from } T \text{ have } \text{ le-trans } (\text{lfp-trans } T) (\lambda P s. \text{ bound-of } P) \]
\[\text{by(auto intro: lfp-trans-lowerbound)} \]
\[\text{with } uP \text{ have } \text{lfp-trans } T P \vdash \lambda s. \text{ bound-of } P \text{ by(auto)} \]
\[\text{also from } uP \text{ have } \ldots \vdash \lambda s. 1 \text{ by(auto)} \]
\[\text{finally show } \text{lfp-trans } T P \vdash \lambda s. 1. \]
\[\text{qed} \]
\[\text{qed} \]

**lemma lfp-trans-lemma2:**
\[\text{fixes } v::'s \text{ trans} \]
\[\text{assumes mono: } \bigwedge t u. [ \text{ le-trans } t u; \bigwedge P. \text{ sound } P \implies \text{ sound } (t P); \]
\[\bigwedge P. \text{ sound } P \implies \text{ sound } (u P) ] \implies \text{ le-trans } (T t) (T u) \]
\[\text{and } nT: \bigwedge t P. [ \bigwedge Q. \text{ sound } Q \implies \text{ sound } (t Q); \text{ sound } P ] \implies \text{ sound } (T t P) \]
\[\text{and } fv: \text{ le-trans } (T v) v \]
\[\text{and } sv: \bigwedge P. \text{ sound } P \implies \text{ sound } (v P) \]
\[\text{shows le-trans } (T (\text{lfp-trans } T)) (\text{lfp-trans } T) \]
\[\text{proof(rule lfp-trans-greatest[where } T=T \text{ and } v=v], \text{ simp-all add:assms)} \]
\[\text{fix } t::'s \text{ trans and } P::'s \text{ expect} \]
\[\text{assume } ft: \text{ le-trans } (T t) t \text{ and } st: \bigwedge P. \text{ sound } P \implies \text{ sound } (t P) \]
\[\text{hence } \text{ le-trans } (\text{lfp-trans } T) t \text{ by(auto intro:lfp-trans-lowerbound)} \]
\[\text{with } ft \text{ st have } \text{ le-trans } (T (\text{lfp-trans } T)) (T t) \]
\[\text{by(irove intro:mono lfp-trans-sound fe sv)} \]
also note \( f_{t} \)

finally show \( \text{le-trans} \ (T \ (\text{lfp-trans} \ T)) \ t \).

qed

\[ \text{lemma } \text{lfp-trans-lemma}3: \]
\[
\text{fixes } v::'s \ trans
\]
\[
\text{assumes } \text{mono}: \forall t. u. \ [ \text{le-trans} \ t \ u; \ \forall P. \ \text{sound} \ P \Longrightarrow \ \text{sound} \ (t \ P); \ \forall P. \ \text{sound} \ P \Longrightarrow \ \text{sound} \ (u \ P) ] \Longrightarrow \ \text{le-trans} \ (T \ t) \ (T \ u)
\]
\[
\text{and } sT: \ \forall t. P. [ \forall Q. \ \text{sound} \ Q \Longrightarrow \ \text{sound} \ (t \ Q); \ \text{sound} \ P ] \Longrightarrow \ \text{sound} \ (T \ t \ P)
\]
\[
\text{and } f_{v}: \ \text{le-trans} \ (T \ v) \ v
\]
\[
\text{and } s_{v}: \ \forall P. \ \text{sound} \ P \Longrightarrow \ \text{sound} \ (v \ P)
\]
\[
\text{shows } \text{le-trans} \ (\text{lfp-trans} \ T) \ (T \ (\text{lfp-trans} \ T))
\]
\[ \text{proof (rule } \text{lfp-trans-lowerbound)} \]
\[
\text{fix } P::'s \ expect
\]
\[
\text{assume } sP: \ \text{sound} \ P
\]
\[
\text{have } n1: \ \forall P. \ \text{sound} \ P \Longrightarrow \ \text{sound} \ (\text{lfp-trans} \ T \ P)
\]
\[
\text{by (iprover intro: lfp-trans-sound } f_{v} \ s_{v})
\]
\[
\text{with } sP \text{ have } n2: \ \text{sound} \ (\text{lfp-trans} \ T \ P)
\]
\[
\text{by (iprover intro: lfp-trans-sound } f_{v} \ s_{v} \ sT)
\]
\[
\text{with } n1 \ sP \text{ show } n3: \ \text{sound} \ (T \ (\text{lfp-trans} \ T) \ P)
\]
\[
\text{by (iprover intro: } sT)
\]
\[
\text{next}
\]
\[
\text{show } \text{le-trans} \ (T \ (T \ (\text{lfp-trans} \ T))) \ (T \ (\text{lfp-trans} \ T))
\]
\[
\text{by (rule mono[OF lfp-trans-lemma2, OF mono], (iprover intro:assms lfp-trans-sound)+)}
\]
\[ \text{qed} \]

\[ \text{lemma } \text{lfp-trans-unfold:} \]
\[
\text{fixes } P::'s \ expect
\]
\[
\text{assumes } \text{mono}: \forall t. u. \ [ \text{le-trans} \ t \ u; \ \forall P. \ \text{sound} \ P \Longrightarrow \ \text{sound} \ (t \ P); \ \forall P. \ \text{sound} \ P \Longrightarrow \ \text{sound} \ (u \ P) ] \Longrightarrow \ \text{le-trans} \ (T \ t) \ (T \ u)
\]
\[
\text{and } sT: \ \forall t. P. [ \forall Q. \ \text{sound} \ Q \Longrightarrow \ \text{sound} \ (t \ Q); \ \text{sound} \ P ] \Longrightarrow \ \text{sound} \ (T \ t \ P)
\]
\[
\text{and } f_{v}: \ \text{le-trans} \ (T \ v) \ v
\]
\[
\text{and } s_{v}: \ \forall P. \ \text{sound} \ P \Longrightarrow \ \text{sound} \ (v \ P)
\]
\[
\text{shows } \text{equiv-trans} \ (\text{lfp-trans} \ T) \ (T \ (\text{lfp-trans} \ T))
\]
\[
\text{by (rule } \text{lfp-trans-sound-antisym, (rule lfp-trans-lemma2[OF mono], (iprover intro:assms lfp-trans-sound)+), (rule lfp-trans-lemma3[OF mono], (iprover intro:assms lfp-trans-sound)+))}
\]
\[ \text{definition } \text{gfp-trans} :: (+'s \ trans \Rightarrow 's \ trans) \Rightarrow 's \ trans
\]
\[
\text{where } \text{gfp-trans} \ T = \text{Sup-trans} \ \{ t. (\forall P. \ \text{unitary} \ P \Longrightarrow \ \text{unitary} \ (t \ P)) \land \ \text{le-utrans} \ t \ (T \ t) \}
\]

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

**Lemma gfp-trans-least:**

\[
\forall t. \left( le-utrans \ t \ (T \ t); \bigwedge_P. \text{unitary } P \implies \text{unitary } (t \ P) \right) \implies le-utrans \ t \ u
\]

Unfolding **gfp-trans-def** by (auto intro: Sup-trans-least)

**Lemma gfp-trans-unitary:**

fixes \( P :: \text{'s expect} \)

assumes \( uP : \text{unitary } P \)

shows \( \text{unitary } (gfp-trans \ T \ P) \)

proof (intro unitaryI2 nnegI2 bounded-byI2)

show \( \forall P. \text{unitary } P \implies \text{unitary } (u \ P) \)

Unfolding **gfp-trans-def Sup-trans-def**

proof (rule Sup-exp-least, clarify)

fix \( t :: \text{trans} \)

assume \( \forall P. \text{unitary } P \implies \text{unitary } (t \ P) \)

with \( uP \)

have \( \text{unitary } (t \ P) \)

by (auto)

thus \( t \ P \vdash \lambda s. 1 \)

by (auto)

next

show \( \text{nneg } (\lambda s. 1 :: \text{real}) \)

by (auto)

qed

let \( ?S = \{ t \ P \mid t \in \{ t, (\forall P. \text{unitary } P \implies \text{unitary } (t \ P)) \land le-utrans \ t \ (T \ t)\} \}

show \( \lambda s. 0 \vdash \text{gfp-trans } T \ P \)

Unfolding **gfp-trans-def Sup-trans-def**

proof (cases)

assume \( \emptyset \)

show \( \lambda s. 0 \vdash \text{Sup-exp } ?S \)

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 \( \text{unitary } Q \)

by (auto)

hence \( \lambda s. 0 \vdash Q \)

by (auto)

also with \( uP \)

have \( \lambda s. 0 \vdash \text{Sup-exp } ?S \)

proof (intro Sup-exp-upper, blast, clarify)

fix \( t :: \text{trans} \)

assume \( \forall Q. \text{unitary } Q \implies \text{unitary } (t \ Q) \)

with \( uP \)

show \( \text{bounded-by } 1 \ (t \ P) \)

by (auto)

qed

finally show \( \lambda s. 0 \vdash \text{Sup-exp } ?S \).

qed

qed

**Lemma gfp-trans-lemma2:**

assumes \( \text{mono} : \bigwedge t. \left( le-utrans \ t \ u; \bigwedge P. \text{unitary } P \implies \text{unitary } (t \ P); \bigwedge_P. \text{unitary } P \implies \text{unitary } (u \ P) \right) \implies le-utrans \ (T \ t) \ (T \ u) \)

and \( hT : \bigwedge t. \left[ \bigwedge Q. \text{unitary } Q \implies \text{unitary } (t \ Q); \text{unitary } P \right] \implies \text{unitary } (T \ t \ P) \)
CHAPTER 3. SEMANTIC STRUCTURES

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: \( \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) \)

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: \( \forall t u. \bigl[ \\forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) ; \bigr] \)
  \( \forall P. \text{unitary } P \Rightarrow \text{unitary } (u P) \bigr] \Rightarrow \text{le-utrans } (T t) (T u) \)
  and hT: \( \forall t P. \bigr[ \\forall Q. \text{unitary } Q \Rightarrow \text{unitary } (t Q) ; \text{unitary } P \bigr] \Rightarrow \text{unitary } (t P) \)
  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: \( \forall t u. \bigl[ \\forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) ; \bigr] \)
  \( \forall P. \text{unitary } P \Rightarrow \text{unitary } (u P) \bigr] \Rightarrow \text{le-utrans } (T t) (T u) \)
  and hT: \( \forall t P. \bigr[ \\forall Q. \text{unitary } Q \Rightarrow \text{unitary } (t Q) ; \text{unitary } P \bigr] \Rightarrow \text{unitary } (t P) \)
  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: \( \forall u P. \text{unitary } P \Rightarrow T u P = t P (u P) \)
  and ft: \( \forall t P. \bigr[ \forall Q. \text{unitary } Q \Rightarrow \text{unitary } (t Q) ; \text{unitary } P \bigr] \Rightarrow \text{unitary } (T t P) \)
  and ft: \( \forall P Q. \text{unitary } P \Rightarrow \text{unitary } Q \Rightarrow \text{unitary } (t P Q) \)
  and mt: \( \forall P Q R. \bigl[ \text{unitary } P ; \text{unitary } Q ; \text{unitary } R \bigr] \Rightarrow t P Q R \)

Q ⊢ t P R
and uP: \( \text{unitary } P \)
and monoT: \( \forall t u. \bigl[ \\forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) ; \bigr] \)
\( \forall P. \text{unitary } P \Rightarrow \text{unitary } (u P) \bigr] \Rightarrow \text{le-utrans } (T t) (T u) \)

shows gfp-trans T P = gfp-exp (t P) (is ?X P = ?Y P)
3.3. INDUCTION

proof (rule antisym)

show ?X P ≤ ?Y P

proof (rule gfp-exp-upperbound)

from mono T T 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 \( R \implies \) 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 \( R \implies \) 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: \( \forall u P. \) sound \( P \) ⇒ \( T u P = t P (u P) \)

and st: \( \forall P Q. \) sound \( P \) ⇒ sound \( Q \) ⇒ sound (t P Q)

and mt: \( \forall P. \) sound \( P \) ⇒ mono-trans (t P)

and monoT: \( \forall a u. \) [ le-trans t u; \( \forall P. \) sound \( P \) ⇒ sound (t P); \( \forall P. \) sound \( P \) ⇒ sound (u P) ] ⇒ le-trans (T t) (T u)

and aT: \( \forall t P. \) \( \forall Q, \) sound \( Q \) ⇒ sound (t Q); sound \( P \) \] ⇒ sound (T t P)

and fs: le-trans (T v) v

and sv: \( \forall P. \) 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-utransD [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 \( \forall P. \text{sound } P \implies t P (v P) = T v P \) by (simp add: tailcall)
also have \( \forall P. \text{sound } P \implies \ldots P \vdash v P \) by (auto intro: le-transD[OF fe])
finally have \( \text{fvP}: \forall P. \text{sound } P \implies t P (v P) \vdash v P \).
have \( \text{suP}: \forall P. \text{sound } P \implies \text{sound } (v P) \) by (rule sv)

show \( \exists X P \leq \exists Y P \)
proof (rule le-transD[OF lfp-trans-lowerbound, OF - - sP]
  show le-trans \( T (\lambda a. \text{lfp-exp } (t a)) \) \( (\lambda a. \text{lfp-exp } (t a)) \)
proof (rule le-transI)
fix \( P::'s expect \)
assume \( sP \)
from \( sP \) have \( \lambda a. \text{lfp-exp } (t a)) P = \text{lfp-exp } (t P) \) by (rule tailcall)
also have \( \text{lfp-exp } (t P) = \text{lfp-exp } (t P) \)
by (iprover intro: lfp-exp-unfold[symmetric] sP st mt fvP svP)
finally show \( \lambda a. \text{lfp-exp } (t a)) P \vdash \text{lfp-exp } (t P) \) by (simp)
qed

definition \( \text{Inf-utrans } :: 's trans set \Rightarrow 's trans \)
where \( \text{Inf-utrans } S = (\text{if } S = \{\} \text{ then } \lambda P s. 1 \text{ else } \text{Inf-trans } S) \)

lemma \( \text{Inf-utrans-lower} \):
[ \( t \in S; \forall t \in S. \forall P. \text{unitary } P \implies \text{unitary } (t P) \) ] \( \implies \) \( \text{le-utrans } (\text{Inf-utrans } S) t \)
unfolding \( \text{Inf-utrans-def} \)
by (cases \( S=\{\}, \)
  auto intro!: le-utransI Inf-exp-lower sound-nneg unitary-sound
  simp: Inf-trans-def)

lemma \( \text{Inf-utrans-greatest} \):
[ \( \forall P. \text{unitary } P \implies \text{unitary } (t P); \forall u \in S. \text{le-utrans } t u \) ] \( \implies \) \( \text{le-utrans } t \)
(\text{Inf-utrans } S)
unfolding \( \text{Inf-utrans-def Inf-trans-def} \)
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 ⇒ ('s ⇒ real) ⇒ ('s ⇒ real)

Abort either always fails, λP s. 0::'c, or always succeeds, λP s. 1::'c.

definition Abort :: 's prog where Abort ≡ λab P s. if ab then 0 else 1

Skip does nothing at all.

definition Skip :: 's prog where Skip ≡ λab P. P

Apply lifts a state transformer into the space of programs.

definition Apply :: ('s ⇒ 's) ⇒ 's prog where Apply f ≡ λab P s. P (f s)

Seq is sequential composition.

definition Seq :: 's prog ⇒ 's prog ⇒ 's prog (infixl :: 59) where Seq a b ≡ (λab. a ab o b ab)

PC is probabilistic choice between programs.

definition PC :: 's prog ⇒ ('s ⇒ real) ⇒ 's prog ⇒ 's prog (- ⨿ - [58,57,57] 57) where PC a P b ≡ λab 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 ≡ λab 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 ≡ λab 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 = (λab Q s. t)
```

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

```
definition Mu :: ('s prog ⇒ 's prog) ⇒ 's prog
where   Mu T ≡ (λab Q s. if ab then lfp-trans (λt. T (Embed t) ab) else gfp-trans (λt. 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 ≡ λab P s. Inf ((λa. f a ab P s) ' S s)
```

```
syntax -SetDC :: pttrn >= ('s => 'a set) => 's prog >= 's prog
(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 ≡ λab P s. ∑a∈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
```

76  CHAPTER 4. THE PGCL LANGUAGE
where
\[
\text{Bind } g f \, ab \equiv \lambda P \, s. \, \text{let } a = g \, s \text{ in } f \, a \, ab \, P \, s
\]

This gives us something like let syntax

**syntax** 
- **-Bind** :: \(pttrn \Rightarrow (s \Rightarrow 'a) \Rightarrow 's \ prog \Rightarrow 's \ prog\)  
  \(- \ is \ - in - [55,55,55,55]\)

**translations** 
- \(x \ is \ f \ in \ a \Rightarrow \text{CONST } \text{Bind } f \ (\%x. \ a)\)

**definition** 
- **flip** :: \(('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'a \Rightarrow 'c\)
  \[\text{where } [\text{simp}]: \text{flip } f = (\lambda b \ a. \ f \ a \ b)\]

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

**syntax** 
- **-PBind** :: \(pttrn \Rightarrow (s \Rightarrow \text{real}) \Rightarrow 's \ prog \Rightarrow 's \ prog\)  
  \(- \ is \ - at - in - [55,55,55,55]\)

**translations** 
- \(\text{bind } x \ at \ p \ in \ a \Rightarrow \text{CONST SetPC } (\%x. \ a) \ (\text{CONST } \text{flip } (\%x. \ p))\)

**syntax** 
- **-DBind** :: \(pttrn \Rightarrow (s \Rightarrow 'a \ \text{set}) \Rightarrow 's \ prog \Rightarrow 's \ prog\)  
  \(- \ is \ - from - in - [55,55,55,55]\)

**translations** 
- \(\text{bind } x \ from \ S \ in \ a \Rightarrow \text{CONST SetDC } (\%x. \ a) \ S\)

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

**syntax** 
- **-assign** :: \ident \Rightarrow 'a \Rightarrow 's \ prog\  
  
**ML**  
\[
\text{fun } \text{assign-tr} - [\text{Const } (\text{name, -}), \ \text{arg}] = \\
\quad \text{Const } (\text{Embedding.Apply, dummyT}) \ \text{\$ Abs } (s, \text{dummyT}, \\
\quad \quad \text{Syntax.const } (\text{suffix Record.updateN } \text{name}) \ \text{\$ Abs } (\text{Name.uu-, dummyT, arg } \text{\$ Bound 1}) \ \text{\$ Bound 0}) \\
\quad | \text{assign-tr} - \text{ts} = \text{raise TERM } (\text{assign-tr}, \text{ts})
\]

**parse-translation**  
\[
\langle \langle [@\{\text{syntax-const -assign\}, \text{assign-tr}\}] \rangle \rangle
\]

**syntax** 
- **-SetPC** :: \ident \Rightarrow (s \Rightarrow 'a \Rightarrow \text{real}) \Rightarrow 's \ prog\  
  \(- \ is \ - at - [66,66,66]\)

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

**syntax**

```plaintext
-set-dc :: ident => ('s => 'a set) => 's prog (- :∈ - [66,66])
```

**ML**

```plaintext
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**

```plaintext
-set-dc-UNIV :: ident => 's prog (any - [66])
```

**translations**

```plaintext
-set-dc-UNIV x => -set-dc x (%.- CONST UNIV)
```

**definition**

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

**definition**

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

If-Then-Else as a degenerate probabilistic choice.

**abbreviation**

```plaintext
if-then-else :: ['s ⇒ bool, 's prog, 's prog] ⇒ 's prog
(If - Then - Else - 58)
where
If P Then a Else b ≡ a * P * b ⊕ b
```

Syntax for loops

**abbreviation**

```plaintext
do-while :: ['s ⇒ bool, 's prog] ⇒ '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:**
\[ \text{wp} \text{ Abort} \ P = (\lambda s. 0) \]
*unfolding* \text{wp-def Abort-def} *by* (simp)

**Lemma eval-wlp-Abort:**
\[ \text{wlp} \text{ Abort} \ P = (\lambda s. 1) \]
*unfolding* \text{wlp-def Abort-def} *by* (simp)

**Lemma eval-wp-Skip:**
\[ \text{wp} \text{ Skip} \ P = P \]
*unfolding* \text{wp-def Skip-def} *by* (simp)

**Lemma eval-wlp-Skip:**
\[ \text{wlp} \text{ Skip} \ P = P \]
*unfolding* \text{wlp-def Skip-def} *by* (simp)

**Lemma eval-wp-Apply:**
\[ \text{wp} (\text{Apply} f) \ P = P \circ f \]
*unfolding* \text{wp-def Apply-def} *by* (simp add: o-def)

**Lemma eval-wlp-Apply:**
\[ \text{wlp} (\text{Apply} f) \ P = P \circ f \]
*unfolding* \text{wlp-def Apply-def} *by* (simp add: o-def)

**Lemma eval-wp-Seq:**
\[ \text{wp} (a ; b) \ P = (\text{wp} a \circ \text{wp} b) \]
*unfolding* \text{wp-def Seq-def} *by* (simp)

**Lemma eval-wlp-Seq:**
\[ \text{wlp} (a ; b) \ P = (\text{wlp} a \circ \text{wlp} b) \]
*unfolding* \text{wlp-def Seq-def} *by* (simp)

**Lemma eval-wp-PC:**
\[ \text{wp} (a \ Q \oplus b) \ P = (\lambda s. Q s * \text{wp} a \ P s + (1 - Q s) * \text{wp} b \ P s) \]
*unfolding* \text{wp-def PC-def} *by* (simp)

**Lemma eval-wlp-PC:**
\[ \text{wlp} (a \ Q \oplus b) \ P = (\lambda s. Q s * \text{wlp} a \ P s + (1 - Q s) * \text{wlp} b \ P s) \]
*unfolding* \text{wlp-def PC-def} *by* (simp)

**Lemma eval-wp-DC:**
\[ \text{wp} (a \bigcap b) \ P = (\lambda s. \text{min} (\text{wp} a \ P s) (\text{wp} b \ P s)) \]
*unfolding* \text{wp-def DC-def} *by* (simp)

**Lemma eval-wlp-DC:**
\[ \text{wlp} (a \bigcap b) \ P = (\lambda s. \text{min} (\text{wlp} a \ P s) (\text{wlp} b \ P s)) \]
*unfolding* \text{wlp-def DC-def} *by* (simp)
lemma eval-wp-AC:
wp (a ∪ b) P = (λs. max (wp a P s) (wp b P s))
unfolding wp-def AC-def by(simp)

lemma eval-wlp-AC:
wlp (a ∪ b) P = (λs. 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 ((λa. 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 ((λa. wlp (p a) R s) ' S s)
unfolding wlp-def SetDC-def by(simp)

lemma eval-wp-SetPC:
wp (SetPC f p) P = (λs. ∑ a∈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 = (λs. ∑ a∈supp (p s). p s a * wlp (f a) P s)
unfolding wlp-def SetPC-def by(simp)

lemma eval-wp-Mu:
wp (µ t. T t) = lfp-trans (λt. wp (T (Embed t)))
unfolding wp-def Mu-def by(simp)

lemma eval-wlp-Mu:
wlp (µ t. T t) = gfp-trans (λt. wlp (T (Embed t)))
unfolding wlp-def Mu-def by(simp)

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

lemma eval-wlp-Bind:
wlp (Bind g f) = (λP 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
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
**CHAPTER 4. THE PGCL LANGUAGE**

**Lemma** nearly-healthy-wlp-Abort:

\[ \text{nearly-healthy} (\text{wlp} \ \text{Abort}) \]

**Proof** (rule nearly-healthyI)

fix \( P : 's \Rightarrow \text{real} \)

show unitary (wlp Abort \( P \))

by (simp add: wp-eval)

next

fix \( P, Q : 's \) expect

assume \( P \vdash Q \) and unitary \( P \) and unitary \( Q \)

thus wlp Abort \( P \vdash \text{wlp} \text{Abort} \( Q \)

unfolding wp-eval by (blast)

qed

**Lemma** healthy-wp-Skip:

\[ \text{healthy} (\text{wp} \ \text{Skip}) \]

by (force intro !: healthy-parts simp [wp-eval])

**Lemma** nearly-healthy-wlp-Skip:

\[ \text{nearly-healthy} (\text{wlp} \ \text{Skip}) \]

by (auto simp [wp-eval])

**Lemma** healthy-wp-Seq:

fixes \( t : 's \text{ prog} \) 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 \( P : 's \Rightarrow \text{real} \)

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 \( P : 's \Rightarrow \text{real} \) and \( Q \)

assume sound \( P \) and sound \( Q \) and \( P \vdash Q \)

with hu have sound (wp \( u \) \( P \)) and sound (wp \( u \) \( Q \))

and wp \( u \) \( P \vdash \text{wp} \text{u} \( Q \) by (auto)

with ht show wp \( t \) (wp \( u \) \( P \)) \( \vdash \text{wp} \text{t} \) (wp \( u \) \( Q \)) by (auto)

next

fix \( P : 's \Rightarrow \text{real} \) and \( c :: \text{real} \) and \( s \)

assume pos: \( 0 \leq c \) and \( sP : \text{ sound} \ P \)

with ht and hu have \( c * \text{wp} \ t (\text{wp} \ u \ P) \ s = \text{wp} \ t (\lambda s. c * \text{wp} \ u \ P \ s) \ s \)

by (auto intro! scalingD)

also with hu and pos and \( sP \) have \( ... = \text{wp} \ t (\text{wp} \ u (\lambda s. c * P \ s)) \ s \)

by (simp add: scalingD[OF healthy-scalingD])

finally show \( c * \text{wp} \ t (\text{wp} \ u \ P) \ s = \text{wp} \ t (\text{wp} \ u (\lambda s. c * P \ s)) \ s \).

qed

**Lemma** nearly-healthy-wlp-Seq:

fixes \( t : 's \text{ prog} \) and \( u \)
assumes \( ht: \text{nearly-healthy (wlp t)} \) and \( hu: \text{nearly-healthy (wlp u)} \)
shows nearly-healthy (wlp (t ;; u))

proof(rule nearly-healthy1, simp-all add:wp-eval)

fix \( b \) and \( P::'s \Rightarrow \text{real} \)
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 \( P Q::'s \Rightarrow \text{real} \)
and \( s::'s \Rightarrow \text{real} \)
assume nQ: \( \text{nneg Q and bQ: bounded-by b Q} \)
Non-negative:
from nQ and bQ and hf have \( 0 \leq \text{wp f} \ Q \ s \) 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 simp:sign-simps)
with nQ and bQ and hg have \( 0 \leq \ldots * \text{wp g} \ Q \ s \)
by(auto intro:mult-nonneg-nonneg)
}
ultimately show \( 0 \leq P s * \text{wp f} \ Q \ s + (1 - P s) * \text{wp g} \ Q \ s \)
by(auto intro:mult-nonneg-nonneg)

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

next

Monotonic:

\( \text{fix } Q \ R ::''s \Rightarrow \text{real and } s \)
\( \text{assume } sQ : \text{sound } Q \text{ and } sR : \text{sound } R \text{ and } \text{le: } Q \vdash R \)
with \( uP \) have \( P \ s \ast \wp f \ Q \ s \leq P \ s \ast \wp f \ R \ s \)
by\( (\text{auto intro:mult-left-mono}) \)
moreover \{ from \ sQ \ sR \text{ le hg}
have \( \wp g Q \ s \leq \wp g R \ s \)
by\( (\text{blast dest: mono-transD}) \)
moreover from \( uP \) have \( 0 \leq 1 - P \ s \)
by\( (\text{auto simp: sign-simps}) \)
ultimately have \( (1 - P \ s) \ast \wp g Q \ s \leq (1 - P \ s) \ast \wp g R \ s \)
by\( (\text{auto intro:mult-left-mono}) \)
\} ultimately show \( P \ s \ast \wp f \ Q \ s + (1 - P \ s) \ast \wp g \ Q \ s \leq P \ s \ast \wp f \ R \ s + (1 - P \ s) \ast \wp g \ R \ s \)
by\( (\text{auto}) \)
next

Scaling:

\( \text{fix } Q ::''s \Rightarrow \text{real and } c ::\text{real and } s::''s \)
\( \text{assume } uQ : \text{unitary } Q \)
from \( uQ \ text{ hf hg}
have \( utQ : \text{unitary } (\wp f Q) \text{ unitary } (\wp g Q) \)
by\( (\text{auto}) \)
moreover from \( uP \) have \( 0 \leq P \ s \leq 1 - P \ s \)
by\( (\text{auto simp: sign-simps}) \)
ultimately show \( 0 \leq \wp f Q \ s \leq \wp g Q \ s \)
by\( (\text{auto}) \)

qed
4.2. HEALTHINESS

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)
  by(auto)
qed

lemma nearly-healthy-wlp-DC:
fixes f::'s prog
assumes hf: nearly-healthy (wp f)
  and hg: nearly-healthy (wp g)
shows nearly-healthy (wp (f ⊞ g))
proof(intro nearly-healthyI bounded-byI le-funI unitaryI2,
  simp-all add:wp-eval, safe)
fix P::'s ⇒ real and s::'s
assume uP: unitary P
with hf hg have atP: unitary (wp f P) unitary (wp g P) by(auto)
thus 0 ≤ wp f P s 0 ≤ wp g P s by(auto)
have min (wp f P s) (wp g P s) ≤ wp f P s by(auto)
also from atP have ... ≤ 1 by(auto)
finally show min (wp f P s) (wp g P s) ≤ 1
fix Q::'s ⇒ real
assume uQ: unitary Q and le: P ⊢ Q
have min (wp f P s) (wp g P s) ≤ wp f P s by(auto)
also from uP uQ le have ...
  ≤ wp f Q s
  by(auto intro:le-funD[OF nearly-healthy-monoD, OF hf])
finally show min (wp f P s) (wp g P s) ≤ wp f Q s
have min (wp f P s) (wp g P s) ≤ wp g P s by(auto)
also from uP uQ le have ...
  ≤ wp g Q s
  by(auto intro:le-funD[OF nearly-healthy-monoD, OF hg])
finally show min (wp f P s) (wp g P s) ≤ wp 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)
4.2. HEALTHINESS

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

from \( nP \) \( bP \) \( \text{assms} \) have \( 0 \leq wp f P s \) by(auto)
thus \( 0 \leq \max (wp f P s) (wp g P s) \) by(auto)

next
fix \( P::'s \Rightarrow \) real and \( Q \) and \( s::'s \)
from \( \text{assms} \) have \( \text{mf}:\text{mono-trans} (wp f) \) and \( \text{mg}:\text{mono-trans} (wp g) \) by(auto)
assume \( sP: \text{sound} P \) and \( sQ: \text{sound} Q \) and \( le: P \vdash Q \)
hence \( wp f P s \leq wp f Q s \) and \( wp g P s \leq wp g Q s \)
by(auto intro:le-funD[OF mono-transD,OF mf] le-funD[OF mono-transD,OF mg])
thus \( \max (wp f P s) (wp g P s) \leq \max (wp f Q s) (wp g Q s) \) by(auto)

next
fix \( P::'s \Rightarrow \) real and \( c: \) real and \( s::'s \)
assume \( sP: \text{sound} P \) and \( \text{pos}: 0 \leq c \)
from \( \text{assms} \) have \( \text{sf}: \text{scaling} (wp f) \) and \( \text{sg}: \text{scaling} (wp g) \) by(auto)
from \( \text{pos} \) 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 \( \text{pos} \)
have \( \ldots = \max (wp f (\lambda s. c * P s) s) (wp g (\lambda s. c * P s) s) \)
by(simp add:scalingD[OF sf] scalingD[OF sg])
finally show \( c * \max (wp f P s) (wp g P s) = \max (wp f (\lambda s. c * P s) s) (wp g (\lambda s. c * P s) s) \).
qed

lemma nearly-healthy-wlp-AC:
fixes \( f::'s \) \( \text{prog} \)
assumes \( \text{hf}:\text{nearly-healthy} (wp f) \)
and \( \text{hg}:\text{nearly-healthy} (wp g) \)
shows \( \text{nearly-healthy} (wp (f \boxplus g)) \)
proof(intro nearly-healthyI bounded-byI nnegI unitaryI2 le-funI, simp-all only:wp-eval)
fix \( b \) and \( P::'s \Rightarrow \) real and \( s::'s \)
assume \( uP: \text{unitary} P \)
with \( \text{hf} \) have \( \text{wp} f P s \leq 1 \) by(auto)
moreover from \( uP \) \( \text{hg} \) have unitary \( (wp g P) \) by(auto)
hence \( \text{wp} g P s \leq 1 \) by(auto)
ultimately show \( \max (wp f P s) (wp g P s) \leq 1 \) by(auto)
from \( uP \) \( \text{hf} \) have unitary \( (wp f P) \) by(auto)
hence \( 0 \leq wp f P s \) by(auto)
thus \( 0 \leq \max (wp f P s) (wp g P s) \) by(auto)
next
fix $P::'s \Rightarrow \text{real and } Q$ and $s::'s$
assume $uP$: unitary $P$ and $uQ$: unitary $Q$ and $\text{le: } P \vdash Q$
hence $\text{wlp f P s} \leq \text{wlp f Q s and wlp g P s} \leq \text{wlp g Q s}$
by(auto intro:le-funD[OF nearly-healthy-monoD, OF hf]
le-funD[OF nearly-healthy-monoD, OF hg])
thus $\max (\text{wlp f P s}) (\text{wlp g P s}) \leq \max (\text{wlp f Q s}) (\text{wlp g Q s})$ by(auto)
qed

lemma healthy-wp-Embed:
healthy $t \Rightarrow \text{healthy (wp (Embed t))}$
unfolding wp-def Embed-def by(simp)

lemma nearly-healthy-wlp-Embed:
nearly-healthy $t \Rightarrow \text{nearly-healthy (wlp (Embed t))}$
unfolding wp-def Embed-def by(simp)

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

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

lemma healthy-wp-SetDC:
fixes prog::'b \Rightarrow 'a prog and $S::'a \Rightarrow 'b set$
assumes healthy: $\forall x s. x \in S s \Rightarrow \text{healthy (wp (prog x))}$
and nonempty: $\forall s. \exists x. x \in S s$
shows healthy ($\text{wp (SetDC prog S)}$) (is healthy $?T$)
proof(intro healthy-parts bounded-byI nnegI le-funI, simp-all only:wp-eval)
fix $b$ and $P::'a \Rightarrow \text{real and } s::'a$
assume $bP$: bounded-by $b P$ and $nP$: nneg $P$
hence $sP$: sound $P$ by(auto)
from nonempty obtain $x$ where $\text{xin: } x \in (\\lambda a. \text{wp (prog a)} P s) ' S s$ by(blast)
moreover from $sP$ and healthy have $\forall x(\lambda a. \text{wp (prog a)} P s) ' S s. 0 \leq x$ by(auto)
ultimately have \( \inf ((\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s) \leq x \)

by(intro cInf-lower bdd-belowI, auto)

also from \( \text{xin} \) and healthy and \( sP \) and \( bP \) have \( x \leq b \) by(blast)

finally show \( \inf ((\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s) \leq b \)

from \( \text{xin} \) and \( sP \) and healthy

show \( 0 \leq \inf ((\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s) \) by(blast intro cInf-greatest)

next

fix \( P::'a \Rightarrow \text{real} \) and \( Q \) and \( s::'a \)

assume \( sP: \text{sound} \ P \) and \( sQ: \text{sound} \ Q \) and \( \text{le}: P \vdash Q \)

from nonempty obtain \( x \) where \( \text{xin}: x \in (\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s \) by(blast)

moreover from \( sP \) and healthy

have \( \forall x \in (\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s. \ 0 \leq x \) by(auto)

moreover

have \( \forall x \in (\lambda a. \wp (\text{prog} \ a) \ Q \ s) \cdot S \ s. \ \exists y \in (\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s. \ y \leq x \)

proof(rule ballI, clarify, rule bexI)

fix \( x \) and \( a \) assume \( \text{ain}: a \in S \ s \)

with healthy and \( sP \) and \( sQ \) and \( \text{le} \)
show \( \wp (\text{prog} \ a) \ P \ s \leq \wp (\text{prog} \ a) \ Q \ s \)

by(auto dest:mono-transD[OF healthy-monoD])

from \( \text{ain} \)
show \( \wp (\text{prog} \ a) \ P \ s \in (\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s \) by(simp)

qed

ultimately

show \( \inf ((\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s) \leq \inf ((\lambda a. \wp (\text{prog} \ a) \ Q \ s) \cdot S \ s) \)

by(intro cInf-mono, blast+)

next

fix \( P::'a \Rightarrow \text{real} \) and \( c::\text{real} \) and \( s::'a \)

assume \( sP: \text{sound} \ P \) and \( \text{pos}: 0 \leq c \)

from nonempty obtain \( x \) where \( \text{xin}: x \in (\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s \) by(blast)

have \( c \star \inf ((\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s) = \)

\( \inf ((\text{op} \star c \cdot ((\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s)) \) (is \( ?U = ?V \))

proof(rule antisym)

show \( ?U \leq ?V \)

proof(rule cInf-greatest)

from nonempty show \( \text{op} \star c \cdot ((\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s) \neq {} \) by(auto)

fix \( x \)
assume \( x \in \text{op} \star c \cdot ((\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s) \)

then obtain \( y \) where \( \text{yin}: y \in (\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s \) and \( \text{rux}: x = c \)

\* \( y \) by(auto)

have \( \inf ((\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s) \leq y \)

proof(intro cInf-lower[OF yin] bdd-belowI)

fix \( z \)
assume \( \text{zin}: z \in (\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s \)

then obtain \( a \) where \( a \in S \ s \) and \( z = \wp (\text{prog} \ a) \ P \ s \) by(auto)

with \( sP \)
show \( 0 \leq z \) by(auto dest:healthy)

qed

with \( \text{pos} \)
show \( c \star \inf ((\lambda a. \wp (\text{prog} \ a) \ P \ s) \cdot S \ s) \leq x \) by(auto intro:mult-left-mono)

qed

show \( ?V \leq ?U \)

proof(cases)
CHAPTER 4. THE PGCL LANGUAGE

assume \( cz: c = 0 \)
moreover { from nonempty obtain \( c \) where \( c \in S s \) by(auto)
  hence \( \exists x. \exists xa \in S s. x = \text{wp} (\text{prog} xa) \) \( P s \) by(auto) }
ultimately show \(?\text{thesis}\) by(simp add:image-def)

next
assume \( cnz: c \neq 0 \)
have inverse \( c * ?V \leq \text{inverse} c * ?U \)
proof(simp add:mult.assoc[symmetric] cnz del:Inf-image-eq, rule cInf-greatest)
  from nonempty show \((\lambda a. \text{wp} (\text{prog} a) P s) \cdot S s \neq \{\}\) by(auto)
  fix \( x \) assume \( x \in (\lambda a. \text{wp} (\text{prog} a) P s) \cdot S s \)
then obtain \( a \) where \( a \in S s \) and \( \text{rwx}: x = \text{wp} (\text{prog} a) P s \) by(auto)
  have Inf \((\text{op} * c \cdot (\lambda a. \text{wp} (\text{prog} a) P s) \cdot S s) \leq c \cdot x \)
proof(intro cInf-lower bdd-belowI)
  from ain show \((c \cdot x) \in (\text{op} * c \cdot (\lambda a. \text{wp} (\text{prog} a) P s) \cdot S s) \)
  by(auto simp:rwx)
  fix \( z \) assume \( z \in (\text{op} * c \cdot (\lambda a. \text{wp} (\text{prog} a) P s) \cdot S s) \)
then obtain \( b \) where \( b \in S s \) and \( \text{rwx}: z = c \cdot wp (\text{prog} b) P s \) by(auto)
  with \( sP \) have \( 0 \leq wp (\text{prog} b) P s \) by(auto dest:healthy)
  with \( \text{pos} \) show \( 0 \leq z \) by(auto simp:rwx intro:mult-nonneg-nonneg)
qed
moreover from \( \text{pos} \) have \( 0 \leq \text{inverse} c \) by(simp)
ultimately have inverse \( c * \text{Inf} (\text{op} * c \cdot (\lambda a. \text{wp} (\text{prog} a) P s) \cdot S s) \leq \text{inverse} c * (c * x) \)
  by(auto intro:mult-left-mono)
also from \( \text{cnz} \) have \( \ldots = x \) by(simp)
finally show inverse \( c * \text{Inf} (\text{op} * c \cdot (\lambda a. \text{wp} (\text{prog} a) P s) \cdot S s) \leq x \).
qed
with \( \text{pos} \) have \( c * (\text{inverse} c * ?V) \leq c * (\text{inverse} c * ?U) \)
  by(auto intro:mult-left-mono)
with \( \text{cnz} \) show \(?\text{thesis}\) by(simp add:mult.assoc[symmetric])
qed
qed
also have \( \ldots = \text{Inf} ((\lambda a. c \cdot \text{wp} (\text{prog} a) P s) \cdot S s) \)
  by(simp add:image-comp[symmetric] o-def)
also from \( sP \) and \( \text{pos} \) have \( \ldots = \text{Inf} ((\lambda a. \text{wp} (\text{prog} a) (\lambda s. c * P s) s) \cdot S s) \)
  by(simp add:scalingD[OF healthy-scalingD, OF healthy] cong:image-cong)
finally show \( c * \text{Inf} ((\lambda a. \text{wp} (\text{prog} a) P s) \cdot S s) = \text{Inf} ((\lambda a. \text{wp} (\text{prog} a) (\lambda s. c * P s) s) \cdot S s) \).
qed

lemma nearly-healthy-wlp-SetDC:
  fixes \( \text{prog: } b \Rightarrow 'a \text{ prog and } S::'a \Rightarrow 'b \text{ set} \)
  assumes healthy: \( \forall x. s. x \in S s \Rightarrow \text{nearly-healthy (wp (prog x))} \)
  and nonempty: \( \exists s. \exists x. x \in S s \)
  shows nearly-healthy (wp (SetDC prog S)) (is nearly-healthy ?T)
proof(intro nearly-healthyI unitaryI2 bounded-byI negI le-fund, simp-all:wp-eval)
4.2. HEALTHINESS

fix $b$ and $P::'a \Rightarrow \text{real and } s::'a$
assume $uP::\text{unitary } P$

from nonempty obtain $x$ where $xin: x \in (\lambda a. \text{wlp} (\text{prog } a) P) \cdot S \ s by (\text{blasts})$
moreover {
from $uP$ healthy
have $\forall x(\lambda a. \text{wlp} (\text{prog } a) P) : S \ s. \text{unitary } x$ by(auto)
  hence $\forall x(\lambda a. \text{wlp} (\text{prog } a) P) : S \ s. \ 0 \leq x \ s$ by(auto)
  hence $\forall y(\lambda a. \text{wlp} (\text{prog } a) P) : S \ s. \ 0 \leq y$ by(auto)
}
ultimately have $\text{Inf} ((\lambda a. \text{wlp} (\text{prog } a) P) s) \cdot S \ s) \leq x \ by (\text{intro } c\text{Inf-lower})$
also from $xin$ healthy $uP$ have $x \leq 1 \ by (\text{blasts})$
finally show $\text{Inf} ((\lambda a. \text{wlp} (\text{prog } a) P) s) \cdot S \ s) \leq 1$.

next
fix $P::'a \Rightarrow \text{real and } Q$ and $s::'a$
assume $uP::\text{unitary } P$ and $uQ::\text{unitary } Q$ and $le:: P \vdash Q$

from nonempty obtain $x$ where $xin: x \in (\lambda a. \text{wlp} (\text{prog } a) P) \cdot S \ s by (\text{blasts})$
moreover {
from $uP$ healthy
have $\forall x(\lambda a. \text{wlp} (\text{prog } a) P) : S \ s. \text{unitary } x$ by(auto)
  hence $\forall x(\lambda a. \text{wlp} (\text{prog } a) P) : S \ s. \ 0 \leq x \ s$ by(auto)
  hence $\forall y(\lambda a. \text{wlp} (\text{prog } a) P) : S \ s. \ 0 \leq y$ by(auto)
}
moreover
have $\forall x(\lambda a. \text{wlp} (\text{prog } a) Q) s) \cdot S \ s. \exists y(\lambda a. \text{wlp} (\text{prog } a) P) s) : S \ s. \ y \leq x$
proof (rule ballI, clarify, rule bexI)
fix $x$ and $a$ assume $ain:\ a \in S \ s$
from $uP uQ \ le \ show \ wlp (\text{prog } a) P \ s \leq \text{wlp} (\text{prog } a) Q \ s$
by (auto intro:le-funD[OF nearly-healthy-unitaryD[OF - uP]]
intro:cInf-greatest)
from $ain$ show $\text{wlp} (\text{prog } a) P \ s \in (\lambda a. \text{wlp} (\text{prog } a) P) \cdot S \ s by (\text{simp})$
qed
ultimately
show $\text{Inf} ((\lambda a. \text{wlp} (\text{prog } a) P) s) \cdot S \ s) \leq \text{Inf} ((\lambda a. \text{wlp} (\text{prog } a) Q) s) \cdot S \ s)$
by (intro cInf-monot, blast+)
qed

lemma healthy-wp-SetPC:
fixes $p::s \Rightarrow 'a \Rightarrow \text{real}$
and $f::'a \Rightarrow 's \text{ prog}$
assumes healthy: $\forall a. a \in \text{supp} (p \ s) \Rightarrow \text{healthy} (wp (f \ a))$
  and sound: $\forall s. \text{sound} (p \ s)$
  and sub-dist: $\forall s. (\sum a \in \text{supp} (p \ s). \ p \ s \ a) \leq 1$
shows healthy \( \text{wp} (\text{SetPC} \; f \; p) \) (\text{is healthy} \; ?X)\\nproof\{(\text{intro healthy-parts bounded-by} \; \text{nnegI le-funI}, \text{simp-all add:wp-eval)}\\nfix \; b \; \text{and} \; P::'s \Rightarrow \text{real and} \; s::'s\\nassume \; bP: \text{bounded-by} \; b \; P \; \text{and} \; nP: \text{nneg} \; P\\nhence \; sP: \text{sound} \; P \; \text{by}(\text{auto})\\n\\n\text{from} \; sP \; \text{and} \; bP \; \text{and} \; \text{healthy} \; \text{have} \; \lambda a. \; a \in \text{supp} \; (p \; s) \Rightarrow \text{wp} \; (f \; a) \; P \; s \leq b\\n\text{by}(\text{blast dest:healthy-bounded-byD})\\n\\nwith \; \text{sound have} \; (\Sigma \; a \in \text{supp} \; (p \; s). \; p \; s \; a \; * \; \text{wp} \; (f \; a) \; P \; s) \leq (\Sigma \; a \in \text{supp} \; (p \; s). \; p \; s \; a \; * \; b)\\n\text{by}(\text{blast intro:setsum-mono mult-left-mono})\\n\\nalso \; \text{have} \; \ldots = (\Sigma \; a \in \text{supp} \; (p \; s). \; p \; s \; a \) * \; b\\n\text{by}(\text{simp add:setsum-left-distrib})\\n\\nalso \{\\n\text{from} \; bP \; \text{and} \; nP \; \text{have} \; 0 \leq b \; \text{by}(\text{blast})\\n\\nwith \; \text{sub-dist have} \; (\Sigma \; a \in \text{supp} \; (p \; s). \; p \; s \; a \) * \; b \leq 1 \; * \; b\\n\text{by}(\text{rule mult-right-mono})\\n\}\\n\\nalso \; \text{have} \; 1 \; * \; b = b \; \text{by}(\text{simp})\\n\\nfinally \; \text{show} \; (\Sigma \; a \in \text{supp} \; (p \; s). \; p \; s \; a \; * \; \text{wp} \; (f \; a) \; P \; s) \leq b .\\n\\nshow \; 0 \leq (\Sigma \; a \in \text{supp} \; (p \; s). \; p \; s \; a \; * \; \text{wp} \; (f \; a) \; P \; s)\\nproof(\text{rule setsum-nonneg, clarify, rule mult-nonneg-nonneg})\\n\\nfix \; x\\n\\n\text{from} \; \text{sound} \; \text{show} \; 0 \leq p \; s \; x \; \text{by}(\text{blast})\\nassume \; x \in \text{supp} \; (p \; s) \; \text{with} \; sP \; \text{and} \; \text{healthy}\\n\\n\text{show} \; 0 \leq \text{wp} \; (f \; x) \; P \; s \; \text{by}(\text{blast})\\n\\nqed\\n\\nnext\\n\\nfix \; P::'s \Rightarrow \text{real and} \; Q::'s \Rightarrow \text{real and} \; s\\nassume \; sP: \text{sound} \; P \; \text{and} \; sQ: \text{sound} \; Q \; \text{and} \; \text{ent}: P \vdash Q\\n\\nwith \; \text{healthy have} \; \lambda a. \; a \in \text{supp} \; (p \; s) \Rightarrow \text{wp} \; (f \; a) \; P \; s \leq \text{wp} \; (f \; a) \; Q \; s\\n\text{by}(\text{blast})\\n\\nwith \; \text{sound have} \; (\Sigma \; a \in \text{supp} \; (p \; s). \; p \; s \; a \; * \; \text{wp} \; (f \; a) \; P \; s) \leq (\Sigma \; a \in \text{supp} \; (p \; s). \; p \; s \; a \; * \; \text{wp} \; (f \; a) \; Q \; s)\\n\text{by}(\text{blast intro:setsum-mono mult-left-mono})\\n\\nnext\\n\\nfix \; P::'s \Rightarrow \text{real and} \; c::\text{real and} \; s::'s\\nassume \; \text{sound: sound} \; P \; \text{and} \; \text{pos}: 0 \leq c\\n\\n\text{have} \; c \; * \; (\Sigma \; a \in \text{supp} \; (p \; s). \; p \; s \; a \; * \; \text{wp} \; (f \; a) \; P \; s) = (\Sigma \; a \in \text{supp} \; (p \; s). \; p \; s \; a \; * \; (c \; * \; \text{wp} \; (f \; a) \; P \; s))\\n\text{(is \; ?A = \; ?B)}\\n\text{by}(\text{simp add:setsum-right-distrib ac-simps})\\n\\nalso \; \text{from} \; \text{sound and} \; \text{pos and} \; \text{healthy}\\n\\n\text{have} \; \ldots = (\Sigma \; a \in \text{supp} \; (p \; s). \; p \; s \; a \; * \; \text{wp} \; (f \; a) \; (\lambda s. \; c \; * \; P \; s) \; s)\\n\text{by}(\text{auto simp:scalingD[OF healthy-scalingD]})\\n\\nfinally \; \text{show} \; ?A = \ldots .\\n\\nqed
4.2. HEALTHINESS

**Lemma nearly-healthy-wlp-SetPC:**

**Fixes** p::s ⇒ 'a ⇒ real

**And** f::'a ⇒ 's prog

**Assumes** healthy: ∃ a. a ∈ supp (p s) ⇒ nearly-healthy (wp (f a))

**And** sound: ∃ a. a ∈ supp (p s)

**And** sub-dist: ∃ a. (∑ a∈supp (p s). p s a) ≤ 1

**Shows** nearly-healthy (wp (SetPC f p)) (is nearly-healthy ?X)

**Proof**

**Fix** b and P::s ⇒ real and s::s

**Assume** uP: unitary P

from uP healthy have ∃ a. a ∈ supp (p s) ⇒ unitary (wp (f a) P) by(auto)

**Hence** ∃ a. a ∈ supp (p s) ⇒ wp (f a) P s ≤ 1 by(auto)

with sound have (∑ a∈supp (p s). p s a * wp (f a) P s) ≤ (∑ a∈supp (p s). p s a * 1)

by(blast intro:setsum-mono mult-left-mono)

also have ... = (∑ a∈supp (p s). p s a)

by(simp add:setsum-left-distrib)

also note sub-dist

finally show (∑ a∈supp (p s). p s a * wp (f a) P s) ≤ 1 .

show 0 ≤ (∑ a∈supp (p s). p s a * wp (f a) P s)

proof(rule setsum-nonneg, clarify, rule mult-nonneg-nonneg)

fix x

from sound show 0 ≤ p s x by(blast)

assume x ∈ supp (p s) with uP healthy

show 0 ≤ wp (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) ⇒ wp (f a) P s ≤ wp (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 * wp (f a) P s) ≤ (∑ a∈supp (p s). p s a * wp (f a) Q s)

by(blast intro:setsum-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 (wp (Apply f))

by(intro nearly-healthyI unitaryI2 nnegI bounded-byI, auto simp:o-def wp-eval)

**Lemma healthy-wp-Bind:**

**Fixes** f::'s ⇒ 'a

**Assumes** hsub: ∃ s. healthy (wp (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 ≤ b by (auto)
from bP nP hsub have nneg (wp (p f s)) P by (auto)
thus 0 ≤ wp (p f s) P s by (auto)
next
fix P Q ::'s expect and s ::'s
assume sound P sound Q P ⊢ Q
thus wp (p f s) P s ≤ 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 ≤ c
thus c * wp (p f s) P s = wp (p f s) (λs. c * P s) s
by (simp add: scalingD [OF healthy-scalingD, OF hsub])
qed

lemma nearly-healthy-wlp-Bind:
fixes f ::'s trans
assumes hsub: ∃s. nearly-healthy (wp (p f s))
shows nearly-healthy (wp (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 (wp (p f s)) P by (auto)
thus 0 ≤ wp (p f s) P s wp (p f s) P s ≤ 1 by (auto)
next
fix Q ::'s expect
assume unitary Q P ⊢ Q
with uP show wp (p f s) P s ≤ wp (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: P. sound P ⇒ sound (t P)
and hu: P. sound P ⇒ 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 \( \text{wp body} \ (t \ P) \ s \leq \text{wp body} \ (u \ P) \ s \)
by(auto intro:le-funD[OF mono-transD, OF healthy-monoD, OF hb])
thus \( «G» \ s * \text{wp body} \ (t \ P) \ s \leq «G» \ s * \text{wp body} \ (u \ P) \ s \)
by(auto intro:mult-left-mono)
qed

lemma wlp-loop-step-mono:
fixes \( t, u :: 's \) trans
assumes mb: nearly-healthy (wp body)
and ht: \( \forall P. \text{unitary} \ (t \ P) \implies \text{unitary} \ (u \ P) \)
and hu: \( \forall P. \text{unitary} \ (t \ P) \implies \text{unitary} \ (u \ P) \)
shows le-utrans \( \text{wp body} \ (t \ P) \ s \leq \text{wp body} \ (u \ P) \ s \)
proof (intro le-utransI le-funI, simp add: wp-eval)
fix \( P :: 's \) expect and \( s :: 's \)
assume uP: unitary \( P \)
with \( \text{le} \) have \( t \ P \vdash u \ P \) by(auto)
moreover from \( uP \) \( \text{ht} \) \( \text{hu} \) have unitary \( (t \ P) \) unitary \( (u \ P) \) by(auto)
ultimately have \( \text{wp body} \ (t \ P) \ s \leq \text{wp body} \ (u \ P) \ s \)
by(rule le-funD[OF nearly-healthy-monoD[OF mb]])
thus \( «G» \ s * \text{wp body} \ (t \ P) \ s \leq «G» \ s * \text{wp body} \ (u \ P) \ s \)
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 \( \lambda s. «G» \ s * \text{wp body} \ (\lambda s. \text{bound-of} \ P) \ s + \langle \text{N} G \rangle \ s * P \ s \vdash \lambda s. \text{bound-of} \ P \)
proof (rule le-funI)
fix \( s \)
from sP have sound \( (\lambda s. \text{bound-of} \ P) \) by(auto)
moreover hence bounded-by \( (\text{bound-of} \ P) \) \( \lambda s. \text{bound-of} \ P \) by(auto)
ultimately have bounded-by \( (\text{bound-of} \ P) \) \( \text{wp body} \ (\lambda s. \text{bound-of} \ P) \)
using hb by(auto)
hence \( \text{wp body} \ (\lambda s. \text{bound-of} \ P) \ s \leq \text{bound-of} \ P \) by(auto)
moreover from sP have \( P \ s \leq \text{bound-of} \ P \) by(auto)
ultimately have \( «G» \ s * \text{wp body} \ (\lambda a. \text{bound-of} \ P) \ s + (1 - «G» \ s) * P \ s \leq 
«G» \ s * \text{bound-of} \ P + (1 - «G» \ s) * \text{bound-of} \ P \)
by(blast intro:add-mono mult-left-mono)
thus \( «G» \ s * \text{wp body} \ (\lambda a. \text{bound-of} \ P) \ s + \langle \text{N} G \rangle \ s * P \ s \leq \text{bound-of} \ P \)
by(simp add:algebra-simps negate-embed)
qed

lemma lfp-loop-greatest:
fixes \( P :: 's \) expect
assumes \( \forall R. \lambda s. \langle G \rangle s * \text{wp body} R s + \langle N \ G \rangle s * P s \vdash R \implies \text{sound} R \)
\[ \implies Q \vdash R \]
and \( hh: \text{healthy} (\text{wp body}) \)
and \( sP: \text{sound} P \)
and \( sQ: \text{sound} Q \)
shows \( Q \vdash \text{lfp-exp} (\lambda Q s. \langle G \rangle s * \text{wp body} Q s + \langle N \ G \rangle s * P s) \)
using \( sP \) by\((\text{auto intro!} \lambda sQ. \text{OF lb sQ})\) \( sP \) \( \text{lfp-loop-fp} hh \)

**lemma** \( \text{lfp-loop-sound} \):
fixes \( P: s \text{'s expect} \)
assumes \( hh: \text{healthy} (\text{wp body}) \)
and \( sP: \text{sound} P \)
shows \( \text{sound} (\text{lfp-exp} (\lambda Q s. \langle G \rangle s * \text{wp body} Q s + \langle N \ G \rangle s * P s)) \)
using \( \text{assms by(\text{auto intro!} \lambda sP. \text{lfp-exp-sound} lfp-loop-fp)} \)

**lemma** \( \text{wlp-loop-step-unitary} \):
fixes \( t u: s \text{'s trans} \)
assumes \( hh: \text{nearly-healthy} (\text{wlp body}) \)
and \( ht: \forall P. \text{unitary} P \implies \text{unitary} (t P) \)
and \( uP: \text{unitary} P \)
shows \( \text{unitary} (\text{wlp body} (; \text{Embed t} \langle G \rangle \oplus \text{Skip}) P) \)

**proof** \((\text{intro unitaryI2 megl bounded-by1, simp-all add:wp-eval})\)
fix \( s: s \text{'s} \)
from \( ht uP \) have \( utP: \text{unitary} (t P) \) by\((\text{auto})\)
with \( hh \) have \( \text{unitary} (\text{wp body} (t P)) \) by\((\text{auto})\)
hence \( 0 \leq \text{wlp body} (t P) s \) by\((\text{auto})\)
with \( uP \) show \( 0 \leq \langle G \rangle s * \text{wlp body} (t P) s + (1 - \langle G \rangle s) * P s \)
by\((\text{auto intro!} \text{add-nonneg-nonneg mult-nonneg-nonneg})\)
from \( ht uP \) have \( \text{bounded-by1} (t P) \) by\((\text{auto})\)
with \( utP hh \) have \( \text{bounded-by1} (\text{wp body} (t P)) \) by\((\text{auto})\)
hence \( \text{wlp body} (t P) s \leq 1 \) by\((\text{auto})\)
with \( uP \) have \( \langle G \rangle s * \text{wlp body} (t P) s + (1 - \langle G \rangle s) * P s \leq \langle G \rangle s * 1 + (1 - \langle G \rangle s) * 1 \)
by\((\text{blast intro! add-mono mult-left-mono})\)
also have \( ... = 1 \) by\((\text{simp})\)
finally show \( \langle G \rangle s * \text{wlp body} (t P) s + (1 - \langle G \rangle s) * P s \leq 1 \).
qed

**lemma** \( \text{wp-loop-step-sound} \):
fixes \( t u: s \text{'s trans} \)
assumes \( hh: \text{healthy} (\text{wp body}) \)
and \( ht: \forall P. \text{sound} P \implies \text{sound} (t P) \)
and \( sP: \text{sound} P \)
shows \( \text{sound} (\text{wp body} (; \text{Embed t} \langle G \rangle \oplus \text{Skip}) P) \)

**proof** \((\text{intro soundI2 megl bounded-by1, simp-all add:wp-eval})\)
fix \( s: s \text{'s} \)
from \( ht sP \) have \( stP: \text{sound} (t P) \) by\((\text{auto})\)
with \( hh \) have \( 0 \leq \text{wp body} (t P) s \) by\((\text{auto})\)
with \( sP \) show \( 0 \leq \langle G \rangle s * \text{wp body} (t P) s + (1 - \langle G \rangle s) * P s \)
by (auto intro: add-nonneg-nonneg mult-nonneg-nonneg)

from hsP 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 ≤ bound-of (t P) using hs by (auto)
hence wp body (t P) s ≤ max (bound-of P) (bound-of (t P)) by (auto)
moreover {
  from sP have P s ≤ bound-of P by (auto)
hence P s ≤ max (bound-of P) (bound-of (t P)) by (auto)
}
ultimately have «G» s * wp body (t P) s + (1 − «G» s) * P s ≤
  «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 ≤
  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 → body od) P =
    gfp-exp (λQ s. «G» s * wlp body Q s + «N G» s * P s)
  (is ?X = gfp-exp (?Y P))
proof —
  let ?Z u = (body ;; Embed u « G » ⊕ Skip)
show ?thesis
proof (unfold wp-eval, intro gfp-pulldown assms le-funI)
  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 at: ∃Q. unitary Q ⇒ 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 (λa. « G » a * wlp body Q a + « 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 \( 0 \leq \langle G \rangle s \cdot \text{wlp body } Q s \cdot + \langle \mathcal{N} \rangle G \rangle s \cdot P s \) 
by (auto intro: add-nonneg-nonneg mult-nonneg-nonneg)

from healthy \( uQ \) have bounded-by 1 (wlp body \( Q \)) by (auto)
with \( uP \) have \( \langle G \rangle s \cdot \text{wlp body } Q s \cdot + (1 - \langle G \rangle s) \cdot P s \leq \langle G \rangle s \cdot 1 + (1 - \langle G \rangle s) \cdot 1 \)
by (blast intro: add-mono mult-left-mono)
also have \( \ldots = 1 \) by (simp)
finally show \( \langle G \rangle s \cdot \text{wlp body } Q s \cdot + (1 - \langle G \rangle s) \cdot P s \leq 1 \)
by (simp add: negate-embed)
qed

next
fix \( P \ Q \ R :: \text{'s expect and s::'s} \)
assume \( uP: \text{unitary } P \) and \( uQ: \text{unitary } Q \) and \( uR: \text{unitary } R \)
and le: \( Q \vdash R \)
hence \( \text{wlp body } Q s \leq \text{wlp body } R s \)
by (blast intro: le-funD [OF nearly-healthy-monoD, OF healthy])
thus \( \langle G \rangle s \cdot \text{wlp body } Q s \cdot + \langle \mathcal{N} \rangle G \rangle s \cdot P s \leq \langle G \rangle s \cdot \text{wlp body } R s \cdot + \langle \mathcal{N} \rangle G \rangle s \cdot P s \)
by (auto intro: mult-left-mono)
qed

next
fix \( t \ u :: \text{'s trans} \)
assume le-utrans \( t \ u \)
\( \land P. \text{unitary } P \Longrightarrow \text{unitary } (t P) \)
\( \land P. \text{unitary } P \Longrightarrow \text{unitary } (u P) \)
thus le-utrans (wlp (\$Z t)) (wlp (\$Z u))
by (blast intro!: wp-loop-step-mono [OF healthy])
qed

Likewise, we can rewrite strict loops.

lemma \( \text{wp-loop-sound} \):
fixes body :: 's prog
assumes \( sP: \text{sound } P \)
and \( hb: \text{healthy } (\text{wp body}) \)
shows \( \text{sound } (\text{wp do } G \longrightarrow \text{body } od P) \)
unfolding wp-eval
proof (intro lfp-trans-sound sP)
let \( \nu = \lambda P s. \text{bound-of } P \)
show le-trans (wp (body :: Embed \( \nu G \oplus \text{Skip} \)) \( \nu \))
by (intro le-transI, simp add: wp-eval lfp-loop-fp [unfolded negate-embed] hb)
show \( \land P. \text{sound } P \Longrightarrow \text{sound } (\nu P) \) by (auto)
qed
4.2. HEALTHINESS

\[(is \ \ ?X = lfp-exp (\ ?Y P))\]

**proof**

- **let** \( ?Z u = (\text{body } ;; \text{Embed } u \ _\ G \ _\oplus \ \text{Skip}) \)

- **show** \( ?\text{thesis} \)

**proof**

(\ unfolding wp-eval, intro \( \text{lfp-pulldown} \ \text{assms le-funI} \ \text{sP} \ \text{mono-transI} \)

- **fix** \( u \ \text{P} \)

- **show** \( \text{wp} (\ ?Z u) \ \text{P} = \ ?Y P (u \ \text{P}) \ \text{by}(\ \text{simp add:wp-eval negate-embed}) \)

**next**

- **fix** \( t::'\text{s trans and} \ \text{P::'}\text{s expect} \)

- **assume** \( ut: \ \land Q. \ \text{sound} \ Q \implies \ \text{sound} (t \ Q) \ \text{and} \ uP: \ \text{sound} \ P \)

- **with** \( \text{healthy show} \ \text{sound} (wp (\ ?Z t) \ P) \ \text{by}(\ \text{rule wp-loop-step-sound}) \)

**next**

- **fix** \( P \ \text{Q::'}\text{s expect} \)

- **assume** \( sP: \ \text{sound} \ P \ \text{and} \ sQ: \ \text{sound} \ Q \)

- **show** \( \text{sound} (\lambda a. \ \langle G \rangle a \ast wp \ \text{body} \ Q \ a + \langle N G \rangle a \ast P a) \)

**proof**

(\ intro soundI2 nnegI bounded-byI)

- **fix** \( s::'\text{s} \)

- **from** \( sQ \ \text{have} \ \text{nneg} \ Q \ \text{bounded-by} \ (\text{bound-of} \ Q) \ \text{by}(\text{auto}) \)

- **with** \( \text{healthy have} \ \text{bounded-by} \ (\text{bound-of} \ Q) \ (wp \ \text{body} \ Q) \ \text{by}(\text{auto}) \)

- **hence** \( wp \ \text{body} \ Q \ s \leq \ \text{bound-of} \ Q \ \text{by}(\text{auto}) \)

- **hence** \( wp \ \text{body} \ Q \ s \leq \ \text{max} \ (\text{bound-of} \ P) \ (\text{bound-of} \ Q) \ \text{by}(\text{auto}) \)

- **moreover** \( \{

- **from** \( sP \ \text{have} \ P \ s \leq \ \text{bound-of} \ P \ \text{by}(\text{auto}) \)

- **hence** \( P \ s \leq \ \text{max} \ (\text{bound-of} \ P) \ (\text{bound-of} \ Q) \ \text{by}(\text{auto}) \)

\} \)

- **ultimately have** \( \langle G \rangle s \ast wp \ \text{body} \ Q \ s + \langle N G \rangle s \ast P s \leq \langle G \rangle s \ast \text{max} \ (\text{bound-of} \ P) + \langle N G \rangle s \ast \text{max} \ (\text{bound-of} \ P) \ (\text{bound-of} \ Q) \)

**by**(auto intro:adantl mono mult-left mono)

- **also have** \( \ldots = \ \text{max} \ (\text{bound-of} \ P) \ (\text{bound-of} \ Q) \ \text{by}(\text{simp add:algebra-simps negate-embed}) \)

- **finally show** \( \langle G \rangle s \ast wp \ \text{body} \ Q \ s + \langle N G \rangle s \ast P s \leq \ \text{max} \ (\text{bound-of} \ P) \ (\text{bound-of} \ Q) \).

- **from** \( sP \ \text{have} \ 0 \leq P s \ \text{by}(\text{auto}) \)

- **moreover from** \( sQ \ \text{healthy have} \ 0 \leq wp \ \text{body} \ Q \ s \ \text{by}(\text{auto}) \)

- **ultimately show** \( 0 \leq \langle G \rangle s \ast wp \ \text{body} \ Q \ s + \langle N G \rangle s \ast P s \)

**by**(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)

- **qed**

**next**

- **fix** \( P \ \text{Q::'}\text{s expect and} \ s::'s \)

- **assume** \( sQ: \ \text{sound} \ Q \ \text{and} \ sR: \ \text{sound} \ R \)

- **and** \( le: \ Q \vdash R \)

- **hence** \( wp \ \text{body} \ Q \ s \leq wp \ \text{body} \ R \ s \)

**by**(blast intro:le-funD[OF mono-transD, OF healthy-monoD, OF healthy])

- **thus** \( \langle G \rangle s \ast wp \ \text{body} \ Q \ s + \langle N G \rangle s \ast P s \leq \langle G \rangle s \ast wp \ \text{body} \ R \ s + \langle N G \rangle s \ast P s \)

**by**(auto intro:mult-left mono)

**next**
\[\text{fix} \ t \ u = \text{trans}\]
\[\text{assume} \ \text{le}: \text{le-trans} \ t \ u\]
\[\text{and} \ st: \wedge P, \ \text{sound} \ P \implies \text{sound} \ (t \ P)\]
\[\text{and} \ su: \wedge P, \ \text{sound} \ P \implies \text{sound} \ (u \ P)\]
\[\text{with} \ \text{healthy show} \ \text{le-trans} \ (wp (\var{Z} \ t)) \ (wp (\var{Z} \ u))\]
\[\text{by}(\text{rule wp-loop-step-mono})\]
\[\text{next}\]
\[\text{from} \ \text{healthy show} \ \text{le-trans} \ (wp (\var{Z} (\lambda P. \ \text{sound} \ P)) \ (\lambda P. \ \text{sound} \ (\var{Z} \ P)))\]
\[\text{by}(\text{intro le-transI}, \ \text{simp add}: wp\text{-eval lfp-loop-fp} [\text{unfolded negate-embed}])\]
\[\text{next}\]
\[\text{fix} \ P:: \text{expect and} \ s:: \text{s}\]
\[\text{assume} \ \text{sound} \ P\]
\[\text{thus} \ \text{sound} \ (\lambda s. \ \text{bound-of} \ P) \ \text{by}(\text{auto})\]
\[\text{qed}\]
\[\text{qed}\]

\text{lemma} \ nearly-healthy-wlp-loop:\n\[\text{fixes} \ \text{body:: \text{prog}}\]
\[\text{assumes} \ h:\text{nearby-healthy} \ (\text{wp body})\]
\[\text{shows} \ nearly-healthy \ (\text{wp} \ (\text{do} \ G \rightarrow \text{body \ ad}))\]
\[\text{proof}(\text{intro nearly-healthyI unitaryI2 nnegI2 bounded-byI2}, \ \text{simp-all add}: \text{wp-Loop1} h)\]
\[\text{fix} \ P:: \text{expect} \ \text{assume} \ uP: \text{unitary} \ P\]
\[\text{let} \ ?X R = \lambda Q s. \var{G} s * \text{wp body} Q s + \var{N} G s * R s\]
\[\text{show} \ \lambda s. \ 0 \vdash \text{gfp-exp} (\var{X} P)\]
\[\text{proof}(\text{rule gfp-exp-upperbound})\]
\[\text{show} \ \text{unitary} \ (\lambda s. \ 0:: \text{real}) \ \text{by}(\text{auto})\]
\[\text{with} \ h:\text{have} \ \text{unitary} \ (\text{wp body} (\lambda s. \ 0)) \ \text{by}(\text{auto})\]
\[\text{with} \ uP \ \text{show} \ \lambda s. \ 0 \vdash (\var{X} P (\lambda s. \ 0))\]
\[\text{by}(\text{blast intro!}: \text{le-funI add-nonneg-nonneg mult-nonneg-nonneg})\]
\[\text{qed}\]

\[\text{show} \ \text{gfp-exp} (\var{X} P) \vdash \lambda s. \ 1\]
\[\text{proof}(\text{rule gfp-exp-least})\]
\[\text{fix} \ Q:: \text{expect} \ \text{assume} \ uQ: \text{unitary} \ Q\]
\[\text{thus} \ Q \vdash \lambda s. \ 1 \ \text{by}(\text{auto})\]
\[\text{qed}\]

\[\text{fix} \ Q:: \text{expect} \ \text{assume} \ uQ: \text{unitary} \ Q \ \text{and} \ \text{le}: \ P \vdash \ Q\]
\[\text{show} \ \text{gfp-exp} (\var{X} P) \vdash \text{gfp-exp} (\var{X} Q)\]
\[\text{proof}(\text{rule gfp-exp-least})\]
\[\text{fix} \ R:: \text{expect assume} \ uR: \text{unitary} \ R\]
\[\text{assume} \ fp: \ R \vdash (\var{X} P R)\]
\[\text{also from} \ \text{le have} \ ... \vdash \var{X} Q R\]
4.2. HEALTHINESS

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 hh: 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::'
  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 hh 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 ⊨
proof
  fix P::'s expect and c::real
assume cnz: c ≠ 0 and mnc: c ≤ c and sP: sound P
with mnc have cpos: c < c by (auto)
  hence mnc: c ≤ 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 hh 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 (prove 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 mnc
  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 ⊢
  by (rule le-funI)
next
from sP have sound (λs. bound-of P) by (auto)
  with hh sP have sound (lfp-exp (?X P))
    by (blast intro: lfp-exp-sound lfp-loop-fp)
with mnc show sound (λs. c * lfp-exp (?X P) s)
  by (auto intro: sound-intros)
4.2. HEALTHINESS

from \(bb \ sP \ nnc\)
show \(\lambda s. \ «G\ s \ast \ wp \ body \ (\lambda s. \ bound-of \ (\lambda s. \ c \ast P \ s)) \ s \ast \ \langle N \ G\ s \ast (c \ast P \ s) \vdash \lambda s. \ bound-of \ (\lambda s. \ c \ast P \ s)\)
by((prover intro;lfp-loop-fp sound-intros))

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

assume \(nzc: c \neq 0\)
show \(?thesis \ (is \ ?X \ P \ c \ s = ?Y \ P \ c \ s)\)
proof((rule fun-cong|where \(x=s\), rule antisym))
from \(nzc \ nnc \ sP \ show \ ?X \ P \ c \vdash ?Y \ P \ c \ by\ (rule \ onesided)\)

moreover from \(nnc \ sP \ have \ \lambda s. \ c \ast \ ?X \ (\lambda s. \ c \ast P \ s) \ (inverse \ c) \ s \vdash \lambda s. \ c \ast \ ?Y \ (\lambda s. \ c \ast P \ s) \ (inverse \ c) \ s\)
by((blast intro:mult-left-mono))
with \(nnc \ have \ \lambda s. \ c \ast \ ?X \ (\lambda s. \ c \ast P \ s) \ (inverse \ c) \ s \vdash \lambda s. \ c \ast \ ?Y \ (\lambda s. \ c \ast P \ s) \ (inverse \ c) \ s\)
by((rule onesided))
with \(nnc \ have \ \lambda s. \ c \ast \ ?X \ (\lambda s. \ c \ast P \ s) \ (inverse \ c) \ s \vdash \lambda s. \ c \ast \ ?Y \ (\lambda s. \ c \ast P \ s) \ (inverse \ c) \ s\)
by((rule onesided))

ultimately have \(?Y \ P \ c \vdash ?X \ P \ c \ by\ (simp \ add:mult.assoc[symmetric])\)
qed

ultimately have \(?G \ s \ast \ wp \ body \ (\lambda s. \ b) \ s \ast \ «\langle N \ G\ s \ast P \ s \leq \ «G\ s \ast b\ + \ 0 \ s \ast b\)\)
by(auto intro!:add-mono mult-left-mono)
also have \(\vdash b \ by\ (simp \ add:negate-embed \ field-simps)\)
finally show \(?G \ s \ast \ wp \ body \ (\lambda s. \ b) \ s \ast \ «\langle N \ G\ s \ast P \ s \leq b \)\.
from \(bP \ nP \ have \ 0 \leq b \ by\ (auto)\)
thus sound \((\lambda s. \ b) \ by\ (auto)\)
qed

from \(bb \ bP \ nP \ show \ \lambda s. \ 0 \vdash \ lfp-exp \ (\lambda Q \ s. \ «G\ s \ast \ wp \ body \ Q \ s \ast \ «\langle N \ G\ s \ast P \ s\)\)
by(auto dest!:sound-nneg intro!:lfp-loop-greatest)
next
fix \( P \) ::'s expect
assume \( sP \): sound \( P \) and \( sQ \): sound \( Q \) and \( le: P \vdash Q \)
show \( \text{lfp-exp} \ (\forall X \ P) \vdash \text{lfp-exp} \ (\forall X \ Q) \)
proof (rule lfp-exp-greatest)
\begin{align*}
\text{fix } R::'s \text{ expect} \\
\text{assume } \lambda s. \ «G» s \ast \text{wp body } R s + \ «N \ G» s \ast P s \vdash R \\
\text{from } le \text{ have } \lambda s. \ «G» s \ast \text{wp body } R s + \ «N \ G» s \ast P s \vdash \\
\lambda s. \ «G» s \ast \text{wp body } R s + \ «N \ G» s \ast Q s \\
by (auto intro:le-funI add-left-mono mult-left-mono) \\
\text{also note } \text{fp} \\
\text{finally show } \text{lfp-exp} \ (\lambda R s. \ «G» s \ast \text{wp body } R s + \ «N \ G» s \ast P s) \vdash R \\
\text{using } sR \text{ by (auto intro:lfp-exp-lowerbound)} \\
\text{next} \\
\text{from } \text{hb } sP \text{ show } \text{sound} \ (\text{lfp-exp} \ (\lambda R s. \ «G» s \ast \text{wp body } R s + \ «N \ G» s \ast P s)) \\
by (rule lfp-loop-sound) \\
\text{from } \text{hb } sQ \text{ show } \lambda s. \ «G» s \ast \text{wp body } (\lambda s. \text{bound-of } Q) s + \ «N \ G» s \ast Q s \vdash \lambda s. \text{bound-of } Q \\
by (rule lfp-loop-fp) \\
\text{from } sQ \text{ show } \text{sound} \ (\lambda s. \text{bound-of } Q) \text{ by (auto)} \\
qed \\
qed \\
qed \end{align*}

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

lemmas healthy-intros =
\begin{align*}
\text{healthy-wp-Abort } & \text{nearly-healthy-wlp-Abort} \\
\text{healthy-wp-Skip } & \text{nearly-healthy-wlp-Skip} \\
\text{healthy-wp-Seq } & \text{nearly-healthy-wlp-Seq} \\
\text{healthy-wp-PC } & \text{nearly-healthy-wlp-PC} \\
\text{healthy-wp-DC } & \text{nearly-healthy-wlp-DC} \\
\text{healthy-wp-AC } & \text{nearly-healthy-wlp-AC} \\
\text{healthy-wp-Embed } & \text{nearly-healthy-wlp-Embed} \\
\text{healthy-wp-Apply } & \text{nearly-healthy-wlp-Apply} \\
\text{healthy-wp-SetDC } & \text{nearly-healthy-wlp-SetDC} \\
\text{healthy-wp-SetPC } & \text{nearly-healthy-wlp-SetPC} \\
\text{healthy-wp-Bind } & \text{nearly-healthy-wlp-Bind} \\
\text{healthy-wp-repeat } & \text{nearly-healthy-wlp-repeat} \\
\text{healthy-wp-loop } & \text{nearly-healthy-wlp-loop} \\
\end{align*}

end

4.3 Continuity

theory Continuity imports Healthiness begin

We rely on one additional healthiness property, continuity, which is shown here separately, 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).
4.3. CONTINUITY

**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:

\[ [ \text{bd-cts } t; \bigwedge i. M i \vdash M (\text{Suc } i); \bigwedge i. \text{sound } (M i); \bigwedge i. \text{bounded-by } b (M i) ] \implies t (\text{Sup-exp } (\text{range } M)) = \text{Sup-exp } (\text{range } (t o M)) \]

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

**lemma** bd-ctsI:

\[ (\bigwedge b. (\bigwedge i. M i \vdash M (\text{Suc } i)); \bigwedge i. \text{sound } (M i)); \bigwedge i. \text{bounded-by } b (M i) \implies t (\text{Sup-exp } (\text{range } M)) = \text{Sup-exp } (\text{range } (t o M)) \]

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

A generalised property for transformers of transformers.

**definition** bd-cts-tr :: ('s trans ⇒ 's trans) ⇒ bool

**where** bd-cts-tr T = (∀M. (∀i. le-trans (M i) (M (Suc i)) ∧ feasible (M i)) →

  equiv-trans (T (Sup-trans (M \ M) \ UNIV))) (Sup-trans ((T o M) \ UNIV)))

**lemma** bd-cts-trD:

\[ [ \text{bd-cts-tr } T; \bigwedge i. \text{le-trans } (M i) (M (\text{Suc } i)); \bigwedge i. \text{feasible } (M i) ] \implies \text{equiv-trans } (T (\text{Sup-trans } (M \ M) \ UNIV)) (\text{Sup-trans } ((T o M) \ UNIV)) \]

**by** (simp add: bd-cts-tr-def)

**lemma** bd-cts-trI:

\[ (\bigwedge M. (\bigwedge i. \text{le-trans } (M i) (M (\text{Suc } i)) \implies (\bigwedge i. \text{feasible } (M i)))) \implies \text{equiv-trans } (T (\text{Sup-trans } (M \ M) \ UNIV)) (\text{Sup-trans } ((T o M) \ UNIV)) \]

**⇒** bd-cts-tr T

**by** (simp add: bd-cts-tr-def)

### 4.3.1 Continuity of Primitives

**lemma** cts-wp-Abort:

bd-cts (wp (Abort :: 's prog))

**proof**

- **have** X: range (λi::nat) (s::'s). 0) = {λs. 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-cts (wp Skip)

**by** (rule bd-ctsI, simp add: wp-def Skip-def o-def)

**lemma** cts-wp-Apply:

bd-cts (wp (Apply f))

**proof**
have \( X \subseteq \{ f \mid \forall a. a \in \text{range } M \} = \{ P \mid P \in \text{range } M \} \) 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 \( a :: 'a \Rightarrow 's \text{ prog} \)
assumes \( ca :: \forall s. \text{ bd-cts } (\text{ wp } (a \ f s)) \)
shows \( \text{ bd-cts } (\text{ wp } (\text{ Bind } f a)) \)
proof(rule bd-ctsI)
  fix \( M : \text{ nat } \Rightarrow 's \text{ expect } c :: \text{ real} \)
  assume chain: \( \forall i. M i \vdash M (\text{ Suc } i) \) and \( sM : \forall i. \text{ sound } (M i) \)
  and \( bM : \forall i. \text{ bounded-by } c (M i) \)
  with \( \text{ bd-ctsD}(\text{OF } ca) \)
  have \( \forall s. \text{ wp } (a \ f s) (\text{ Sup-exp } (\text{ range } M)) = \)
    \( \text{ Sup-exp } \ (\text{ range } (\text{ wp } (a \ f s) \circ M)) \)
    by(auto)
  moreover have \( \forall s. \{ \text{ fa } s \mid \text{ fa } s \in \text{ range } (\lambda x. \text{ wp } a (f s) (M x)) \} = \)
    \( \{ \text{ fa } s \mid \text{ fa } s \in \text{ range } (\lambda x s. \text{ wp } a (f s) (M x) s) \} \)
    by(auto)
  ultimately show \( \text{ wp } (\text{ Bind } f a) (\text{ Sup-exp } (\text{ range } M)) = \)
    \( \text{ Sup-exp } (\text{ range } (\text{ wp } (\text{ Bind } f a) \circ M)) \)
    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 \( a b :: 's \text{ prog} \)
assumes \( ca : \text{ bd-cts } (\text{ wp } a) \)
  and \( cb : \text{ bd-cts } (\text{ wp } b) \)
  and \( ha : \text{ healthy } (\text{ wp } a) \)
  and \( hb : \text{ healthy } (\text{ wp } b) \)
shows \( \text{ bd-cts } (\text{ wp } (a \sqcap b)) \)
proof(rule bd-ctsI, rule antisym)
  fix \( M : \text{ nat } \Rightarrow 's \text{ expect } c :: \text{ real} \)
  assume chain: \( \forall i. M i \vdash M (\text{ Suc } i) \) and \( sM : \forall i. \text{ sound } (M i) \)
  and \( bM : \forall i. \text{ bounded-by } c (M i) \)
  from \( ha hb \) have \( \text{ hab } : \text{ healthy } (\text{ wp } (a \sqcap b)) \) by(rule healthy-intros)
  from \( bM \) have \( \text{ leSup } : \forall i. M i \vdash \text{ Sup-exp } (\text{ range } M) \) by(auto intro:Sup-exp-upper)
  from \( sM bM \) have \( \text{ sSup } : \text{ sound } (\text{ Sup-exp } (\text{ range } M)) \) by(auto intro:Sup-exp-sound)
  show \( \text{ Sup-exp } (\text{ range } (\text{ wp } (a \sqcap b) \circ M)) \vdash \text{ wp } (a \sqcap b) (\text{ Sup-exp } (\text{ range } M)) \)
proof(rule Sup-exp-least, clarsimp, rule le-funI)
  fix \( i s \)
  from \( \text{ mono-transD}(\text{OF } \text{ healthy-} \text{ monoD}(\text{OF } \text{ hab}]) \) \( \text{ leSup } sM \) \( s\text{Sup} \)
  have \( \text{ wp } (a \sqcap b) (M i) \vdash \text{ wp } (a \sqcap b) (\text{ Sup-exp } (\text{ range } M)) \) by(auto)
4.3. CONTINUITY

thus $wp\ (a\ 
\cap\ b)\ (M\ i)\ s \leq\ wp\ (a\ 
\cap\ b)\ (\text{Sup-exp}\ (\text{range}\ M))\ s$ by(auto)

from $\text{hab}\ s\ \text{Sup}\ \text{have}\ \text{sound}\ (wp\ (a\ 
\cap\ b)\ (\text{Sup-exp}\ (\text{range}\ M)))$ by(auto)

thus $\text{nneg}\ (wp\ (a\ 
\cap\ b)\ (\text{Sup-exp}\ (\text{range}\ M)))$ by(auto)

qed

from $sM\ bM\ ha\ \text{have}\ \bigwedge i.\ \text{bounded-by}\ c\ (wp\ a\ (M\ i))$ by(auto)

hence $baM:\ \bigwedge i.\ wp\ a\ (M\ i)\ s \leq\ c$ by(auto)

from $sM\ bM\ \text{hb}\ \text{have}\ \bigwedge i.\ \text{bounded-by}\ c\ (wp\ b\ (M\ i))$ by(auto)

hence $bbM:\ \bigwedge i.\ wp\ b\ (M\ i)\ s \leq\ c$ by(auto)

show $wp\ (a\ 
\cap\ b)\ (\text{Sup-exp}\ (\text{range}\ M)) \vdash\ \text{Sup-exp}\ (\text{range}\ (wp\ (a\ 
\cap\ b)\circ M))$

proof(simp add:wp-eval o-def, rule le-funI)

fix $s:\ s'$

from $\text{bd-ctsD}[OF\ ca,\ of\ M,\ \text{OF}\ \text{chain}\ sM\ bM]\ \text{bd-ctsD}[OF\ cb,\ of\ M,\ \text{OF}\ \text{chain}\ sM\ bM]$

have $\text{min}\ (wp\ a\ (\text{Sup-exp}\ (\text{range}\ M)))\ (wp\ b\ (\text{Sup-exp}\ (\text{range}\ M)))\ s =\ \text{min}\ (\text{Sup-exp}\ (\text{range}\ (wp\ a\ o\ M)))\ (\text{Sup-exp}\ (\text{range}\ (wp\ b\ o\ M)))\ s$

by(simp)

also {
have $\{f\ s\ |\ f\ \in\ \text{range}\ (\lambda x.\ wp\ a\ (M\ x))\} =\ \text{range}\ (\lambda i.\ wp\ a\ (M\ i))\ s$

and $\{f\ s\ |\ f\ \in\ \text{range}\ (\lambda x.\ wp\ b\ (M\ x))\} =\ \text{range}\ (\lambda i.\ wp\ b\ (M\ i))\ s$

by(auto)

hence $\text{min}\ (\text{Sup-exp}\ (\text{range}\ (wp\ a\ o\ M)))\ (\text{Sup-exp}\ (\text{range}\ (wp\ b\ o\ M)))\ s$

= $\text{min}\ (\text{Sup}\ (\text{range}\ (\lambda i.\ wp\ a\ (M\ i))\ s))\ (\text{Sup}\ (\text{range}\ (\lambda i.\ wp\ b\ (M\ i))\ s))$

by(simp add:Sup-exp-def o-def)
}

also {
have $(\lambda i.\ wp\ a\ (M\ i))\ s =\ Sup\ (\text{range}\ (\lambda i.\ wp\ a\ (M\ i))\ s)$

proof(rule increasing-LIMSEQ)

fix $n$

from $\text{mono-transD}[OF\ \text{healthy-monoD},\ OF\ ha]\ \text{sm}\ \text{chain}$

show $wp\ a\ (M\ n)\ s \leq\ wp\ a\ (M\ (\text{Suc}\ n))\ s$ by(auto intro:le-funD)

from $baM\ \text{show}\ wp\ a\ (M\ n)\ s \leq\ \text{Sup}\ (\text{range}\ (\lambda i.\ wp\ a\ (M\ i))\ s)$

by(intro cSup-upper bdd-aboveI, auto)

fix $c:\ \text{real}\ \text{assume}\ \text{pe}:\ 0 < e$

from $baM\ \text{have}\ c\ \text{Sup}:\ \text{Sup}\ (\text{range}\ (\lambda i.\ wp\ a\ (M\ i))\ s)\ \in\ \text{closure}\ (\text{range}\ (\lambda i.\ wp\ a\ (M\ i))\ s)$

by(blast intro:closure-contains-Sup)

with $\text{pe}\ \text{obtain}\ y\ \text{where}\ \text{yin}:\ y\ \in\ (\text{range}\ (\lambda i.\ wp\ a\ (M\ i))\ s)$

and $\text{dy}:\ \text{dist}\ y\ (\text{Sup}\ (\text{range}\ (\lambda i.\ wp\ a\ (M\ i))\ s))) <\ e$

by(blast dest:iffD1[OF closure-approachable])

from $\text{yin}\ \text{obtain}\ i\ \text{where}\ y\ =\ wp\ a\ (M\ i)\ s$ by(auto)

with $\text{dy}\ \text{have}\ \text{dist}\ (wp\ a\ (M\ i)\ s)\ (\text{Sup}\ (\text{range}\ (\lambda i.\ wp\ a\ (M\ i))\ s))) <\ e$

by(simp)

moreover from $baM\ \text{have}\ wp\ a\ (M\ i)\ s \leq\ \text{Sup}\ (\text{range}\ (\lambda i.\ wp\ a\ (M\ i))\ s)$

by(intro cSup-upper bdd-aboveI, auto)
ultimately have $\sup (\text{range } (\lambda i. \wp b (M_i) s)) \leq \wp a (M_i) s + e$
by(simp add:dist-real-def)
thus $\exists i. \sup (\text{range } (\lambda i. \wp b (M_i) s)) \leq \wp a (M_i) s + e$ by(auto)
qed

moreover
have $(\lambda i. \wp b (M_i) s) \longrightarrow \sup (\text{range } (\lambda i. \wp b (M_i) s))$
proof(rule increasing-LIMSEQ)
  fix n
  from mono-transD[OF healthy-monoD, OF hb] sM chain
  show $\wp b (M_n) s \leq \wp b (M (\text{Suc } n))$ sby(auto intro:le-funD)
  from bbM show $\wp b (M_n) s \leq \sup (\text{range } (\lambda i. \wp b (M_i) s))$
  by(intro cSup-upper bdd-aboveI, auto)

fix $c :: \text{real}$ assume $pc : 0 < e$
from bbM have cSup: $\sup (\text{range } (\lambda i. \wp b (M_i) s)) \in \text{closure } (\text{range } (\lambda i. \wp b (M_i) s))$
by(blast intro:closure-contains-Sup)
with $pc$ obtain $y$ where $\text{yin: } y \in (\text{range } (\lambda i. \wp b (M_i) s))$
  and $dy$: $\text{dist } y (\sup (\text{range } (\lambda i. \wp b (M_i) s))) < e$
  by(blast dest:iffD[OF closure-approachable])
from $\text{yin}$ obtain $i$ where $y = \wp b (M_i) s$ by(auto)
with $dy$ have $\text{dist } (\wp b (M_i) s) (\sup (\text{range } (\lambda i. \wp b (M_i) s))) < e$
by(simp)
moreover from bbM have $\wp b (M_i) s \leq \sup (\text{range } (\lambda i. \wp b (M_i) s))$
  by(intro cSup-upper bdd-aboveI, auto)
ultimately have $\sup (\text{range } (\lambda i. \wp b (M_i) s)) \leq \wp b (M_i) s + e$
by(simp add:dist-real-def)
thus $\exists i. \sup (\text{range } (\lambda i. \wp b (M_i) s)) \leq \wp b (M_i) s + e$ by(auto)
qed
ultimately have $(\lambda i. \min (\wp a (M_i) s) (\wp b (M_i) s)) \longrightarrow \sup (\text{range } (\lambda i. \wp a (M_i) s)) (\sup (\text{range } (\lambda i. \wp b (M_i) s)))$
by(rule tendsto-min)
moreover have $\text{bdd-above } (\sup (\lambda i. \min (\wp a (M_i) s) (\wp b (M_i) s)))$
proof(intro bdd-aboveI, clarsimp)
fix $i$
have $\min (\wp a (M_i) s) (\wp b (M_i) s) \leq \wp a (M_i) s$ by(auto)
also { 
  from $\text{ha sM}$ have $\text{bounded-by } c$ $\wp a (M_i)$ by(auto)
  hence $\wp a (M_i) s \leq c$ by(auto)
}
finally show $\min (\wp a (M_i) s) (\wp b (M_i) s) \leq c$ .
qed
ultimately
have $\min (\sup (\text{range } (\lambda i. \wp a (M_i) s)) (\sup (\text{range } (\lambda i. \wp b (M_i) s))))$
by(blast intro:LIMSEQ-le-const2 cSup-upper min.mono[OF baM bbM])

also { 
    have range (\lambda i. \min (wp a (M i) s) (wp b (M i) s)) = 
        \{ f s | f \in range (\lambda i s. min (wp a (M i) s) (wp b (M i) s)) \} 
    by (auto) 
    hence Sup (range (\lambda i. \min (wp a (M i) s) (wp b (M i) s)))) = 
        Sup-exp (range (\lambda i s. min (wp a (M i) s) (wp b (M i) s)) s) 
    by (simp add: Sup-exp-def) 
} 
finally show min (wp a (Sup-exp (range M)) s) (wp b (Sup-exp (range M)) s) 
    \leq Sup-exp (range (\lambda i s. min (wp a (M i) s) (wp b (M i) s)))) s . 
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 \Rightarrow 's expect and c :: real 
  assume chain: \forall i. M i \vdash M (Suc i) and sM: \forall i. sound (M i) 
          and bM: \forall 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 \forall i. sound ((wp b o M) i) by (auto) 
    moreover from chain sM 
    have \forall i. (wp b o M) i \vdash (wp b o M) (Suc i) 
        by (auto intro: mono-transD[OF healthy-monoD, OF hb]) 
    moreover from sM bM hb have \forall 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 (\lambda i. wp a (wp b (M i))))) 
    by (simp add: o-def) 
  finally show wp a (wp b (Sup-exp (range M))) = 
        Sup-exp (range (\lambda i. 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 \( \text{wp: unitary } p \)
shows \( \text{bd-cts(\text{wp (PC a p b)})} \)
proof\( (\text{rule bd-ctsI, rule ext, simp add:o-def wp-eval}) \)
fix \( M::\text{nat} \Rightarrow 's \text{ expect and c::real and s:'s} \)
assume chain: \( \bigwedge i. M i \vdash M (\text{Suc } i) \) and \( sM: \bigwedge i. \text{sound (M } i) \)
and \( bM: \bigwedge i. \text{bounded-by c (M } i) \)
from \( sM \) have \( \bigwedge i. \text{nneg (M } i) \) by(auto)
with \( bM \) have \( nc: \theta \leq c \) by(auto)
from chain \( sM bM \) have \( \text{wp a (Sup-exp (range M)) = Sup-exp (range (wp a o M))} \)
by(rule bd-ctsD[OF ca])
hence \( \text{wp a (Sup-exp (range M)) } s = \text{Sup-exp (range (wp a o M)) } s \)
by(simp)
also \( \{ \)
have \( \{f s | f. f \in \text{range (} \lambda x. \text{wp a (M x)}\} = \text{range (} \lambda i. \text{wp a (M } i) s\} \)
by(auto)
hence \( \text{Sup-exp (range (wp a o M)) } s = \text{Sup (range (} \lambda i. \text{wp a (M } i) s\}) \)
by(simp add:Sup-exp-def o-def)
\} 
finally have \( p s * \text{wp a (Sup-exp (range M)) } s = \)
\( p s * \text{Sup (range (} \lambda i. \text{wp a (M } i) s\}) \) by(simp)
also have \( ... = \text{Sup } \{p s * x | x. x \in \text{range (} \lambda i. \text{wp a (M } i) s\}\}
proof\( (\text{rule cSup-mult, blast, clarsimp}) \)
from \( \text{wp show } \theta \leq p s \) by(auto)
fix \( i \)
from \( sM bM ha \) have \( \text{bounded-by c (wp a (M } i) ) \) by(auto)
thus \( \text{wp a (M } i) s \leq c \) by(auto)
qed
also \( \{ \)
have \( \{p s * x | x. x \in \text{range (} \lambda i. \text{wp a (M } i) s\}) = \text{range (} \lambda i. p s * \text{wp a (M } i) s\) \)
by(auto)
hence \( \text{Sup } \{p s * x | x. x \in \text{range (} \lambda i. \text{wp a (M } i) s\}) = \text{Sup (range (} \lambda i. p s * \text{wp a (M } i) s\}) \) by(simp)
\} 
finally have \( p s * \text{wp a (Sup-exp (range M)) } s = \text{Sup (range (} \lambda i. p s * \text{wp a (M } i) s\}) \)
moreover \( \{ \)
from chain \( sM bM \) have \( \text{wp b (Sup-exp (range M)) = Sup-exp (range (wp b o M))} \)
by(rule bd-ctsD[OF cb])
hence \( \text{wp b (Sup-exp (range M)) } s = \text{Sup-exp (range (wp b o M)) } s \)
by(simp)
also \( \{ \)
have \( \{f s | f. f \in \text{range (} \lambda x. \text{wp b (M x)}\} = \text{range (} \lambda i. \text{wp b (M } i) s\) \)
by(auto)
hence \( \text{Sup-exp (range (wp b o M)) } s = \text{Sup (range (} \lambda i. \text{wp b (M } i) s\}) \)
4.3. CONTINUITY

\[
\begin{align*}
\text{by (simp add: Sup-exp-def o-def)} & \\
\text{finally have} \ (1 - p \ s) \ast \ wp \ b \ (\text{Sup-exp} \ (\text{range} \ M)) \ s = & \\
\ (1 - p \ s) \ast \ \text{Sup} \ \{\lambda x. \ wp \ b \ (M \ i) \ s\} \ \text{by (simp)} \\
\text{also have ... =} \ \text{Sup} \ \{\lambda x. \ wp \ b \ (M \ i) \ s\} \\
\text{proof (rule cSup-mul, blast, clarsimp)} & \\
\text{from wp show} \ 0 \leq 1 - p \ s \ \text{by (auto simp: sign-simps)} & \\
\text{fix} \ i & \\
\text{from sM bM hb have bounded-by c (wp b (M \ i)) by (auto)} \\
\text{thus wp b (M \ i) s \leq c by (auto)} & \\
\text{qed} \\
\text{also} & \\
\text{have} \ \{\lambda x. \ wp \ b \ (M \ i) \ s\} = & \\
\text{proof (simp: cSup-upper bdd-aboveI)} & \\
\text{from Suc n have} \ 0 \leq p \ s \ \text{by (auto)} & \\
\text{ultimately have} \ \lambda x. \ wp \ a (M \ i) \ s \leq p \ s \ \text{by (auto intro: mult-left-mono)} & \\
\text{also from wp nc have} \ \lambda x. \ wp \ s \ \text{by (blast intro: mult-right-mono)} & \\
\text{finally have} \ baM: \ \lambda x. \ wp \ a (M \ i) \ s \leq c \ . & \\
\text{have lima: (\lambda x. \ wp \ a (M \ i) \ s) \longrightarrow} \ \text{Sup} \ \{\lambda x. \ wp \ a (M \ i) \ s\} \\
\text{proof (rule increasing-LIMSEQ)} & \\
\text{fix n} & \\
\text{from sM chain healthy-monoD[OF ha]} \ \text{have} \ wp \ a (M \ n) \vdash wp \ a (M \ Suc n) & \\
\text{by (auto)} & \\
\text{with wp show} \ p \ s \ \text{by (blast intro: mult-left-mono)} & \\
\text{from baM show} \ p \ s \ \text{by (intro cSup-upper bdd-aboveI, auto)} & \\
\text{next} & \\
\text{fix c::real} & \\
\text{assume pe:} \ 0 < c & 
\end{align*}
\]
CHAPTER 4. THE PGCL LANGUAGE

from baM have \( \text{Sup}\ (\text{range}\ (\lambda i.\ p\ s\ *\ wp\ a\ (M\ i)\ s)) \in\ \\
closure\ (\text{range}\ (\lambda i.\ p\ s\ *\ wp\ a\ (M\ i)\ s))\) \\
by(blast\ intro:\closure-contains-Sup) 

thm closure-approachable 

with pe obtain \( g\) where \( g\in\text{range}\ (\lambda i.\ p\ s\ *\ wp\ a\ (M\ i)\ s)\) 

and dy: \( \text{dist}\ g\ (\text{Sup}\ (\text{range}\ (\lambda i.\ p\ s\ *\ wp\ a\ (M\ i)\ s))) < e\) 

by(blast\ dest:iffD1[OF\ closure-approachable]) 

from \( g\in\text{range}\ (\lambda i.\ p\ s\ *\ wp\ a\ (M\ i)\ s)\) 

ultimately have \( \text{Sup}\ (\text{range}\ (\lambda i.\ p\ s\ *\ wp\ a\ (M\ i)\ s)) \leq p\ s\ *\ wp\ a\ (M\ i)\ s + e\) 

by(simp add:dist-real-def) 

thus \( \exists i.\ \text{Sup}\ (\text{range}\ (\lambda i.\ p\ s\ *\ wp\ a\ (M\ i)\ s)) \leq p\ s\ *\ wp\ a\ (M\ i)\ s + e\) 

by(auto) 

qed 

from bM sM hb have \( \land i.\ \text{bounded-by}\ c\ (wp\ b\ (M\ i))\) 

by(auto) 

hence \( \land i.\ wp\ b\ (M\ i)\ s \leq c\) 

by(auto) 

moreover from \( wp\) have \( 0 \leq (1\ -\ p\ s)\) 

by(auto simp:sign-simps) 

ultimately have \( \land i.\ (1\ -\ p\ s)\ *\ wp\ b\ (M\ i)\ s \leq (1\ -\ p\ s)\ *\ c\) 

by(auto intro:mult-left-mono) 

also \{ 

from wp have \( 1\ -\ p\ s \leq 1\) 

by(auto) 

with nc have \( (1\ -\ p\ s)\ *\ c \leq 1\ *\ c\) 

by(blast intro:mult-right-mono) 

\} 

also have \( 1\ *\ c = c\) by(simp) 

finally have \( b\ M:\ \land i.\ (1\ -\ p\ s)\ *\ wp\ b\ (M\ i)\ s \leq c.\) 

have \( \text{limb}\ (\lambda i.\ (1\ -\ p\ s)\ *\ wp\ b\ (M\ i)\ s)\) 

\(\longrightarrow\) \(\text{Sup}\ (\text{range}\ (\lambda i.\ (1\ -\ p\ s)\ *\ wp\ b\ (M\ i)\ s))\) 

proof(rule increasing-LIMSEQ) 

fix \( n\) 

from \( sM\ \text{chain}\ healthy-monoD[\text{OF}\ hb]\) have \( wp\ b\ (M\ n)\ \vdash\ wp\ b\ (M\ (Suc\ n))\) 

by(auto) 

moreover from \( wp\) have \( 0 \leq (1\ -\ p\ s)\) 

by(auto simp:sign-simps) 

ultimately show \( (1\ -\ p\ s)\ *\ wp\ b\ (M\ n)\ s \leq (1\ -\ p\ s)\ *\ wp\ b\ (M\ (Suc\ n))\) 

by(blast intro:mult-left-mono) 

from \( b\ M\ \text{show}\ (1\ -\ p\ s)\ *\ wp\ b\ (M\ n)\ s \leq \text{Sup}\ (\text{range}\ (\lambda i.\ (1\ -\ p\ s)\ *\ wp\ b\ (M\ i)\ s))\) 

by(intro cSup-upper bdd-approachable, auto) 

next 

fix \( e::\text{real}\) 

assume \( pe:\ 0 < e\)
4.3. CONTINUITY

from \(bbM\) have \(\text{Sup} (\text{range} (\lambda i. (1 - p s) \ast \text{wp} b (M i) s)) \in \text{closure} (\text{range} (\lambda i. (1 - p s) \ast \text{wp} b (M i) s))\)

by(blast intro:closure-contains-Sup)

with \(pe\) obtain \(y\) where \(yin: y \in \text{range} (\lambda i. (1 - p s) \ast \text{wp} b (M i) s)\)

and \(dy: \text{dist} y (\text{Sup} (\text{range} (\lambda i. (1 - p s) \ast \text{wp} b (M i) s))) < e\)

by(blast dest:iffD1[OF closure-approachable])

from \(yin\) obtain \(i\) where \(y = (1 - p s) \ast \text{wp} b (M i) s\) by(auto)

with \(dy\) have \(\text{dist} ((1 - p s) \ast \text{wp} b (M i) s) (\text{Sup} (\text{range} (\lambda i. (1 - p s) \ast \text{wp} b (M i) s))) < e\)

by(simp)

moreover from \(bbM\)

have \((1 - p s) \ast \text{wp} b (M i) s \leq \text{Sup} (\text{range} (\lambda i. (1 - p s) \ast \text{wp} b (M i) s))\)

by(intro csSup-upper bdd-aboveI, auto)

ultimately have \(\text{Sup} (\text{range} (\lambda i. (1 - p s) \ast \text{wp} b (M i) s)) \leq (1 - p s) \ast \text{wp} b (M i) s + e\) by(auto)

qed

from \(\text{lma}\) \(\text{limb}\) have \((\lambda i. p s \ast \text{wp} a (M i) s + (1 - p s) \ast \text{wp} b (M i) s)\)

\(\text{ultimately}\)

\(\text{have}\) \(\text{Sup} (\text{range} (\lambda i. (1 - p s) \ast \text{wp} b (M i) s))\)

by(rule tendssto-add)

moreover from \(\text{add-mono}[OF baM bbM]\)

have \(\lambda i. p s \ast \text{wp} a (M i) s + (1 - p s) \ast \text{wp} b (M i) s \leq \text{Sup} (\text{range} (\lambda i. p s \ast \text{wp} a (M i) s + (1 - p s) \ast \text{wp} b (M i) s))\)

by(auto)

ultimately have \(\text{Sup} (\text{range} (\lambda i. p s \ast \text{wp} a (M i) s)) + \text{Sup} (\text{range} (\lambda i. (1 - p s) \ast \text{wp} b (M i) s)) \leq \text{Sup} (\text{range} (\lambda i. p s \ast \text{wp} a (M i) s + (1 - p s) \ast \text{wp} b (M i) s))\)

by(blast intro: LIMSEQ-le-const2)

\}

also \{\n
have \(\text{range} (\lambda i. p s \ast \text{wp} a (M i) s + (1 - p s) \ast \text{wp} b (M i) s) = \{f s \mid f \in \text{range} (\lambda x s. p s \ast \text{wp} a (M x) s + (1 - p s) \ast \text{wp} b (M x) s)\}\)

by(auto)

hence \(\text{Sup} (\text{range} (\lambda i. p s \ast \text{wp} a (M i) s + (1 - p s) \ast \text{wp} b (M i) s)) = \text{Sup-exp} (\text{range} (\lambda x s. p s \ast \text{wp} a (M x) s + (1 - p s) \ast \text{wp} b (M x) s)) s\)

by(simp add:Sup-exp-def)

\}

finally\n
have \(p s \ast \text{wp} a (\text{Sup-exp} (\text{range} M)) s + (1 - p s) \ast \text{wp} b (\text{Sup-exp} (\text{range} M))\)

\(s \leq \text{Sup-exp} (\text{range} (\lambda i s. p s \ast \text{wp} a (M i) s + (1 - p s) \ast \text{wp} b (M i) s)) s\)

moreover

have \(\text{Sup-exp} (\text{range} (\lambda i s. p s \ast \text{wp} a (M i) s + (1 - p s) \ast \text{wp} b (M i) s)) s \leq p s \ast \text{wp} a (\text{Sup-exp} (\text{range} M)) s + (1 - p s) \ast \text{wp} b (\text{Sup-exp} (\text{range} M)) s\)
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)
moreover 
   from leSup sSup ha hb have sound (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 sSup 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 ≤ 1 − p s by (auto simp: sign-simps)
   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))
   by (blast intro: add-mono mult-left-mono)
moreover 
   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 wp have ⋀s. 0 ≤ p s ⋀s. 0 ≤ 1 − p s by (auto simp: sign-simps)
   ultimately show unneg (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 (λx 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 (λp. SetPC a (λx. 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
4.3. CONTINUITY

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: setsum.insert)
  also from nz
  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: setsum-distribute)
  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
  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 $a$.

assumes $ca$: bd-cts $a$
  and $ha$: healthy $a$
  and $nnc$: $0 \leq c$

shows bd-cts $(\lambda P. c \ast a P s)$

proof
  \begin{itemize}
  \item fix $M$: nat \Rightarrow 's expect and $d$: real and $s':$
    \begin{itemize}
    \item assume chain: $\forall i. M i \vdash M (Suc i)$ and $sM: \forall i. \text{sound} (M i)$
      and $bM: \forall i. \text{bounded-by} d (M i)$
    \end{itemize}
  \end{itemize}

from $sM$ have $\forall i. \text{nnc} (M i)$ by
  \begin{itemize}
  \item with $bM$ have $\forall i. \text{nnd}: 0 \leq d$ by
  \end{itemize}

from $sM$ have $\forall i. \text{sound} (\text{Sup-exp} (\text{range} M))$ by
  \begin{itemize}
  \item with healthy-scaling $of \text{ha}$
    \begin{itemize}
    \item have $c \ast a (\text{Sup-exp} (\text{range} M)) s = a (\lambda s. c \ast \text{Sup-exp} (\text{range} M) s) s$
      by
        \begin{itemize}
        \item simp add: Sup-exp-sound
        \end{itemize}
    \end{itemize}
  \end{itemize}

also \begin{itemize}
  \item have $\forall s. \{ f \mid f \in \text{range} M \} = \text{range} (\lambda i. M i s)$ by
    \begin{itemize}
    \item simp add: Sup-exp-def
    \end{itemize}
  \item hence $a (\lambda s. c \ast \text{Sup-exp} (\text{range} M) s) s =$
    \begin{itemize}
    \item (\lambda s. \text{Sup} (\{ c \ast x \mid x \in \text{range} (\lambda i. M i s) \}) s$
      by
        \begin{itemize}
        \item subst cSup-mult, blast+
        \end{itemize}
    \end{itemize}
  \end{itemize}

also \begin{itemize}
  \item have $\forall s. \{ c \ast x \mid x \in \text{range} (\lambda i. M i s) \} = \text{range} (\lambda i. c \ast M i s)$ by
    \begin{itemize}
    \item simp add: X
    \end{itemize}
  \end{itemize}

also \begin{itemize}
  \item have $\lambda s. \text{range} (\lambda i. c \ast M i s) = \{ f \mid f \in \text{range} (\lambda i s. c \ast M i s) \}$
    by(auto)
  \item hence $\lambda s. \text{Sup} (\{ c \ast x \mid x \in \text{range} (\lambda i. M i s) \}) s =$
    \begin{itemize}
    \item (\lambda s. \text{Sup} (\{ c \ast x \mid x \in \text{range} (\lambda i. c \ast M i s) \}) s$
      by
        \begin{itemize}
        \item simp add: Sup-exp-def
        \end{itemize}
    \end{itemize}
  \end{itemize}

also \begin{itemize}
  \item have $\lambda s. \text{Sup} (\lambda i. c \ast M i s) = \text{Sup-exp} (\lambda i s. c \ast M i s)$
    by(auto)
  \item hence $\lambda s. \text{Sup} (\{ c \ast x \mid x \in \text{range} (\lambda i. c \ast M i s) \}) s =$
    \begin{itemize}
    \item (\lambda s. \text{Sup} (\lambda i. c \ast M i s) s)$
      by
        \begin{itemize}
        \item simp
        \end{itemize}
    \end{itemize}
  \end{itemize}

also \begin{itemize}
  \item from le-fun $of \text{chain}$
    \begin{itemize}
    \item have $\forall i. (\lambda s. c \ast M i s) \vdash (\lambda s. c \ast M (Suc i) s)$
      by(auto intro: le-fun $of \text{mult-left-mono}$)
    \end{itemize}
  \end{itemize}

moreover from $sM$ nnc

have $\forall i. \text{sound} (\lambda s. c \ast M i s)$
  by(auto intro: sound-intros)
4.3. CONTINUITY

moreover from bM nnc
have ∃i. bounded-by (c * a) (∀s. c * M i s)
  by(auto intro:mult-left-mono)
ultimately
have a (Sup-exp (range (λi s. c * M i s))) =
  Sup-exp (range (a o (λi s. c * M i s)))
  by(rule bd-ctsD[OF ca])

hence a (Sup-exp (range (λi s. c * M i s))) s =
  Sup-exp (range (a o (λi s. c * M i s))) s
  by(auto)
}
also have Sup-exp (range (a o (λi s. c * M i s))) s =
  Sup-exp (range (λx. a (λs. c * M x s))) s
  by(simp add:o-def)
also \{ \from nnc sM
  have ∃x. a (λs. c * M x s) = (λs. c * a (M x) s)
    by(auto intro:scalingD[OF healthy-scalingD, OF ha, symmetric])
  hence Sup-exp (range (λx. a (λs. c * M x s))) s =
    Sup-exp (range (λx s. c * a (M x) s)) s
    by(simp)
\}
finally show c * a (Sup-exp (range M)) s = Sup-exp (range (λx 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 supp: setsup 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 → setsup 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 $\text{sump}: \text{setsum} \ p \ (\text{supp} \ p) \leq 1$
and $xni: x \notin F$

**assume IH**: $\exists p, F = \text{supp} \ p \Rightarrow$
$\text{unitary} \ p \rightarrow \text{setsum} \ p \ (\text{supp} \ p) \leq 1 \rightarrow$
$(\forall x \in \text{supp} \ p. \ \text{bd-cts} (wp \ (a \ x))) \rightarrow$
$(\forall x \in \text{supp} \ p. \ \text{healthy} (wp \ (a \ x))) \rightarrow$
$\text{bd-cts} (wp \ (\text{SetPC} \ a \ (\lambda-. \ p)))$

**from $\forall F \ pstep \ have \ \text{fsupp: finite} \ (\text{supp} \ p) \ by(auto)**

**from $\forall xni \ have \ nzp: p \ x \neq 0 \ by(simp \ add:supp-def)**

**have xy-le-sum:**
$\exists y. y \in \text{supp} \ p \Rightarrow y \neq x \Rightarrow p \ x + p \ y \leq \text{setsum} \ p \ (\text{supp} \ p)$

**proof**
- **fix y assume yin: y \in \text{supp} \ p \ and yne: y \neq x**
- **from up have 0 \leq \text{setsum} \ p \ (\text{supp} \ p - \{x,y\})**
  - **by(auto intro:setsum-nonneg)**
- **hence p \ x + p \ y \leq p \ x + p \ y + \text{setsum} \ p \ (\text{supp} \ p - \{x,y\})**
  - **by(auto)**
- **also {**
  - **from yin yne fsupp**
  - **have p \ y + \text{setsum} \ p \ (\text{supp} \ p - \{x,y\}) = \text{setsum} \ p \ (\text{supp} \ p - \{x\})**
    - **by(subst \ \text{setsum.insert}[\text{symmetric}], \ (\text{blast intro:}setsum.cong)+)**
  - **moreover from \forall xin \ have \ p \ x + \text{setsum} \ p \ (\text{supp} \ p - \{x\}) = \text{setsum} \ p \ (\text{supp} \ p)**
    - **by(subst \ \text{setsum.insert}[\text{symmetric}], \ (\text{blast intro:}setsum.cong)+)**
  - **ultimately have p \ x + p \ y + \text{setsum} \ p \ (\text{supp} \ p - \{x, y\}) = \text{setsum} \ p \ (\text{supp} \ p) \ by(simp)**
  - **}**
- **finally show p \ x + p \ y \leq \text{setsum} \ p \ (\text{supp} \ p).**

**qed**

**have n1p: \forall y. y \in \text{supp} \ p \Rightarrow y \neq x \Rightarrow p \ x \neq 1**

**proof**(rule ccontr, simp)
- **assume px1: p \ x = 1**
- **fix y assume yin: y \in \text{supp} \ p \ and yne: y \neq x**
- **from up have 0 \leq 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 \leq \text{setsum} \ p \ (\text{supp} \ p)**
    - **by(rule xy-le-sum)**
  - **finally show False using sump by(simp)**
  - **qed**

**show \text{bd-cts} (wp \ (\text{SetPC} \ a \ (\lambda-. \ p)))**

**proof**(cases $F = \{\}$)
case True with pstep have supp p = \{x\} by(simp)

hence wp (SetPC a (\lambda p. p x * wp (a x) P s))
  by(simp add:SetPC-sgl wp-def)

moreover {
  from wp ca ha xin have bd-cts (wp (a x)) healthy (wp (a x)) 0 ≤ p x
    by(auto)
  hence bd-cts (\lambda P 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)

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: \forall x∈supp (dist-remove p x). healthy (wp (a x))
  by(auto)

from supp-sub ca have cra: \forall 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] 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 wp have 0 ≤ p y 0 ≤ 1 − p x by(auto simp:sign-simps)
  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 ≤ setsum p (supp p)
  by(auto intro:xy-le-sum)

also note sump

finally have p y ≤ 1 − p x by(auto)

moreover from wp have p x ≤ 1 by(auto)
moreover from \( gin \ yne \) have \( p x \neq 1 \) by(rule n1p)
ultimately show \( p y / (1 - p x) \leq 1 \) by(auto)
qed
qed

from \( xin \) have \( pxn0: p x \neq 0 \) by(auto simp:supp-def)
from \( gin \ yne \) have \( pxn1: p x \neq 1 \) by(rule n1p)

from \( pxn0 \ pxn1 \) have setsum (dist-remove \( p x \)) (supp (dist-remove \( p x \))) =
setsum (dist-remove \( p x \)) (supp \( p - \{x\} \))
by(simp add:supp-dist-remove)
also have \( ... = (\sum y \in supp \( p - \{x\} \). p y / (1 - p x)) \)
by(simp add:dist-remove-def)
also have \( ... = (\sum y \in supp \( p - \{x\} \). p y) / (1 - p x) \)
by(simp add:setsum-divide-distrib)
also 
from \( xin \) have insert \( x \) (supp \( p \)) = supp \( p \) by(auto)
with fsupp have \( p x \leq 1 \) by(auto)

from \( pxn0 \ pxn1 \) have setsum (dist-remove \( p x \)) (supp (dist-remove \( p x \))) =
setsum (dist-remove \( p x \)) (supp \( p - \{x\} \))
by(simp add:supp-dist-remove)
also have \( ... = (\sum y \in supp \( p - \{x\} \). p y) / (1 - p x) \)
by(simp add:setsum-divide-distrib)
also note sump
finally have setsum \( p (supp \( p - \{x\} \)) \leq 1 - p x \) by(auto)
moreover 
from \( up \) have \( p x \leq 1 \) by(auto)
with \( pxn1 \) have \( p x < 1 \) by(auto)
hence \( 0 < 1 - p x \) by(auto)
}
ultimately have setsum \( p (supp \( p - \{x\} \)) / (1 - p x) \leq 1 \)
by(auto)

finally have sdp: setsum (dist-remove \( p x \)) (supp (dist-remove \( p x \))) \leq 1 .

from \( Fsupp udp sdp hra cra IH \) have cts-dr: bd-cts (wp (SetPC a (\( \lambda \). dist-remove \( p x \))))
by(auto)

from \( up \) have upx: unitary (\( \lambda \). \( p x \)) 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 \( \text{assms} \) show ?thesis by(auto)
qed

lemma cts-wp-SetPC:
fixes \( a : 'a \Rightarrow 's \text{ prog} \)
assumes \( ca: \forall x s. x \in (\text{supp} (p s)) \Rightarrow \text{bd-cts} (wp (a x)) \)
4.3. CONTINUITY

and has: $\forall x. x \in (\text{supp} \ (p \ s)) \implies \text{healthy} \ (wp \ (a \ x))$
and wp: $\forall s. \text{unitary} \ (p \ s)$
and sum: $\forall s. \text{setsum} \ (p \ s) \ (\text{supp} \ (p \ s)) \leq 1$
and fsum: $\forall s. \text{finite} \ (\text{supp} \ (p \ s))$
shows bd-cts \ (wp \ (\text{SetPC} \ a \ p))

proof –
from assms have bd-cts \ (wp \ (\text{Bind} \ p \ (\lambda p. \text{SetPC} \ a \ (\lambda \ x. \ p))))
by (iprover intro:cts-\text{wp-Bind} \ cts-\text{wp-SetPC-const})
thus ?thesis by (simp add: SetPC-Bind[symmetric])
qed

lemma wp-SetDC-Bind:
SetDC \ a \ S = \text{Bind} \ S \ (\lambda S. \text{SetDC} \ a \ (\lambda \ -. \ 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 \ (\lambda -. \ \{\} \ x \ S) = a x \prod \ \text{SetDC} \ a \ (\lambda -. \ S)
proof (intro ext, simp add: SetDC-def DC-def del: Inf-image-eq)
      fix ab P s
      from fS have A: finite \ ([\ x \ a \ x \ ab \ P \ s = (\lambda x. \ a \ x \ ab \ P \ s) \ x \ S])
            and B: finite \ ((\lambda x. \ a \ x \ ab \ P \ s) \ x \ S) \ by (auto)
      from neS have C: insert \ (\lambda x. \ a \ x \ ab \ P \ s = \{\} \ x \ S)
            and D: \ (\lambda x. \ a \ x \ ab \ P \ s) \ x \ S \neq \{\} \ by (auto)
      from A \ C \ have \ Inf \ (\lambda x. \ a \ x \ ab \ P \ s = \{\} \ x \ S)
      by (auto intro: Inf-finite)
      also from B \ D \ have \ \ldots = \min \ (\lambda x. \ a \ x \ ab \ P \ s = \{\} \ x \ S)
      by (auto intro: Min-finite)
      also from B \ D \ have \ \ldots = \min \ (\lambda x. \ a \ x \ ab \ P \ s = \{\} \ x \ S)
      by (simp add: Inf-finite Min-finite)
      finally show \ Inf \ (\lambda x. \ a \ x \ ab \ P \ s = \{\} \ x \ S)
      = \min \ (\lambda x. \ a \ x \ ab \ P \ s = \{\} \ x \ S)
      by (auto intro: Inf-finite Min-finite)
      qed

lemma SetDC-singleton:
SetDC \ a \ (\lambda -. \ \{\}) = a x
by (simp add: SetDC-def)

lemma cts-\text{wp-SetDC-const}:
fixes a::\ a \Rightarrow \ \text{\ 's prog}
assumes ca: $\forall x. x \in S \implies \text{bd-cts} \ (wp \ (a \ x))$
and has: $\forall x. x \in S \implies \text{healthy} \ (wp \ (a \ x))$
and fS: finite \ S
and neS: \ S \neq \ \{\}
shows bd-cts \ (wp \ (\text{SetDC} \ a \ (\lambda -. \ S)))
proof –
have finite \ S \implies \ S \neq \{\} \implies
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 (wp (repeat n body))
4.3. CONTINUITY

shows bd-cts-tr \((\lambda x. \text{wp} (\text{body} :: \text{Embed} x \cdot G \oplus \text{Skip}))\) (is bd-cts-tr ?F)

proof\((\text{rule bd-cts-trI}, \text{rule le-trans-antisym})\)

fix \(M::\text{nat} \Rightarrow 's\text{ trans and } \text{h:real}\)
assume chain: \(\bigwedge i. \text{le-trans} (M i) (M (\text{Suc } i))\)
and \(fM::\bigwedge i. \text{feasible} (M i)\)
show \(s\)

assume \(fM\)
from \(s\)
assume \(s\)

fix \(M\)

show \(\forall \text{n: real}\) \(\text{bound-of} P\) \(s\)

hence \(s\)

proof\(\text{by(auto)}\)

from \(fM\) have \(f\sup\): feasible \((\text{Sup-trans} (\text{range } M))\)
\(\text{by(auto intro:feasible-Sup-trans)}\)

moreover from \(s Q fM\) have \(M t Q \vdash \text{Sup-trans} (\text{range } M) Q\)
\(\text{by(auto intro:Sup-trans-upper2)}\)

ultimately have \(\text{wp body} (M t Q) \vdash \text{wp body} (\text{Sup-trans} (\text{range } M) Q)\)
using \(\text{healthy-monoD[of } \text{h} \text{ by(auto)}\)

hence \(\bigwedge s. \text{wp body} (M t Q) s \leq \text{wp body} (\text{Sup-trans} (\text{range } M) Q) s\)
\(\text{by(rule le-funD)}\)

thus \(?F (M t)\) \(\vdash ?F (\text{Sup-trans} (\text{range } M)) Q\)
\(\text{by(intro le-funI, simp add:wp-eval mult-left-mono)}\)

show \(\text{n: real}\) \(\text{bound-of} P\) \(s\)
proof\(\text{by(auto intro:add-nonneg-nonneg simp:wp-eval mult-left-mono)}\)

fix \(s::'s\)
from \(f\sup sQ\) have \(\text{sound} (\text{Sup-trans} (\text{range } M) Q) \text{ by(auto)}\)
with \(\text{h} \text{ have} \text{sound} (\text{wp body} (\text{Sup-trans} (\text{range } M) Q)) \text{ by(auto)}\)

hence \(0 \leq \text{wp body} (\text{Sup-trans} (\text{range } M) Q) s\) \(\text{by(auto)}\)

moreover from \(sQ\) have \(0 \leq Q s\) \(\text{by(auto)}\)

ultimately show \(0 \leq \langle G s \rangle s * \text{wp body} (\text{Sup-trans} (\text{range } M) Q) s + (1 - \langle G s \rangle s) * Q s\)
\(\text{by(auto intro:add-nonneg-nonneg mult:nonneg-nonneg)}\)

qed

next

fix \(P::'s\) expect assume \(sP::\text{sound } P\)
thus \(\text{n: real}\) \(\text{bound-of} P\) \(\text{by(auto)}\)

show \(\forall u \in \text{range} (\lambda x. \text{wp} (\text{body} :: \text{Embed} x \cdot G \oplus \text{Skip}) \circ M)\).

\(\forall R. \text{n: real}\) \(\text{bound-of} P\) \(\text{by(auto)}\)

proof\(\text{by(auto intro conjI simp:wp-eval)}\)

fix \(u::\text{nat}\) and \(R::'s\) expect and \(s::'s\)
assume \(nR::\text{n: real}\) and \(bR::\text{bound-of} P\)

hence \(sR::\text{sound } \text{by(auto)}\)
with \(fM\) have \(sMuR::\text{sound} (M u R) \text{ by(auto)}\)

with \(\text{h}\) have \(\text{sound} (\text{wp body} (M u R)) \text{ by(auto)}\)
hence $0 \leq \text{wp body} (M \ u \ R) \ s$ by(auto)
moreover from nR have $0 \leq R \ s$ by(auto)
ultimately show $0 \leq «G» \ s \ * \ \text{wp body} (M \ u \ R) \ s + (1 - «G» \ s) \ * \ R \ s$
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 \ \leq \ \text{bound-of P} \ by(auto)
morover from bR have $R \ s \ \leq \ \text{bound-of P} \ by(auto)$
ultimately have $«G» \ s \ * \ \text{wp body} (M \ u \ R) \ s + (1 - «G» \ s) \ * \ R \ s \ \leq
«G» \ s \ * \ \text{bound-of P} + (1 - «G» \ s) \ * \ \text{bound-of P}$
by(auto intro:1add mono mult-left mono)
also have ... = bound-of P by(simp add: algebra-simps)
finally show $«G» \ s \ * \ \text{wp body} (M \ u \ R) \ s + (1 - «G» \ s) \ * \ R \ s \ \leq \ \text{bound-of P}$
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)
fix P::'s expect and s::'s
assume sP: sound P
have $\{ t \ P \ | \ t \ \in \ \text{range} \ M \} = \ \text{range} \ (\lambda i. \ M \ i \ P)$
by(blast)
hence wp body (Sup-trans (range M) \ P) \ s \ = \ wp body (Sup-exp (range (\lambda i. \ M \ i \ P))) \ s$
by(simp add: Sup-trans-def)
also {
from sP fM have $\bigwedge i. \ \text{sound} \ (M \ i \ P)$ by(auto)
morover from sP chain have $\bigwedge i. \ M \ i \ P \ \vdash \ M \ (\text{Suc} \ i) \ \ P$ by(auto)
morover {
from sP have bounded-by (bound-of P) P by(auto)
with sP fM have $\bigwedge i. \ \text{bounded-by} \ (\text{bound-of P}) \ (M \ i \ P)$ by(auto)
}
ultimately have wp body (Sup-exp (range (\lambda i. \ M \ i \ P))) \ s =
Sup-exp (range (\lambda i. \ wp body (M \ i \ P))) \ s
by(subst bd-ctsD[OF cb], auto simp: o-def)
}
also have Sup-exp (range (\lambda i. \ wp body (M \ i \ P))) \ s =
Sup \{ f \ s \ | \ f \ \in \ \text{range} \ (\lambda i. \ wp body (M \ i \ P)) \}
by(simp add: Sup-exp-def)
finally have «G» \ s \ * \ \text{wp body} (Sup-trans (range M) \ P) \ s + (1 - «G» \ s) \ * \ P \ s
s =
«G» \ s \ * \ Sup \{ f \ s \ | \ f \ \in \ \text{range} \ (\lambda i. \ wp body (M \ i \ P)) \} + (1 - «G» \ s)
\ * \ P \ s$
by(simp)
also {
from sP fM have $\bigwedge i. \ \text{sound} \ (M \ i \ P)$ by(auto)
morover from sP fM have $\bigwedge i. \ \text{bounded-by} \ (\text{bound-of P}) \ (M \ i \ P)$ by(auto)
ultimately have \( \bigwedge_i \text{bounded-by} (\text{bound-of } P) (wp \text{ body } (M i P)) \) using \( \text{hb} \) by(auto)

hence bound: \( \bigwedge_i \text{wp body } (M i P) s \leq \text{bound-of } P \) by(auto)

moreover

have \( \{x \mapsto \lambda s * x | x. \ x \in \{f s | f. \ f \in \text{range } (\lambda i. \text{wp body } (M i P) ) \}\} = \{x \mapsto \lambda s * f s | f. \ f \in \text{range } (\lambda i. \text{wp body } (M i P) ) \}\) by(auto)

ultimately

have \( \{(G s) * \text{Sup } \{f s | f. \ f \in \text{range } (\lambda i. \text{wp body } (M i P) )\} = \text{Sup } \{(G s) * f s | f. \ f \in \text{range } (\lambda i. \text{wp body } (M i P) )\}\) by(auto)

moreover

have \( \{x + (1 - (G s)) * P s | x. \ x \in \{G s + f s | f. \ f \in \text{range } (\lambda i. \text{wp body } (M i P) )\}\} = \{G s + f s + (1 - (G s)) * P s | f. \ f \in \text{range } (\lambda i. \text{wp body } (M i P) )\}\) by(auto)

moreover from bound \( sP \) have \( \bigwedge_i (G s) * \text{wp body } (M i P) s \leq \text{bound-of } P \) by(auto)

ultimately

have \( \{(G s) * \text{Sup } \{f s | f. \ f \in \text{range } (\lambda i. \text{wp body } (M i P) )\} + (1 - G s) * \text{P s} = \text{Sup } \{(G s) * f s + (1 - (G s)) * P s | f. \ f \in \text{range } (\lambda i. \text{wp body } (M i P) )\}\) by(auto)

ultimately

have \( \{(G s) * \text{Sup } \{f s | f. \ f \in \text{range } (\lambda i. \text{wp body } (M i P) )\} + (1 - G s) * \text{P s} = \text{Sup } \{(G s) * f s + (1 - (G s)) * P s | f. \ f \in \text{range } (\lambda i. \text{wp body } (M i P) )\}\) by(auto)

also \{ \}

have \( \bigwedge_i (G s) * \text{wp body } (M i P) s + (1 - G s) * \text{P s} = \text{((\lambda x. \text{wp body } (x i G) \oplus \text{Skip})) o M i P s} \) by(auto)

also have \( \bigwedge i. \ i \leq \text{Sup } \{f s | f. \ f \in \{t P | t. \ t \in \text{range } ((\lambda x. \text{wp body } (x i G) \oplus \text{Skip}) o M) \}\} \) by(auto)

proof(intro cSup-upper bdd-aboveI, blast, clarsimp simp:wp-eval)

fix \( i \)

from \( sP \) have \( bP: \text{bounded-by } (\text{bound-of } P) \) by(auto)

with \( sP M \) have \( \text{sound } (M i P) \) by(auto)

with \( bP \) have \( \text{bounded-by } (\text{bound-of } P) (\text{wp body } (M i P)) \) by(auto)

with \( bP \) have \( \text{bounded-by } (\text{bound-of } P) (\text{wp body } (M i P)) \) by(auto)

hence \( (G s) * \text{wp body } (M i P) s + (1 - G s) * \text{P s} \leq \text{(bound-of } P) \) by(auto)

\( (G s) * (\text{bound-of } P) + (1 - G s) * (\text{bound-of } P) \) by(auto)

(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 + (I ~ «G») s * P s ≤ bound-of P .

qed

finally have Sup {«G» s * f s + (I ~ «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)) o 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)) o M))} =

Sup-trans (range ((λx. wp (body ;; Embed x « G » ⊕ Skip)) o M)) P s
by (simp add: Sup-trans-def Sup-exp-def)

finally show «G» s * wp body (Sup-trans (range M) P) s + (I ~ «G») s * P s ≤

Sup-trans (range ((λx. wp (body ;; Embed x « G » ⊕ Skip)) o 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 » 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** \texttt{lfp-wp-loop-unfold}:  
\texttt{fixes body::'s prog}  
\texttt{assumes hb: healthy (wp body)}  
\texttt{and sP: sound P}  
\texttt{shows lfp-exp (\lambda Q s. «G» s * wp body Q s + «N G» s * P s)} =  
\texttt{(\lambda s. «G» s * wp body (lfp-exp (\lambda Q s. «G» s * wp body Q s + «N G» s * P s)) s + «N G» s * P s)}

proof (rule lfp-exp-unfold)

\texttt{from assms show mono-trans (\lambda Q s. «G» s * wp body Q s + «N G» s * P s)}  
\texttt{by (blast intro:wp-loop-step-mono-trans)}

\texttt{from sP show sound (\lambda s. «G» s * wp body (\lambda s. bound-of P) s + «N G» s * P s)}  
\texttt{by (auto)}

\texttt{fix Q::'s expect}  
\texttt{assume sound Q}  
\texttt{with assms show sound (\lambda s. «G» s * wp body Q s + «N G» s * P s)}  
\texttt{by (intro wp-loop-step-sound [unfolded wp-eval, simplified, folded negate-embed], auto)}

qed

**lemma** \texttt{wp-loop-step-unitary}:  
\texttt{fixes body::'s prog}  
\texttt{assumes hb: healthy (wp body)}  
\texttt{and uP: unitary P and uQ: unitary Q}  
\texttt{shows unitary (\lambda s. «G» s * wp body Q s + «N G» s * P s)}

proof (intro unitaryI2 nnegI bounded-byI)

\texttt{fix s::'s}  
\texttt{from uQ hb have uuQ: unitary (wp body Q) by (auto)}

\texttt{with uP have 0 \leq wp body Q s 0 \leq P s by (auto)}

\texttt{thus 0 \leq «G» s * wp body Q s + «N G» s * P s by (auto intro:add-nonneg-nonneg mult-nonneg-nonneg)}

\texttt{from uP uuQ have wp body Q s \leq 1 P s \leq 1 by (auto)}

\texttt{hence «G» s * wp body Q s + «N G» s * P s \leq «G» s * 1 + «N G» s * 1 by (blast intro:add-mono mult-left-mono)}

\texttt{also have ... = 1 by (simp add:negate-embed)}

\texttt{finally show «G» s * wp body Q s + «N G» s * P s \leq 1}}

qed

**lemma** \texttt{lfp-loop-unitary}:  
\texttt{fixes body::'s prog}  
\texttt{assumes hb: healthy (wp body)}  
\texttt{and uP: unitary P}  
\texttt{shows unitary (lfp-exp (\lambda Q 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:**

```plaintext
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: \( \forall S. [ \forall x \in S. P (\text{fst } x) (\text{snd } x); \forall x \in S. \text{feasible (fst } x); \forall x \in S. \forall Q. \text{unitary } Q \implies \text{unitary (snd } x Q) ] \implies P (\text{Sup-trans (fst } S)) (\text{Inf-utrans (snd } S)) \)
— The property holds at limit points.
and IH: \( \forall t u. [ \forall Q. \text{unitary } Q \implies \text{unitary (u Q)} ] \implies P (\text{wp (body ;; Embed t « G » ⊕ Skip)}) (\text{wlp (body ;; Embed u « G » ⊕ Skip)}) \)
— The inductive step. The property is preserved by a single loop iteration.
and P-equiv: \( \forall t t' u u'. [ P t u; \text{equiv-trans } t t' \implies \text{equiv-utrans } u u' ] \implies P t' u' \)
— 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)**

```plaintext
let ?X t = wp (body ;; Embed t « G » ⊕ Skip)
let ?Y t = wlp (body ;; Embed t « G » ⊕ Skip)
let ?M = \{ x. P (\text{fst } x) (\text{snd } x) \land \text{feasible (fst } x) \land (\forall Q. \text{unitary } Q \implies \text{unitary (snd } x Q)) \land \text{le-trans (fst } x) (\text{lfp-trans } ?X) \land \text{le-utrans (gfp-trans } ?Y) (\text{snd } x) \}

have fSup: feasible (\text{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 (\text{fst } ?M) Q ⊢ λs. b
  unfolding Sup-trans-def
  using nQ bQ by (auto intro!: Sup-exp-least)
show λs. 0 ⊢ Sup-trans (\text{fst } ?M) Q
proof (cases)
  assume empty: ?M = {}
  show ?thesis by (simp add: Sup-trans-def Sup-def-empty)
next
  assume ne: ?M ≠ {}
  hence ?thesis by (rule nonempty-witness)
  then obtain x where xin: x ∈ ?M by (rule exE)
  hence ffix: feasible (\text{fst } x) by (simp)
```
with \( nQ \ bQ \) have \( \lambda s. \, 0 \not\vdash \operatorname{fst} \ x \ Q \) by\( (\text{auto}) \)
also from \( \text{xin} \) have \( \operatorname{fst} \ x \ Q \vdash \operatorname{Sup-trans} \ (\operatorname{fst} \ ?M) \ Q \)
\text{apply}(\text{intro \ Sup-trans-upper2}[OF \ \operatorname{imageI} - nQ \ bQ], \ assumption) 
\text{apply}(\text{clarsimp}, \ \text{blast intro: sound-nneg}[OF \ \text{feasible-sound}] \ \text{feasible-boundedD})
done
finally show \( \lambda s. \, 0 \not\vdash \operatorname{Sup-trans} \ (\operatorname{fst} \ ?M) \ Q \).
qed

qed
finally show \( \lambda s. \emptyset \vdash \text{Inf-utrans (snd } \ ?M) \ P \).

qed

qed

have \( \text{wp-loop-mono: } \forall t u. \left[ \text{le-trans } t u ; \forall P. \text{sound } P \Rightarrow \text{sound } (t P) ; \right] \left[ \forall P. \text{sound } P \Rightarrow \text{sound } (u P) \right] \Rightarrow \text{le-trans } (?X t) (?X u) \)

proof(intro le-transI le-funI, simp add:wp-eval)

fix \( t u : \cdot \text{ trans and } P : : \cdot \text{ expect and } s : \cdot \cdot \)

assume \( \text{le: le-trans } t u \)

and \( \text{st: } \forall P. \text{sound } P \Rightarrow \text{sound } (t P) \)

and \( \text{su: } \forall P. \text{sound } P \Rightarrow \text{sound } (u P) \)

and \( \text{sp: sound } P \)

hence \( \text{sound } (t P) \text{ sound } (u P) \)

by(auto)

with \( \text{healthy-monoD}[OF hwlp] \) \( \text{le sP} \) have \( \text{wp body } (t P) \vdash \text{wp body } (u P) \)

by(auto)

from \( \text{hwp} \) have \( \text{hX: } \forall t. \text{healthy } t \Rightarrow \text{healthy } (?X t) \)

by(auto intro:healthy-intros)

from \( \text{hwlp} \) have \( \text{hY: } \forall t. \text{nearly-healthy } t \Rightarrow \text{nearly-healthy } (?Y t) \)

by(auto intro:healthy-intros)

have \( \text{PLimit: } P \ (\text{Sup-trans (fst } \ ?M)) \ (\text{Inf-utrans (snd } \ ?M)) \)

by(auto intro:Limit)

have \( \text{feasible-lfp-loop: } \)

feasible \( (\text{lfp-trans } ?X) \)

proof(intro feasibleI bounded-byI2 nnegI2,
4.4. CONTINUITY AND INDUCTION FOR LOOPS

\[\text{smp-all add:wp-Loop1[unfolded wp-eval] soundI2 hwp}\]
\[\text{fix P::'s expect and b::real}\]
\[\text{assume bP: bounded-by b P and nP: nneg P}\]
\[\text{hence sP: sound P by(auto)}\]
\[\text{show lfp-exp (λQ s. « G » s * wp body Q s + « N G » s * P s) \(\vdash\) λs. b}\]
\[\text{proof(intro lfp-exp-lowerbound le-funI)}\]
\[\text{fix s::'s}\]
\[\text{from bP nP have nnb: 0 ≤ b by(auto)}\]
\[\text{hence sound (λs. b) bounded-by b (λs. b) by(auto)}\]
\[\text{with hwp have bounded-by b (wp body (λs. b)) by(auto)}\]
\[\text{with bP have wp body (λs. b) s ≤ b P s ≤ b by(auto)}\]
\[\text{hence «G» s * wp body (λs. b) s + «N G» s * P s ≤ «G» s * b + «N G» s * b}\]
\[\text{by(auto intro:add-mono mult-left-mono)}\]
\[\text{thus «G» s * wp body (λs. b) s + «N G» s * P s ≤ b}\]
\[\text{by(simp add:negate-embed algebra-simps)}\]
\[\text{from nnb show sound (λs. b) by(auto)}\]
\[\text{qed}\]
\[\text{from hwp sP show λs. 0 \(\vdash\) lfp-exp (λQ s. « G » s * wp body Q s + « N G » s * P s)}\]
\[\text{(s * b)}\]
\[\text{by(blast intro!:lfp-exp-greatest lfp-loop-fp)}\]
\[\text{qed}\]
\[\text{have unitary-gfp:}\]
\[\forall P. \text{unitary P } \Rightarrow \text{unitary (gfp-trans ?Y P)}\]
\[\text{proof(intro unitaryI2 nnegI2 bounded-byI2,}\]
\[\text{smp-all add:wp-Loop1[unfolded wp-eval] hwp)}\]
\[\text{fix P::'s expect}\]
\[\text{assume aP: unitary P}\]
\[\text{show λs. 0 \(\vdash\) gfp-exp (λQ s. « G » s * wp body Q s + « N G » s * P s)}\]
\[\text{proof(rule gfp-exp-upperbound[OF le-funI])}\]
\[\text{fix s::'}s\]
\[\text{from hwp aP have 0 ≤ wp body (λs. 0) s 0 ≤ P s by(auto dest!:unitary-sound)}\]
\[\text{thus 0 ≤ «G» s * wp body (λs. 0) s + «N G» s * P s}\]
\[\text{by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)}\]
\[\text{show unitary (λs. 0) by(auto)}\]
\[\text{qed}\]
\[\text{show gfp-exp (λQ s. « G » s * wp body Q s + « N G » s * P s) \(\vdash\) λs. 1}\]
\[\text{by(auto intro:gfp-exp-least)}\]
\[\text{qed}\]
\[\text{have fX:}\]
\[\forall t. \text{feasible t } \Rightarrow \text{feasible (?X t)}\]
\[\text{proof(intro feasibleI nnegI bounded-byI, simp-all add:wp-eval)}\]
\[\text{fix t::'s trans and Q::'s expect and b::real and s::'}s\]
\[\text{assume ft: feasible t and bQ: bounded-by b Q and nQ: nneg Q}\]
\[\text{hence nneg (t Q) bounded-by b (t Q) by(auto)}\]
\[\text{moreover hence stQ: sound (t Q) by(auto)}\]
\[\text{ultimately have wp body (t Q) s ≤ b using hwp by(auto)}\]
CHAPTER 4. THE PGCL LANGUAGE

moreover from \( bQ \) have \( Q s \leq b \) by(auto)
ultimately have \( \langle G \rangle s \ast \text{wp body} (t Q) s + (1 - \langle G \rangle s) \ast Q s \leq (\langle G \rangle s \ast b + (1 - \lg s) \ast b)
\)
by(auto intro:add-mono mult-left-mono)
thus \( \langle G \rangle s \ast \text{wp body} (t Q) s + (1 - \langle G \rangle s) \ast Q s \leq b \)
by(simp add:algebra-simps)

from \( nQ stQ \) have \( 0 \leq \text{wp body} (t Q) s 0 \leq Q s \) by(auto)
thus \( 0 \leq \langle G \rangle s \ast \text{wp body} (t Q) s + (1 - \langle G \rangle s) \ast Q s \)
by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg) qed

have \( uY: \)
\( \forall t P. (\forall P. \text{unitary } P \Longrightarrow \text{unitary } (t P)) \Longrightarrow \text{unitary } P \Longrightarrow \text{unitary } (?Y t P) \)
proof(auto unitaryI2 nnegI bounded-byI simp-all add:wp-eval)
fix \( t::s \text{ trans and } P::s \text{ expect and } s::s \)
assume \( ut: \forall P. \text{unitary } P \Longrightarrow \text{unitary } (t P) \)
and \( up: \text{unitary } P \)

hence \( utP: \text{unitary } (t P) \) by(auto)

with \( hwlp \) have \( ubtP: \text{unitary } (\text{wp body} (t P)) \) by(auto)

with \( up \) have \( 0 \leq P s 0 \leq \text{wp body} (t P) s \) by(auto)

thus \( 0 \leq \langle G \rangle s \ast \text{wp body} (t P) s + (1 - \langle G \rangle s) \ast P s \)
by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)

from \( uP ubtP \) have \( P s \leq 1 \) \( \text{wp body} (t P) s \leq 1 \) by(auto)

hence \( \langle G \rangle s \ast \text{wp body} (t P) s + (1 - \langle G \rangle s) \ast P s \leq \langle G \rangle s \ast 1 + (1 - \langle G \rangle s) \ast 1 \)
by(blast intro:add-mono mult-left-mono)

also have \( \ldots = 1 \) by(simp add:algebra-simps)
finally show \( \langle G \rangle s \ast \text{wp body} (t P) s + (1 - \langle G \rangle s) \ast P s \leq 1 \).
qed

have \( fu-lfp: \text{le-trans } (\text{Sup-trans } (\text{fst } ?M)) \) \( (\text{lfp-trans } ?X) \)
using feasile-nnegD[OF feasile-lfp-loop]
by(intro le-transI[OF Sup-trans-least2], blast+)

hence \( \text{le-trans } (?X (\text{Sup-trans } (\text{fst } ?M))) \) \( (?X (\text{lfp-trans } ?X)) \)
by(auto intro:wp-loop-mono feasible-sound[OF fSup]
feasile-sound[OF feasile-lfp-loop])
also have \( \text{equiv-trans } \ldots (\text{lfp-trans } ?X)\)
proof(rule iffD1[OF equiv-trans-comm, OF lfp-trans-unfold], iprover intro:wp-loop-mono)
fix \( t::s \text{ trans and } P::s \text{ expect} \)
assume \( st: \forall Q. \text{sound } Q \Longrightarrow \text{sound } (t Q) \)
and \( sP: \text{sound } P \)
show sound \( (?X t P) \)
proof(auto unitaryI2 bounded-byI nnegI, simp-all add:wp-eval)
fix \( s::s \)
from \( sP st hwlp \) have \( 0 \leq P s 0 \leq \text{wp body} (t P) s \) by(auto)
thus \( 0 \leq \langle G \rangle s \ast \text{wp body} (t P) s + (1 - \langle G \rangle s) \ast P s \)
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 I − «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 fu-gfp: le-utrans (gfp-trans ?Y) (Inf-utrans (snd ' ?M))
by(auto intro:Inf-utrans-greatest unitary-gfp)

have equin-utrans (gfp-trans ?Y) (?Y (gfp-trans ?Y))
by(auto intro!:gfp-trans-unfold wp-loop-mono uY)
also from fu-gfp have le-utrans (?Y (gfp-trans ?Y)) (?Y (Inf-utrans (snd ' ?M)))
by(auto intro:wp-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(prover intro:IH)
moreover from fSup have feasible (?X (Sup-trans (fst ' ?M))) by(rule fX)
moreover have λP. 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 image[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 →
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\^i \oplus Skip\)) \^ i) (\(\lambda\)P s. 0)

lemma iterates-0[simp]:
iterates body G 0 = (\(\lambda\)P s. 0)
by(simp add:iterates-def)

lemma iterates-Suc[simp]:
iterates body G (Suc i) = wp (body ;; Embed (iterates body G i) "G\^i \oplus 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)
4.4. CONTINUITY AND INDUCTION FOR LOOPS

**Lemma 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 (auto)

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 = λx. 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 f(Sup: 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)
4.4. CONTINUITY AND INDUCTION FOR LOOPS

negate-embed)

have equiv-trans (?F ?X) ?X
unfolding 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 \( \bigwedge i. \) le-trans (iterates body G i) (wp do G \( \rightarrow \) body od)
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
unfolding wp-eval
proof (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)
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 \( \Rightarrow \) ?Y P
proof (rule Sup-trans-least2, clarsimp)
    show \( \forall u \in \text{range} (\lambda x. \wp (\text{body} :: \text{Embed} x \langle G, \oplus \text{Skip} \rangle) \circ \text{iterates body} G) \).
        \( \forall R. \) nneg R \( \land \) bounded-by (bound-of P) R \( \rightarrow \)
        nneg (u R) \( \land \) 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 \( \forall P. \) sound P \( \Rightarrow \) sound (iterates body G i P) by (auto)
moreover note hb
ultimately have sound (wp (\text{body} :: \text{Embed} (iterates body G i) \langle G, \oplus \text{Skip} \rangle) Q)
by (iprover intro: wp-loop-step-sound)
thus nneg (wp (\text{body} :: \text{Embed} (iterates body G i) \langle G, \oplus \text{Skip} \rangle) Q)
    by (auto)
from nQ bQ iterates-healthy [OF hb] hb
show bounded-by (bound-of P) (wp (\text{body} :: \text{Embed} (iterates body G i)
\[ \langle G \rangle \oplus \text{Skip} \] Q \\
\text{by (rule } wp\text{-loop-step-bounded)} \\
\text{qed} \\
\text{from } sP \text{ show } \text{nneq } P \text{ bounded-by } (\text{bound-of } P) \ P \text{ by (auto)} \\
\text{next} \\
\text{fix } Q:\text{\textquoteleft}'s \text{ expect} \\
\text{assume } nQ: \text{nneq } Q \text{ and } bQ: \text{bounded-by } (\text{bound-of } P) \ Q \\
\text{hence } \text{sound } Q \text{ by (auto)} \\
\text{with } fSup \text{ have } \text{sound } (\text{Sup-trans } (\text{range } (\text{iterates body } G)) \ Q) \text{ by (auto)} \\
\text{thus } \text{nneq } (\text{Sup-trans } (\text{range } (\text{iterates body } G)) \ Q) \text{ by (auto)} \\
\text{fix } i \\
\text{show } \text{wp } (\text{body } ;; \text{Embed } (\text{iterates body } G \ i) \ \langle G \rangle \oplus \text{Skip}) \ Q \vdash \\
\text{Sup-trans } (\text{range } (\text{iterates body } G)) \ Q \\
\text{proof (rule } \text{Sup-trans-upper2}[\text{OF } - - nQ bQ]) \\
\text{from } \text{iterates-healthy}[\text{OF } hb] \\
\text{show } \forall u \in \text{range } (\text{iterates body } G). \\
\forall R. \text{nneq } R \land \text{bounded-by } (\text{bound-of } P) \ R \rightarrow \\
\text{nneq } (u \ R) \land \text{bounded-by } (\text{bound-of } P) \ (u \ R) \\
\text{by (auto)} \\
\text{have } \text{wp } (\text{body } ;; \text{Embed } (\text{iterates body } G \ i) \ \langle G \rangle \oplus \text{Skip}) = \text{iterates body } G \ (\text{Suc } i) \\
\text{by (simp)} \\
\text{also have } \ldots \in \text{range } (\text{iterates body } G) \\
\text{by (blast)} \\
\text{finally show } \text{wp } (\text{body } ;; \text{Embed } (\text{iterates body } G \ i) \ \langle G \rangle \oplus \text{Skip}) \in \\
\text{range } (\text{iterates body } G) . \\
\text{qed} \\
\text{qed} \\
\text{finally show } \text{le-trans } (\text{?F } ?Y) \ ?Y . \\
\text{fix } P:\text{\textquoteleft}'s \text{ expect} \\
\text{assume } \text{sound } P \\
\text{with } fSup \text{ show } \text{sound } (\text{?Y } P) \text{ by (auto)} \\
\text{qed} \\
\text{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::\text{'s prog} \\
assumes hh: \text{healthy } (\text{wp body}) \\
and cb: \text{bd-cts } (\text{wp body}) \\
and sP: \text{sound } P \\
shows (\lambda i. \text{iterates body } G \ i \ P) s \dashv \vdash \text{wp } (\text{do } G \rightarrow \text{body od }) \ P \ s \\
\text{proof} – \\
\text{let } ?X = \{ f \ s | f. f \in \{ t \ P \mid t. t \in \text{range } (\text{iterates body } G) \} \} \\
\text{have } \text{closure-Sup}: \text{Sup } ?X \in \text{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 \( \forall j. \) bounded-by (bound-of P) (iterates body G j P)
by (auto)
thus iterates body G i P s \( \leq \) bound-of P by (auto)
qed

have \( \lambda i. \) iterates body G i P s \[\rightarrow\] Sup \( \{ f s \mid f. f \in \{ t P \mid t. t \in \text{range} (\text{iterates body G}) \} \}
proof (rule LIMSEQ-I)
fix \( r :: \text{real} \)
assume posr: \( 0 < r \)
with closure-Sup obtain y where yin: y \( \in \) ?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 \( i \leq j \rightarrow \text{le-trans} (\text{iterates body G i}) (\text{iterates body G j}) \)
proof (induct j, simp, clarify)
fix k
assume IH: \( i \leq k \rightarrow \text{le-trans} (\text{iterates body G i}) (\text{iterates body G k}) \)
and le: \( i \leq \text{Suc} k \)
show \( \text{le-trans} (\text{iterates body G i}) (\text{iterates body G (Suc k)}) \)
proof (cases i = Suc k, simp)
assume i \( \neq \) Suc k
with le have \( i \leq k \) by (auto)
with IH have \( \text{le-trans} (\text{iterates body G i}) (\text{iterates body G k}) \) by (auto)
also note iterates-increasing[of hb]
finally show \( \text{le-trans} (\text{iterates body G i}) (\text{iterates body G (Suc k)}) . \)
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 \( \forall j. \) bounded-by (bound-of P) (iterates body G j P)
by (auto)
hence \( \forall j. \) iterates body G j P s \( \leq \) bound-of P by (auto)
hence \( \forall j. \) iterates body G j P s \( \leq \) Sup ?X
by (intro cSup-upper bdd-aboveI, auto)
}
ultimately have \( \forall j. i \leq j \implies \)
\begin{align*}
\| \text{iterates body G j P s} - \text{Sup ?X} \| & \leq \\
\| \text{iterates body G i P s} - \text{Sup ?X} \| & \leq \\
\end{align*}
by (auto)
also from ey yit have \( \| \text{iterates body G i P s} - \text{Sup ?X} \| < r \)
by (simp add: dist-real-def)

finally show \( \exists \text{no.} \, \forall n \geq \text{no.} \) \( \text{norm (iterates body } G \text{ n } P \text{ s} - \)
\( \text{Sup} \{ f \text{ s} \mid f \in \{ t \text{ P} \mid t \in \text{range (iterates body } G) \} \}) \)
\(< r \)

by (auto)

qed

moreover

from \( \text{hb cb sP} \) have \( \text{wp do } G \rightarrow \text{body od } P \text{ s} = \text{Sup-trans (range (iterates body } G) )} \)
\( P \text{ s} \)

by (simp add: equiv-transD [OF lfp-iterates])

moreover have \( \ldots = \text{Sup} \{ f \text{ s} \mid f \in \{ t \text{ P} \mid t \in \text{range (iterates body } G) \} \}) \)

by (simp add: Sup-trans-def Sup-exp-def)

ultimately show \( \text{thesis by (simp)} \)

qed

The iterates themselves are all continuous.

**Lemma cts-iterates:**

fixes \( \text{body}' : \text{s prog} \)

assumes \( \text{hb: healthy (wp body)} \)

and \( \text{cb: bd-cts (wp body)} \)

shows \( \text{bd-cts (iterates body } G \text{ i)} \)

proof (induct i, simp-all)

have \( \text{range (} \lambda (n::nat) \text{ (} s::'s) \text{. } 0::\text{real}) = \{ \lambda s. 0::\text{real}\} \)

by (auto)

thus \( \text{bd-cts (} \lambda P (s::'s). 0) \)

by (intro bd-ctsI, simp add: o-def Sup-exp-def)

next

fix \( i \)

assume \( \text{IH: bd-cts (iterates body } G \text{ i)} \)

thus \( \text{bd-cts (wp (body :: Embed (iterates body } G \text{ i)} \ominus G \ominus \text{Skip)} \)} \)


healthy-intros iterates-healthy cb hb)

qed

Therefore so is the loop itself.

**Lemma cts-wp-loop:**

fixes \( \text{body}' : \text{s prog} \)

assumes \( \text{hb: healthy (wp body)} \)

and \( \text{cb: bd-cts (wp body)} \)

shows \( \text{bd-cts (wp do } G \rightarrow \text{body od)} \)

proof (rule bd-ctsI)

fix \( M::\text{nat} \Rightarrow 's \text{ expect and b::real} \)

assume \( \text{chain: } \bigwedge i. M i \vdash M (\text{Suc } i) \)

and \( sM: \bigwedge i. \text{sound (} M i \) \)

and \( bM: \bigwedge i. \text{bounded-by } b (M i) \)

from \( sM \text{ bM iterates-healthy[OF } \text{hb}] \)

have \( \bigwedge j i. \text{bounded-by } b (\text{iterates body } G \text{ i } (M j)) \)

by (blast)

hence \( iB: \bigwedge j i. \text{iterates body } G \text{ i } (M j) \text{ s } \leq b \)

by (auto)
from \(sM \vdash \text{m} \text{Sup}: \text{sound} (\text{Sup-exp} (\text{range} M))\) 
by(auto intro;Sup-exp-sound) 
with lfp-iterates[OF \(\text{OF hb cb}\)] 
\text{have wp do } G \rightarrow \text{body od} (\text{Sup-exp} (\text{range} M)) = 
\text{Sup-trans} (\text{range} (\text{iterates body} G)) (\text{Sup-exp} (\text{range} M))
by(simp add:equite-transD) 
also \{
from chain \(sM \vdash \text{m}\)
\text{have } \\(\forall i. \text{iterates body} G \circ M (\text{range} (\text{iterates body} G \circ M)) = \text{Sup-exp} (\text{range} (\text{iterates body} G \circ M))\)
by(black intro;bd-ctsD cts-iterates[OF \(\text{OF hb cb}\)])
\text{hence } (\text{Sup-exp} (\text{range} M)) \mid t. t \in \text{range} (\text{iterates body} G)\} = 
\text{Sup-exp} (\text{range} (\text{iterates body} G)) (\text{range} (\text{iterates body} G)\} = 
\text{Sup-exp} (\text{range} (\text{iterates body} G)) \mid t. t \in \text{range} (\text{iterates body} G)\}
by(simp add:Sup-trans-def) 
\}
also \{
\text{have } \\(\forall s. \{ f s \mid \exists t. f = (\lambda s. \text{Sup} \{ f s \mid f \in \text{range} (t \circ M)\}) \land \text{t} \in \text{range} (\text{iterates body} G)\} = 
\text{range} (\lambda i. \text{Sup} (\text{iterates body} G \circ M j) s)))\}
(is \(\forall s. \forall s = \forall y s\))
\text{proof(intro antisym subsetI)}
fix \(s x\)
assume \(x \in \forall s x\)
then obtain \(t\) where \(\text{rwx}: x = \text{Sup} \{ f s \mid f \in \text{range} (t \circ M)\}\) 
and \(t \in \text{range} (\text{iterates body} G)\) by(auto)
then obtain \(i\) where \(t = \text{iterates body} G \circ M j\) by(auto)
with \(\text{rux}\) have \(x = \text{Sup} \{ f s \mid f \in \text{range} (\lambda j. \text{iterates body} G \circ M j)\}\)
by(simp add:o-def)
moreover have \(\{ f s \mid f \in \text{range} (\lambda j. \text{iterates body} G \circ M j)\} = \text{range} (\lambda j. \text{iterates body} G \circ M j) s)\) by(auto)
ultimately have \(x = \text{Sup} (\text{range} (\lambda j. \text{iterates body} G \circ M j) s))\)
by(simp)
thus \(x \in \text{range} (\lambda i. \text{Sup} (\text{iterates body} G \circ M j) s))\)
by(auto)
next
fix \(s x\)
assume \(x \in \forall y s\)
then obtain \(i\) where \(A: x = \text{Sup} (\text{range} (\lambda j. \text{iterates body} G \circ M j) s)\)
by(auto)
have \(\forall s. \{ f s \mid f \in \text{range} (\lambda j. \text{iterates body} G \circ M j)\} = \text{range} (\lambda j. \text{iterates body} G \circ M j) s)\) by(auto)
\text{hence } B: (\lambda s. \text{Sup} (\text{iterates body} G \circ M j s)) = (\lambda s. \text{Sup} \{ f s \mid f \in \text{range} (\text{iterates body} G \circ M)\})
by(simp add:o-def)
have $C$: iterates body $G$ $i \in \text{range (iterates body $G$)}$ by(auto)

have $\exists f, x = f s \land$

$(\exists t. f = (\lambda s. \text{Sup } \{f s \mid f \in \text{range (t o M)}\}) \land t \in \text{range (iterates body $G$)})$

by(iprover intro:A B C)

thus $x \in ?X s$ by(simp)

qed

hence $\text{Sup-exp } \{\text{Sup-exp (range (t o M))} \mid t. t \in \text{range (iterates body $G$)}\} =$

$(\lambda s. \text{Sup } \text{(range (}\lambda i. \text{Sup } \text{(range (}\lambda j. \text{iterates body G i (M j) s)})))))$

by(simp add:Sup-exp-def)

} also have $(\lambda s. \text{Sup } \text{(range (}\lambda i. \text{Sup } \text{(range (}\lambda j. \text{iterates body G i (M j) s)})))) =$

$(\lambda s. \text{Sup } \text{(range (}\lambda (i.j). \text{iterates body G i (M j) s)})))$

(is $?X = ?Y$)

proof(rule ext, rule antisym)

fix $s::'s$

show $?Y s \leq ?X s$

proof(rule cSup-least, blast, clarify)

fix $i j::nat$

from $iB$ have iterates body $G$ $i (M j) s \leq \text{Sup } \text{(range (}\lambda j. \text{iterates body G i (M j) s))}$

by(intro cSup-upper bdd-aboveI, auto)

also from $iB$ have $... \leq \text{Sup } \text{(range (}\lambda i. \text{Sup } \text{(range (}\lambda j. \text{iterates body G i (M j) s)}))))$

by(intro cSup-upper cSup-least bdd-aboveI, (blast intro:cSup-least)+)

finally show iterates body $G$ $i (M j) s \leq$

$\text{Sup } \text{(range (}\lambda i. \text{Sup } \text{(range (}\lambda j. \text{iterates body G i (M j) s)}))})$

. qed

have $\lambda i j. \text{iterates body G i (M j) s} \leq$

$\text{Sup } \text{(range (}\lambda (i.j). \text{iterates body G i (M j) s)})$

by(rule cSup-upper, auto intro:iB)

thus $?X s \leq ?Y s$

by(intro cSup-least, blast, clarify, simp del:Sup-image-eq, blast intro:cSup-least)

qed

also have $... = (\lambda s. \text{Sup } \text{(range (}\lambda j. \text{Sup } \text{(range (}\lambda i. \text{iterates body G i (M j) s}))}))$

(is $?X = ?Y$)

proof(rule ext, rule antisym)

fix $s::'s$

have $\lambda i j. \text{iterates body G i (M j) s} \leq$

$\text{Sup } \text{(range (}\lambda (i.j). \text{iterates body G i (M j) s)})$

by(rule cSup-upper, auto intro:iB)

thus $?Y s \leq ?X s$

by(intro cSup-least, blast, clarify, simp del:Sup-image-eq, blast intro:cSup-least)

show $?X s \leq ?Y s$

proof(rule cSup-least, blast, clarify)

fix $i j::nat$

from $iB$ have iterates body $G$ $i (M j) s \leq \text{Sup } \text{(range (}\lambda i. \text{iterates body G i (M j) s})$
4.4. CONTINUITY AND INDUCTION FOR LOOPS

\[(M \, j) \, s]\)

by (intro cSup-upper bdd-aboveI, auto)
also from \(iB\) have \(\ldots \leq \text{Sup} (\text{range} \, (\lambda i. \text{Sup} (\text{range} \, (\lambda i. \text{iterates body G} \, i \, (M \, j) \, s))))\)

finally show \(\text{iterates body G} \, i \, (M \, j) \, s \leq \text{Sup} (\text{range} \, (\lambda i. \text{Sup} (\text{range} \, (\lambda i. \text{iterates body G} \, i \, (M \, j) \, s))))\)

qed

also {

have \(\bigwedge s. \text{range} \, (\lambda i. \text{Sup} (\text{range} \, (\lambda i. \text{iterates body G} \, i \, (M \, j) \, s)))) = \{ \{ f \, s \mid f \in \text{range} \, ((\lambda P \, s. \text{Sup} \, \{ f \, s \mid (\exists t. f = t \, P \land t \in \text{range} \, (\text{iterates body G}))\}) \circ M) \}) (i \, s \in i \, Y \, s)\)

proof (intro antisym subsetI)

fix \(s\, x\)
assume \(x \in i \, Y \, s\)
then obtain \(j\) where \(\text{rwx: } x = \text{Sup} (\text{range} \, (\lambda i. \text{iterates body G} \, i \, (M \, j) \, s))\)

by (auto)

morerover {

have \(\bigwedge s. \text{range} \, (\lambda i. \text{iterates body G} \, i \, (M \, j) \, s) = \{ \{ f \, s \mid (\exists t. f = t \, (M \, j) \land t \in \text{range} \, (\text{iterates body G}))\}\)
by (auto)

hence \((\lambda s. \text{Sup} (\text{range} \, (\lambda i. \text{iterates body G} \, i \, (M \, j) \, s))) \in \text{range} \, ((\lambda P \, s. \text{Sup} \, \{ f \, s \mid (\exists t. f = t \, P \land t \in \text{range} \, (\text{iterates body G}))\}) \circ M)\)
by (simp add: o_def)
}

ultimately show \(x \in i \, Y \, s\) by (auto)

next

fix \(s\, x\)
assume \(x \in i \, Y \, s\)
then obtain \(P\) where \(\text{rwx: } x = P \, s\)
and Pin: \(P \in \text{range} \, ((\lambda P \, s. \text{Sup} \, \{ f \, s \mid (\exists t. f = t \, P \land t \in \text{range} \, (\text{iterates body G}))\}) \circ M)\)
by (auto)

then obtain \(j\) where \(P = (\lambda s. \text{Sup} \, \{ f \, s \mid (\exists t. f = t \, (M \, j) \land t \in \text{range} \, (\text{iterates body G}))\})\)
by (auto)

also {

have \(\bigwedge s. \{ f \, s \mid (\exists t. f = t \, (M \, j) \land t \in \text{range} \, (\text{iterates body G}))\} = \text{range} \, (\lambda i. \text{iterates body G} \, i \, (M \, j) \, s)\) by (auto)

hence \( (\lambda s. \text{Sup} \, \{ f \, s \mid (\exists t. f = t \, (M \, j) \land t \in \text{range} \, (\text{iterates body G}))\}) = (\lambda s. \text{Sup} (\text{range} \, (\lambda i. \text{iterates body G} \, i \, (M \, j) \, s)))\)
by (simp)
}

finally have \(x = \text{Sup} (\text{range} \, (\lambda i. \text{iterates body G} \, i \, (M \, j) \, s))\)
by (simp add: rwx)

thus \(x \in i \, Y \, s\) by (simp)

qed
hence \((\lambda s. \text{Sup} (\text{range} (\lambda j . \text{Sup} (\text{range} (\lambda i . \text{iterates body G i} (M j) s))))) = \text{Sup-exp} (\text{range} (\text{Sup-trans} (\text{range} (\text{iterates body G})) o M))\)
by(simp add:Sup-exp-def Sup-trans-def)
}
also have \(\text{Sup-exp} (\text{range} (\text{Sup-trans} (\text{range} (\text{iterates body G})) o M)) = \text{Sup-exp} (\text{range} (\text{wp do G } \rightarrow \text{body od o M}))\)
by(simp add:o-def equiv-transD[OF lfp-iterates, OF hb cb, OF sM])
finally show \(\text{wp do G } \rightarrow \text{body od} (\text{Sup-exp} (\text{range} M)) = \text{Sup-exp} (\text{range} (\text{wp do G } \rightarrow \text{body od o M}))\).
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 algebraic 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)
4.5. SUBLINEARITY

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 by
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 by 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 \[ ... = (P \ s \ a \ * \ wp \ x \ Q \ s + P \ s \ b \ * \ wp \ x \ R \ s) + \]
\[ (1 - P \ s) * (a * wp y \ Q s + (1 - P \ s) * b * wp y \ R s) \circ c \]
by(simp add:ac-simps)
also
have \[ ... = P \ s * (a * wp x \ Q s + b * wp x \ R s) + \]
\[ (1 - P \ s) * (a * wp y \ Q s + b * wp y \ R s) \circ \]
\[ (P \ s * c + (1 - P \ s) * c) \]
by(simp add:field-simps)
also
have \[ ... \leq (P \ s * (a * wp x \ Q s + b * wp x \ R s) \circ P \ s * c) + \]
\[ ((1 - P \ s) * (a * wp y \ Q s + b * wp y \ R s) \circ (1 - P \ s) * c) \]
by(rule tminus-add-mono)
also
\{
  from \( uP \) have \( 0 \leq P \ s \) and \( 0 \leq 1 - P \ s \)
  by(auto simp:sign-simps)
  hence \( (P \ s * (a * wp x \ Q s + b * wp x \ R s) \circ P \ s * c) + \)
\[ ((1 - P \ s) * (a * wp y \ Q s + b * wp y \ R s) \circ (1 - P \ s) * c) = \]
\[ P \ s * (a * wp x \ Q s + b * wp x \ R s \circ c) + \]
\[ (1 - P \ s) * (a * wp y \ Q s + b * wp y \ R s \circ c) \]
by(simp add:tminus-left-distrib)
\}
also
\{
  from \( sQ \ sR \) \( nna \ nnb \ nnc \ slx \)
  have \( a * \ wp x \ Q s + b * \ wp x \ R s \circ c \leq \)
\[ \wp x (\lambda s. a * Q s + b * R s \circ c) s \]
  by(blast)
  moreover
  from \( sQ \ sR \) \( nna \ nnb \ nnc \ sly \)
  have \( a * \ wp y \ Q s + b * \ wp y \ R s \circ c \leq \)
\[ \wp y (\lambda s. a * Q s + b * R s \circ c) s \]
  by(blast)
  moreover
  from \( uP \) have \( 0 \leq P \ s \) and \( 0 \leq 1 - P \ s \)
  by(auto simp:sign-simps)
  ultimately
  have \( P \ s * (a * \ wp x \ Q s + b * \ wp x \ R s \circ c) + \)
\[ (1 - P \ s) * (a * \ wp y \ Q s + b * \ wp y \ R s \circ c) \leq \]
\[ P \ s * \ wp x (\lambda s. a * Q s + b * R s \circ c) s + \]
\[ (1 - P \ s) * \ wp y (\lambda s. a * Q s + b * R s \circ c) s \]
by(blast intro:add-mono mult-left-mono)
\}
finally
show \( a * (P \ s * \ wp x \ Q s + (1 - P \ s) * \ wp y \ Q s) + \)
\[ b * (P \ s * \ wp x \ R s + (1 - P \ s) * \ wp y \ R s) \circ c \leq \]
\[ P \ s * \ wp x (\lambda s. a * Q s + b * R s \circ c) s + \]
\[ (1 - P \ s) * \ wp y (\lambda s. a * Q s + b * R s \circ c) s \]
qed

lemma sublinear-wp-DC:
fixes $x$: prog
assumes slx: sublinear $(\text{wp } x)$ and sly: sublinear $(\text{wp } y)$
shows sublinear $(\text{wp } (x \sqcup y))$
proof (rule sublinearI, simp only: wp-eval)
  fix $R$: real and $Q$: real and $s$: s
shows sublinear $(\text{wp } (x \sqcup y))$
proof (rule sublinearI, simp only: wp-eval)
  fix $R$: real and $Q$: real and $s$: s
  and a::real and b::real and c::real
assume slR: sound $R$ and sQ: sound $Q$
  and nna: $0 \leq a$ and nnb: $0 \leq b$ and nnc: $0 \leq c$
from nna nnb
have $a \cdot \min (\text{wp } x \ Q \ s + b \cdot \text{wp } x \ R \ s) +$
  $b \cdot \min (\text{wp } y \ Q \ s + b \cdot \text{wp } y \ R \ s) \sqcup c =$
  $\min (a \cdot \text{wp } x \ Q \ s + b \cdot \text{wp } y \ Q \ s) +$
  $\min (b \cdot \text{wp } x \ R \ s + b \cdot \text{wp } y \ R \ s) \sqcup c$
proof (simp add: min-distrib)
also
have ... $\leq \min (a \cdot \text{wp } x \ Q \ s + b \cdot \text{wp } x \ R \ s)$
  $(a \cdot \text{wp } y \ Q \ s + b \cdot \text{wp } y \ R \ s) \sqcup c$
proof (auto intro!: tminus-left-mono)
also
have ... $= \min (a \cdot \text{wp } x \ Q \ s + b \cdot \text{wp } x \ R \ s \sqcup c)$
  $(a \cdot \text{wp } y \ Q \ s + b \cdot \text{wp } y \ R \ s \sqcup c)$
proof (rule min-tminus-distrib)
also
  from slx sQ sR nna nnb nnc
  have $a \cdot \text{wp } x \ Q \ s + b \cdot \text{wp } x \ R \ s \sqcup c \leq$
    $\text{wp } (\lambda s. a \cdot Q \ s + b \cdot R \ s \sqcup c) \ s$
proof (blast)
moreover
  from sly sQ sR nna nnb nnc
  have $a \cdot \text{wp } y \ Q \ s + b \cdot \text{wp } y \ R \ s \sqcup c \leq$
    $\text{wp } (\lambda s. a \cdot Q \ s + b \cdot R \ s \sqcup c) \ s$
proof (blast)
ultimately
  have $\min (a \cdot \text{wp } x \ Q \ s + b \cdot \text{wp } x \ R \ s \sqcup c)$
    $(a \cdot \text{wp } y \ Q \ s + b \cdot \text{wp } y \ R \ s \sqcup c) \leq$
    $\min (\text{wp } x (\lambda s. a \cdot Q \ s + b \cdot R \ s \sqcup c) \ s)$
    $(\text{wp } y (\lambda s. a \cdot Q \ s + b \cdot R \ s \sqcup c) \ s)$
proof (auto)
}
finally show $a \cdot \min (\text{wp } x \ Q \ s) (\text{wp } y \ Q \ s) +$
  $b \cdot \min (\text{wp } x \ R \ s) (\text{wp } y \ R \ s) \sqcup c \leq$
  $\min (\text{wp } x (\lambda s. a \cdot Q \ s + b \cdot R \ s \sqcup c) \ s)$
  $(\text{wp } y (\lambda s. a \cdot Q \ s + b \cdot R \ s \sqcup c) \ s)$
proof (auto)
qed

As for continuity, we insist on a finite support.

lemma sublinear-wp-SetPC:
fixes $p$: 'a ⇒ 's prog
assumes \( \text{slp}: \forall s. a \in \text{supp} \{ P s \} \implies \text{sublinear} \{ \text{wp} \{ p a \} \} \)
and \( \text{sum:} \forall s. (\sum a \in \text{supp} \{ P s \} . P s a) \leq 1 \)
and \( \text{nnP:} \forall s. 0 \leq P s a \)
and \( \text{fin:} \forall s. \text{finite} \{ \text{supp} \{ P s \} \} \)
shows \( \text{sublinear} \{ \text{wp} \{ \text{SetPC} p P \} \} \)
proof(rule sublinearI, simp add:wp-eval)
fix \( R::s \Rightarrow \text{real} \) and \( Q::s \Rightarrow \text{real} \) and \( s::s \)
assume \( sR\): sound \( R \) and \( sQ\): sound \( Q \)
and \( nna\): \( 0 \leq a \) and \( nnb\): \( 0 \leq b \) and \( nnc\): \( 0 \leq c \)
have \( a \ast (\sum a \in \text{supp} \{ P s \} . P s a') \ast \text{wp} \{ p a' \} Q s + b \ast \text{wp} \{ p a' \} R s \) +
\( b \ast (\sum a \in \text{supp} \{ P s \} . P s a') \ast \text{wp} \{ p a' \} R s \) \ominus c =
\( (\sum a \in \text{supp} \{ P s \} . P s a') \ast (a \ast \text{wp} \{ p a' \} Q s + b \ast \text{wp} \{ p a' \} R s) \) \ominus c 
by(simp add:field-simps setsum-right-distrib setsum-distrib)
also have \( \ldots \leq \)
\( (\sum a \in \text{supp} \{ P s \} . P s a') \ast (a \ast \text{wp} \{ p a' \} Q s + b \ast \text{wp} \{ p a' \} R s) \) \ominus (\sum a \in \text{supp} \{ P s \} . P s a') \ast c 
proof(rule tminus-right-antimono)
have \( (\sum a \in \text{supp} \{ P s \} . P s a') \ast c \leq (\sum a \in \text{supp} \{ P s \} . P s a') \ast c 
by(simp add:setsum-left-distrib)
also from \( \text{sum} \) and \( \text{nnc} \) have \( \ldots \leq 1 \ast c 
by(rule \text{mult-right-mono})
finally show \( (\sum a \in \text{supp} \{ P s \} . P s a') \ast c \leq c \) by(simp)
qed
also from \( \text{fin} \)
have \( \ldots \leq (\sum a \in \text{supp} \{ P s \} . P s a') \ast (a \ast \text{wp} \{ p a' \} Q s + b \ast \text{wp} \{ p a' \} R s) \) \ominus (P s a') \ast c 
by(blast intro:tminus-setsum-mono)
also have \( \ldots = (\sum a \in \text{supp} \{ P s \} . P s a') \ast (a \ast \text{wp} \{ p a' \} Q s + b \ast \text{wp} \{ p a' \} R s \ominus c) 
\) 
by(simp add:nnP tminus-left-distrib)
also \{ 
from \( \text{slp} sQ sR nna nnb nnc 
\)
have \( \forall a', a' \in \text{supp} \{ P s \} \Rightarrow a \ast \text{wp} \{ p a' \} Q s + b \ast \text{wp} \{ p a' \} R s \ominus c \leq \text{wp} \{ p a' \} (\lambda s. a \ast Q s + b \ast R s \ominus c) s 
\) 
by(blast)
with \( \text{nnP} \)
\( \forall a' \in \text{supp} \{ P s \} . P s a' \ast (a \ast \text{wp} \{ p a' \} Q s + b \ast \text{wp} \{ p a' \} R s \ominus c) \) 
\( \leq \)
\( (\sum a \in \text{supp} \{ P s \} . P s a' \ast \text{wp} \{ p a' \} (\lambda s. a \ast Q s + b \ast R s \ominus c) s) \) 
by(blast intro:setsum-mono mult-left-mono) \}
finally
show \( a \ast (\sum a \in \text{supp} \{ P s \} . P s a' \ast \text{wp} \{ p a' \} Q s) + 
\) 
\( b \ast (\sum a \in \text{supp} \{ P s \} . P s a' \ast \text{wp} \{ p a' \} R s) \) \ominus c \leq 
\( (\sum a \in \text{supp} \{ P s \} . P s a' \ast \text{wp} \{ p a' \} (\lambda s. a \ast Q s + b \ast R s \ominus c) s) \) . 
qed

lemma \( \text{sublinear-wp-SetDC} \):
4.5. SUBLINEARITY

fixes p::'a ⇒ 's prog
assumes slp: \( \forall s. a \in S \Rightarrow \text{sublinear} (wp (p a)) \)
   and hp: \( \forall s. a \in S \Rightarrow \text{healthy} (wp (p a)) \)
   and ne: \( \forall s. S \neq \emptyset \)
shows sublinear (wp (SetDC p S))

proof (rule sublinearI, simp add: wp-eval del: Inf-image-eq, rule cInf-greatest)
fix P::'s ⇒ real and Q::'s ⇒ real and s::'s and y x y
and a::real and b::real and c::real
assume slp: 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 slp 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 slp 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: minus-left-mono add-mono)
also from xin slp slp sQ nna nnb nnc
have ... ≤ wp (p x) (λs. a * P s + b * Q s ⊕ c) s
   by (blast)

finally show a * Inf ((λa. wp (p a) P s) ' S s) + b * Inf ((λa. wp (p a) Q s) ' S s) ⊕ c ≤ y
   by (simp add: rwy)
qed

lemma sublinear-wp-Embed:
sublinear t ⇒ sublinear (wp (Embed t))
by (simp add: wp-eval)

lemma sublinear-wp-repeat:
\[ \text{sublinear} (wp p); \text{healthy} (wp p) \] ⇒ \text{sublinear} (wp (repeat n p))
by (induct n, simp-all add: sublinear-wp-Seq sublinear-wp-Skip healthy-wp-repeat)

lemma sublinear-wp-Bind:
\[ \text{\\&s. sublinear} (wp (a (f s))) \] ⇒ \text{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 → body od))

**Proof**

- **have** \( \forall P \ s. \ \text{sound} \ P \rightarrow \text{wp} \ (\text{do} \ G \rightarrow \text{body} \ \text{od}) \ P \ s \odot 1 \leq \text{wp} \ (\text{do} \ G \rightarrow \text{body} \ \text{od}) \ (\lambda s. \ P \ s \odot 1) \ s \)

**Proof** (rule loop-induct [OF hb nhb], safe)

- **fix** \( S::(’s \ \text{trans} \times ’s \ \text{trans}) \ \text{set} \ \text{and} \ P::’s \ \text{expect} \ \text{and} \ s::’s \)
- **assume** saS: \( \forall x \in S. \ \forall P \ s. \ \text{sound} \ P \rightarrow \text{fst} x \ P \ s \odot 1 \leq \text{fst} x \ (\lambda s. \ P \ s \odot 1) \ s \)
  - **and** sP: \( \text{sound} \ P \)
  - **and** fS: \( \forall x \in S. \ \text{feasible} \ (\text{fst} x) \)

from sP have sPm: \( \text{sound} \ (\lambda s. \ P \ s \odot 1) \) by (auto intro: tminus-sound)

- **have** \( \text{nnSup}: \lambda s. \ 0 \leq \text{Sup-trans} \ (\text{fst} \ s) \ (\lambda s. \ P \ s \odot 1) \ s \)
- **proof** (cases \( S::[] \), simp add: Sup-trans-def Sup-exp-def)
  - **fix** s
  - **assume** \( S \neq \{\} \)
  - **then obtain** x where \( x::x \in S \) by (auto)
    - **with** fS sPm have \( 0 \leq \text{fst} x \ (\lambda s. \ P \ s \odot 1) \ s \) by (auto)
    - **also from** xin fS sPm have \( ... \leq \text{Sup-trans} \ (\text{fst} \ s) \ (\lambda s. \ P \ s \odot 1) \ s \)
      - by (auto intro: le-funD [OF Sup-trans-upper2])
    - finally show \( \text{thesis} \ s \).
  qed

- **have** \( \lambda x s. \ \text{fst} x \ P \ s \leq (\text{fst} x \ P \ s \odot 1) + 1 \) by (simp add: tminus-def)
  - **also from** saS sP
    - **have** \( \lambda x s. \ x::x \in S \implies (\text{fst} x \ P \ s \odot 1) + 1 \leq \text{fst} x \ (\lambda s. \ P \ s \odot 1) \ s + 1 \)
      - by (auto intro: add-right-mono)
    - **also**
      - **from** sP have \( \text{sound} \ (\lambda s. \ P \ s \odot 1) \) by (auto intro: tminus-sound)
        - **with** fS have \( \lambda x s. \ x::x \in S \implies \text{fst} x \ (\lambda s. \ P \ s \odot 1) \ s + 1 \leq \text{Sup-trans} \ (\text{fst} \ s) \ (\lambda s. \ P \ s \odot 1) \ s + 1 \)
          - by (blast intro!: add-right-mono le-funD [OF Sup-trans-upper2])
    - }
  - finally have \( le::\lambda s. \ \forall x \in S. \ \text{fst} x \ P \ s \leq \text{Sup-trans} \ (\text{fst} \ s) \ (\lambda s. \ P \ s \odot 1) \ s + 1 \)
    - by (auto)
  - **moreover from** nnSup have \( \lambda s. \ 0 \leq \text{Sup-trans} \ (\text{fst} \ s) \ (\lambda s. \ P \ s \odot 1) \ s + 1 \)
    - 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 ⊇ 1) + (1 − «G» s) * (P s ⊇ 1)
    by(simp add:tminus-left-distrib)
also {
  from ft sP have wp body (t P) s ⊇ 1 ≤ wp body (λs. t P s ⊇ 1) s
    by(auto intro:sub-distribD[OF sdb])
  also {
    from IH sP have λs. t P s ⊇ 1 ⊬ t (λs. P s ⊇ 1) by(auto)
    with sP ft s2 s3 have wp body (λs. t P s ⊇ 1) s ≤ wp body (t (λs. P s ⊇ 1)) s
      by(blast intro:le-funD[OF mono-transD, OF healthy-monoD, OF hbl])
  }
  finally have «G» s * (wp body (t P) s ⊇ 1) + (1 − «G» s) * (P s ⊇ 1) ≤
    «G» s * wp body (t (λs. P s ⊇ 1)) s + (1 − «G» s) * (P s ⊇ 1)
    by(auto intro:add-right-mono mult-left-mono)
}
finally show «G» s * wp body (t P) s + (1 − «G» s) * P s ⊇ 1 ≤
\(\llangle G \rrangle s \ast \text{wp body } (t (\lambda s. P s \ominus 1)) s + (1 - \llangle G \rrangle s) \ast (P s \ominus 1)\).

\hspace{1cm} \text{qed}

\text{next}

\text{fix } t t' :: s \text{ trans and } P :: s \text{ expect and } s :: s

\text{assume } IH: \forall P s. \text{ sound } P \rightarrow t P s \ominus 1 \leq t (\lambda a. P a \ominus 1) s

\hspace{1cm} \text{and eq: equiv-trans } t t' \text{ and } sP: \text{ sound } P

\text{from } sP \text{ have } t' P s \ominus 1 = t P s \ominus 1 \text{ by } (\text{simp add: equiv-transD}[OF eq])

\text{also from } sP IH \text{ have } \ldots \leq t (\lambda s. P s \ominus 1) s \text{ by } (\text{auto})

\hspace{1cm} \text{also } \{

\hspace{2cm} \text{from } sP \text{ have } \text{sound } (\lambda s. P s \ominus 1) \text{ by } (\text{simp add: tminus-sound})

\hspace{4cm} \text{hence } t (\lambda s. P s \ominus 1) s = t' (\lambda s. P s \ominus 1) s \text{ by } (\text{simp add: equiv-transD}[OF eq])

\hspace{1cm} \}

\hspace{1cm} \text{finally show } t' P s \ominus 1 \leq t' (\lambda s. P s \ominus 1) s .

\hspace{1cm} \text{qed}

\hspace{1cm} \text{thus } ?\text{thesis by } (\text{auto intro!: sub-distrib1})

\hspace{1cm} \text{qed}

For sub-additivity, we again use the limit-of-iterates characterisation. Firstly, all iterates are sublinear:

\textbf{lemma sublinear-iterates:}

\text{assumes } hb: \text{ healthy } (\text{wp body})

\hspace{1cm} \text{and } sb: \text{ sublinear } (\text{wp body})

\text{shows } \text{sublinear } (\text{iterates } body G i)

\hspace{1cm} \text{by } (\text{induct } i, \text{ 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.

\textbf{lemma sub-add-wp-loop:}

\text{fixes body::s prog}

\text{assumes } sb: \text{ sublinear } (\text{wp body})

\hspace{1cm} \text{and cb: } \text{bd-cts } (\text{wp body})

\hspace{2cm} \text{and hwp: } \text{healthy } (\text{wp body})

\text{shows } \text{sub-add } (\text{wp } (\text{do } G \rightarrow \text{body od}))

\text{proof}

\text{fix } P Q :: s \text{ expect and } s :: s

\text{assume } sP: \text{ sound } P \text{ and } sQ: \text{ sound } Q

\text{from } hwp cb sP \text{ have } (\lambda i. \text{iterates } body G i P s) \rightarrow\rightarrow wp do G \rightarrow body od P s

\hspace{1cm} \text{by } (\text{rule loop-iterates})

\hspace{1cm} \text{moreover}

\text{from } hwp cb sQ \text{ have } (\lambda i. \text{iterates } body G i Q s) \rightarrow\rightarrow wp do G \rightarrow body od Q s

\hspace{1cm} \text{by } (\text{rule loop-iterates})

\hspace{1cm} \text{ultimately}

\text{have } (\lambda i. \text{iterates } body G i P s + \text{iterates } body G i Q s) \rightarrow\rightarrow
wp do G $\rightarrow$ body od P s + wp do G $\rightarrow$ 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 $\bigwedge i. \text{iterates body } G i P s + \text{iterates body } G i Q s \leq \text{iterates body } G i (\lambda s. P s + Q s)$ s 
by (rule sub-addD)
}
moreover 
from sP sQ have sound $(\lambda s. P s + Q s)$ by (blast intro:sound-intros)
with hwp cb have $(\lambda i. \text{iterates body } G i (\lambda s. P s + Q s) s) \Rightarrow \Rightarrow \Rightarrow \Rightarrow wp do G $\rightarrow$ body od $(\lambda s. P s + Q s) s$
by (blast intro:loop-iterates)
}
ultimately
show wp do G $\rightarrow$ body od P s + wp do G $\rightarrow$ body od Q s $\leq$ wp do G $\rightarrow$ body od $(\lambda s. P s + Q s) s$
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 $\rightarrow$ 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::'
  assume sP: sound P and sQ: sound Q
  note hb = well-def-wp-healthy[OF wb]
  from addb sP sQ
  have wp b (λs. P s + Q s) = (λs. wp b P s + wp b Q s)
    by(blast dest:additiveD)
  with adda sP sQ hb
  show wp a (wp b (λs. 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) ] ⇒ additive (wp (a ≺ b))
  by(rule additiveI, simp add:additiveD field-simps wp-eval)

DC is not additive.
4.6. DETERMINISM

**Lemma** *additive-wp-SetPC:*
\[
\begin{align*}
\forall x. s. x \in \text{supp} (p s) \Rightarrow \text{additive} (wp (a x)); \\
\forall s. \text{finite} (\text{supp} (p s)) \Rightarrow \text{additive} (wp (\text{SetPC} a p))
\end{align*}
\]
*by* (rule additiveI,
    simp add: wp-eval additiveD distrib-left setsum.distrib)

**Lemma** *additive-wp-Bind:*
\[
\begin{align*}
\forall x. \text{additive} (wp (a (f x))) \Rightarrow \text{additive} (wp (\text{Bind} f a))
\end{align*}
\]
*by* (simp add: wp-eval additive-def)

**Lemma** *additive-wp-Embed:*
\[
\begin{align*}
\text{additive} t \Rightarrow \text{additive} (wp (\text{Embed} t))
\end{align*}
\]
*by* (simp add: wp-eval)

**Lemma** *additive-wp-repeat:*
\[
\begin{align*}
\text{additive} (wp a) \Rightarrow \text{well-def} a \Rightarrow \text{additive} (wp (\text{repeat} n a))
\end{align*}
\]
*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:*
\[
\begin{align*}
\text{maximal} (wp \text{Skip})
\end{align*}
\]
*by* (simp add: maximal-def wp-eval)

**Lemma** *max-wp-Apply:*
\[
\begin{align*}
\text{maximal} (wp (\text{Apply} f))
\end{align*}
\]
*by* (auto simp: wp-eval o-def)

**Lemma** *max-wp-Seq:*
\[
\begin{align*}
\forall a. \text{maximal} (wp a); \text{maximal} (wp b) \Rightarrow \text{maximal} (wp (a ;; b))
\end{align*}
\]
*by* (simp add: wp-eval maximal-def)

**Lemma** *max-wp-PC:*
\[
\begin{align*}
\forall a. \text{maximal} (wp a); \text{maximal} (wp b) \Rightarrow \text{maximal} (wp (a \oplus b))
\end{align*}
\]
*by* (rule maximalI, simp add: maximalD field-simps wp-eval)

**Lemma** *max-wp-DC:*
\[
\begin{align*}
\forall a. \text{maximal} (wp a); \text{maximal} (wp b) \Rightarrow \text{maximal} (wp (a \sqcap b))
\end{align*}
\]
*by* (rule maximalI, simp add: wp-eval maximalD)

**Lemma** *max-wp-SetPC:*
\[
\begin{align*}
\forall a. a \in \text{supp} (P s) \Rightarrow \text{maximal} (wp (p a)); \forall s. (\sum a \in \text{supp} (P s). P s a) =
\end{align*}
\]
\[
1 \implies \maximal (wp (SetPC p P)) \\
\quad \text{by (auto simp:maximalD wp-def SetPC-def setsum-left-distrib[symmetric])}
\]

**lemma** max-wp-SetDC:
\[
\begin{align*}
\text{fixes} & \quad p: 'a \Rightarrow 's \text{ prog} \\
\text{assumes} & \quad mp: \forall s a. a \in S s \implies \maximal (wp (p a)) \\
\text{and ne:} & \quad \forall s. S s \neq \{\} \\
\text{shows} & \quad \maximal (wp (SetDC p S))
\end{align*}
\]
\[
\text{proof} (\text{rule maximalI, rule ext, unfold wp-eval})
\]
\[
\begin{align*}
\text{fix} & \quad c :: \text{real} \quad \text{and} \quad s :: 's \\
\text{assume} & \quad 0 \leq c \\
\text{hence} & \quad \Inf ((\lambda a. wp (p a) (\lambda -. c) s) \cdot S s) = \Inf ((\lambda -. c) \cdot S s) \\
\text{using} & \quad mp \quad \text{by (simp add:maximalD cong:image-cong)} \\
\text{also} & \quad \{ \\
\text{from} & \quad \text{ne have} \quad \exists a. a \in S s \quad \text{by (blast)} \\
\text{hence} & \quad \Inf ((\lambda -. c) \cdot S s) = c \\
\text{by (simp add:image-def)} \\
\} \\
\text{finally show} & \quad \Inf ((\lambda a. wp (p a) (\lambda -. c) s) \cdot S s) = c
\end{align*}
\]
\[\text{qed}\]

**lemma** max-wp-Embed:
\[
\begin{align*}
\maximal t \implies & \quad \maximal (wp (Embed t)) \\
\text{by (simp add:wp-eval)}
\end{align*}
\]

**lemma** max-wp-repeat:
\[
\maximal (wp a) \implies \maximal (wp (repeat n a)) \\
\text{by (induct n, simp-all add:max-wp-Skip max-wp-Seq)}
\]

**lemma** max-wp-Bind:
\[
\begin{align*}
\text{assumes} & \quad ma: \forall s. \maximal (wp (a (f s))) \\
\text{shows} & \quad \maximal (wp (Bind f a))
\end{align*}
\]
\[
\text{proof} (\text{rule maximalI, rule ext, simp add:wp-eval})
\]
\[
\begin{align*}
\text{fix} & \quad c :: \text{real} \quad \text{and} \quad s \\
\text{assume} & \quad 0 \leq c \\
\text{with} & \quad ma \quad \text{have} \quad wp (a (f s)) (\lambda -. c) = (\lambda -. c) \quad \text{by (blast)} \\
\text{thus} & \quad wp (a (f s)) (\lambda -. c) s = c \quad \text{by (auto)}
\end{align*}
\]
\[\text{qed}\]

**lemmas** max-intros =
\[
\begin{align*}
\text{max-wp-Skip} & \quad \text{max-wp-Apply} \\
\text{max-wp-Seq} & \quad \text{max-wp-PC} \\
\text{max-wp-DC} & \quad \text{max-wp-SetDC} \\
\text{max-wp-SetPC} & \quad \text{max-wp-Embed} \\
\text{max-wp-Bind} & \quad \text{max-wp-repeat}
\end{align*}
\]

A healthy transformer that terminates is maximal.

**lemma** healthy-term-max:
assumes \(ht\): healthy \(t\)
and \(trm\): \(\lambda s. 1 \vdash t (\lambda s. 1)\)
shows maximal \(t\)
proof (intro maximalI ext)
fix \(c::\text{real}~\text{and}~s\)
assume \(nnc: 0 \leq c\)

have \(t (\lambda s. c) s = t (\lambda s. 1) s\) by (simp)
also from \(nnc\) \(\text{healthy-scalingD}[\text{OF}~ht]\)

finally show \(t (\lambda s. c) s = c\).
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) \implies determ (wp b) \implies \text{well-def} b \implies determ (wp (a ;; b))
by (intro determI fa-intros max-intros, auto)

lemma det-wp-PC:
determ (wp a) \implies determ (wp b) \implies determ (wp (a P \oplus b))
by (intro determI fa-intros max-intros, auto)

lemma det-wp-SetPC:
\((\forall x. s. x \in \text{supp} (p s)) \implies \text{determ} (wp (a x))) \implies \\
(\forall s. \text{finite} (\text{supp} (p s))) \implies \\
(\forall s. \text{setsum} (p s) (\text{supp} (p s)) = 1) \implies \\
\text{determ} (wp (\text{SetPC} a p))
by (intro determI fa-intros max-intros, auto)

lemma det-wp-Bind:
\((\forall x. \text{determ} (wp (a (f x)))) \implies \text{determ} (wp (\text{Bind} f a))\)
by (intro determI fa-intros max-intros, auto)

lemma det-wp-Embed:
determ \(t\) \implies determ (wp (Embed \(t\)))
by (simp add: wp-eval)

lemma det-wp-repeat:
determ (wp a) ⇒ well-def a ⇒ 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: health-
iness 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 con-
junction is distributed between them 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 inter-
pretations (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)
4.7. WELL-DEFINED PROGRAMS.

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-a-a: wp-under-wlp a
  and wp-a-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) ⊢ wp a (wp b P) by(auto)
also {
  from wp-u-b and uP have wp b P ⊢ wp b P by(blast)
  with h-wlp-a and h-wlp-b and h-wp-b and uP
  have wp a (wp b P) ⊢ wp a (wp b P) by(blast intro:nearly-healthy-monoD[OF h-wlp-a])
}
finally show wp a (wp b P) ⊢ wp a (wp 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-a-a: wp-under-wlp a
  and wp-a-b: wp-under-wlp b
  and uP: unitary P
shows wp-under-wlp (a PC b)
proof(rule wp-under-wlpI, unfold wp-eval, rule le-funI)
fix Q::a ⇒ 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 \( wp\text{-}u\text{-}b \) 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 \( wp\text{-}u\text{-}a \) 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 \( wp\text{-}under\text{-}wlp\text{-}DC \): 

assumes \( wp\text{-}u\text{-}a: \ wp\text{-}under\text{-}wlp\ a \)

and \( wp\text{-}u\text{-}b: \ wp\text{-}under\text{-}wlp\ b \)

shows \( wp\text{-}under\text{-}wlp\ (a \land b) \)

proof (rule wp-under-wlpI, unfold wp-eval, rule le-funI)

fix \( Q:\ 'a \Rightarrow \text{real} \) and \( s \)

assume \( uQ: \text{unitary} Q \)

from \( wp\text{-}u\text{-}a\ uQ \) have \( wp\ a\ Q_s \leq wlp\ a\ Q_s \) by(blast)

moreover

from \( wp\text{-}u\text{-}b\ uQ \) 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 \( wp\text{-}under\text{-}wlp\text{-}SetPC \):

assumes \( wp\text{-}u\text{-}f: \ \bigwedge s\ a. \ a \in supp\ (P_s) \Rightarrow wp\text{-}under\text{-}wlp\ (f\ a) \)

and \( nP:\ \bigwedge s\ a. \ a \in supp\ (P_s) \Rightarrow 0 \leq P_s a \)

shows \( wp\text{-}under\text{-}wlp\ (\text{SetPC}\ f\ P) \)

proof (rule wp-under-wlpI, unfold wp-eval, rule le-funI)

fix \( Q:\ 'a \Rightarrow \text{real} \) and \( s \)

assume \( uQ: \text{unitary} Q \)

from \( wp\text{-}u\text{-}f\ uQ\ nP \)

show \( \bigwedge a\in\text{supp}\ (P_s). \ P_s a * wp\ (f\ a)\ Q_s \leq \bigwedge a\in\text{supp}\ (P_s). \ P_s a * wlp\ (f\ a)\ Q_s \)

by(auto intro!:setsum-mono mult-left-mono)

qed
lemma \text{wp-under-wlp-SetDC}:

\text{assumes wp-u-f: } \forall s, a \in S \implies \text{wp-under-wlp } (f a)

\text{and hf: } \forall s, a \in S \implies \text{healthy } (wp (f a))

\text{and nS: } \forall s. S s \neq \{}

\text{shows } \text{wp-under-wlp } (\text{SetDC } f S)

\text{proof (rule wp-under-wlpI, rule le-funI, unfold wp-eval)}

fix \ Q \colon \text{ real and } s

\text{assume \ uQ: } \text{unitary } Q

\text{show } \text{Inf } ((\lambda a. \text{ wp } (f a) Q s) \ ' S s) \leq \text{Inf } ((\lambda a. \text{ wlp } (f a) Q s) \ ' S s)

\text{proof (rule cInf-mono)}

\text{from nS show } (\lambda a. \text{ wlp } (f a) Q s) \ ' S s \neq \{}

\text{by (blast)}

\text{fix } x \text{ assume } \text{xin: } x \in (\lambda a. \text{ wp } (f a) Q s) \ ' S s

\text{then obtain } a \text{ where } \text{ain: } a \in S s \text{ and } \text{xrw: } x = \text{ wp } (f a) Q s

\text{by (blast)}

\text{with } \text{wp-u-f } uQ

\text{have } \text{wp } (f a) Q s \leq \text{wlp } (f a) Q s \text{ by (blast)}

\text{moreover from } \text{ain have } \text{wp } (f a) Q s \in (\lambda a. \text{ wp } (f a) Q s) \ ' S s

\text{by (blast)}

\text{ultimately show } \exists y \in (\lambda a. \text{ wp } (f a) Q s) \ ' S s. y \leq x

\text{by (auto simp: xrw)}

next

\text{fix } y \text{ assume } \text{yin: } y \in (\lambda a. \text{ wp } (f a) Q s) \ ' S s

\text{then obtain } a \text{ where } \text{ain: } a \in S s \text{ and } \text{yrw: } y = \text{ wp } (f a) Q s

\text{by (blast)}

\text{with hf } aQ \text{ have } \text{unitary } (\text{wp } (f a) Q) \text{ by (auto)}

\text{with yrw show } 0 \leq y \text{ by (auto)}

qed

qed

lemma \text{wp-under-wlp-Embed}:

\text{wp-under-wlp } (\text{Embed } t)

\text{by (rule wp-under-wlpI, unfold wp-eval, blast)}

lemma \text{wp-under-wlp-loop}:

\text{fixes body::'s prog}

\text{assumes hwp: } \text{healthy } (\text{wp body})

\text{and hwlp: } \text{nearly-healthy } (\text{wlp body})

\text{and wp-under: } \text{wp-under-wlp body}

\text{shows wp-under-wlp } (\text{do } G \longrightarrow \text{ body od})

\text{proof (rule wp-under-wlpI)}

\text{fix } P::'s \text{ expect}

\text{assume } uP: \text{ unitary } P \text{ hence } sP: \text{ sound } P \text{ by (auto)}

let \ ?X \ Q s = «G» s * \text{ wp body } Q s + «N G» s * P s

let \ ?Y \ Q s = «G» s * \text{ wlp body } Q s + «N G» s * P s
CHAPTER 4. THE PGCL LANGUAGE

show wp (do G → body od) P ⊢ wlp (do G → body od) P
proof (simp add: hwp hwlwp 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])
    by (auto intro: add-mono mult-left-mono)
}
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-pconj[intro]:
  ⇒ sub-distrib-pconj prog
4.7. WELL-DEFINED PROGRAMS.

unfolding sub-distrib-pconj-def by(simp)

lemma sub-distrib-pconjD[dest]:
\[ P \&\& Q \iff \text{wp prog (P \&\& Q)} \]
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-wlp-b: 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 wp a (wp b P) \&\& wp a (wp b Q) ⇒ wp a (wp b P \&\& wp b Q)
  by(blast intro!:sub-distrib-pconjD[OF sdp-a])
also { from sdp-b and uP and uQ
  have wp 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 (wp 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 wp a (wp 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 \((a \land b)\)

proof\[(\text{rule sub-distrib-pconjI, unfold wp-eval, rule le-funI})\]
fix \(P::'s \Rightarrow \text{real}\) and \(Q::'s \Rightarrow \text{real}\) and \(s::'s\)
assume \(uP: \text{unitary } P\) and \(uQ: \text{unitary } Q\)

\[
\begin{align*}
\text{have } ((\lambda s. \text{min } (\text{wlp } a \ P \ s) \ (\text{wlp } b \ P \ s)) \land \&
& (\lambda s. \text{min } (\text{wp } a \ Q \ s) \ (\text{wp } b \ Q \ s))) \ s \leq \\
& \text{min } (\text{wlp } a \ P \ s \ & \text{wp } a \ Q \ s) \ (\text{wlp } b \ P \ s \ & \text{wp } b \ Q \ s) \\
\text{unfolding exp-conj-def by (rule min-conj)}
\end{align*}
\]
also \{
\[
\begin{align*}
\text{have } (\lambda s. \text{wp } a \ P \ s \ & \text{wp } a \ Q \ s) = \text{wp } a \ P \ & \text{wp } a \ Q \\
& \text{by (simp add: exp-conj-def)}
\end{align*}
\]
also from sdp-a \(uP\)\(\text{ have } ... \vdash \text{wp } a \ (P \ & \text{wp } a \ Q)\) s
by (blast dest: sub-distrib-pconjD)
finally have wp a P s \& wp a Q s \leq wp a (P \ & \text{wp } a \ Q) s
by (rule le-funD)
moreover \{
\[
\begin{align*}
\text{have } (\lambda s. \text{wp } b \ P \ s \ & \text{wp } b \ Q \ s) = \text{wp } b \ P \ & \text{wp } b \ Q \\
& \text{by (simp add: exp-conj-def)}
\end{align*}
\]
also from sdp-b \(uP\)\(\text{ have } ... \vdash \text{wp } b \ (P \ & \text{wp } b \ Q)\) s
by (blast)
finally have wp b P s \& wp b Q s \leq wp b (P \ & \text{wp } b \ Q) s
by (rule le-funD)
\}
ultimately
\[
\begin{align*}
\text{have } \text{min } (\text{wp } a \ P \ s \ & \text{wp } a \ Q \ s) \ (\text{wp } b \ P \ s \ & \text{wp } b \ Q \ s) \leq \\
& \text{min } (\text{wp } a \ (P \ & \text{wp } a \ Q) s \ (\text{wp } b \ (P \ & \text{wp } b \ Q) s \text{ by (auto)}
\end{align*}
\}
finally
show \((\lambda s. \text{min } (\text{wp } a \ P \ s) \ (\text{wp } b \ P \ s)) \land \&
& (\lambda s. \text{min } (\text{wp } a \ Q \ s) \ (\text{wp } b \ Q \ s))) \ s \leq \\
& \text{min } (\text{wp } a \ (P \ & \text{wp } a \ Q) \ s \ (\text{wp } b \ (P \ & \text{wp } b \ 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 \((a \oplus b)\)
proof\[(\text{rule sub-distrib-pconjI, unfold wp-eval, rule le-funI})\]
fix \(Q::'s \Rightarrow \text{real}\) and \(R::'s \Rightarrow \text{real}\) and \(s::'s\)
assume \(uQ: \text{unitary } Q\) and \(uR: \text{unitary } R\)
4.7. WELL-DEFINED PROGRAMS.

have \( nnA \): \( 0 \leq P s \) and \( nnB \): \( 0 \leq 1 - P s \)
using \( uP \) by \((auto \ simp\: sign\: simps)\)

note \( nn = nnA \ nnB \)

have 
\((\lambda s. P s * wp a Q s + (1 - P s) * wp b Q s) \land \land \)
\((\lambda s. P s * wp a R s + (1 - P s) * wp b R s)) s = \)
\(((P s * wp a Q s + (1 - P s) * wp b Q s) + \)
\((P s * wp a R s + (1 - P s) * wp b R s)) \lor 1 \)
by \((simp add\: exp\: conj\: def \ pconj\: def)\)

also have \( ... = P s * \)
\(((wp a Q s + wp a R s) \lor P s) + \)
\(((1 - P s) * (wp b Q s + wp b R s) \lor (1 - P s)) \)
by \((rule \ tminus\: add\: mono)\)

also have \( ... = (P s * (wp a Q s + wp a R s) \lor 1)) + \)
\(((1 - P s) * (wp b Q s + wp b R s) \lor 1)) \)
by \((simp add\: nn \ tminus\: left\: distrib)\)

also have \( ... = P s * \)
\(((wp a Q \land \land wp a R) s) + \)
\(((1 - P s) * ((wp b Q \land \land wp b R) s) \lor (1 - P s) * wp b (Q \land \land R) s) \)
by \((simp add\: exp\: conj\: def \ pconj\: def)\)
also \{ 
from \( sdpa \) \( sdpa \) \( uQ \) \( aR \)
have \( P s * (wp a Q \land \land wp a R) s \leq P s * wp a (Q \land \land R) s \)
and \( (1 - P s) * (wp b Q \land \land wp b R) s \leq (1 - P s) * wp b (Q \land \land R) s \)
by \((auto \ intro\: le\: fun\: D \ sub\: distr\: ib \ pconj\: D \ nn \ mult\: left\: mono)\)

hence \( P s * (wp a (Q \land \land R) s + (1 - P s) * wp b (Q \land \land R) s) \leq \)
\( P s * wp a (Q \land \land R) s + (1 - P s) * wp b (Q \land \land R) s \)
by \((auto)\)
\}
finally show 
\((\lambda s. P s * wp a Q s + (1 - P s) * wp b Q s) \land \land \)
\((\lambda s. P s * wp a R s + (1 - P s) * wp b R s)) s \leq \)
\( P s * wp a (Q \land \land R) s + (1 - P s) * wp b (Q \land \land R) s \).

qed

lemma \( sdpa\: Embed\): 
\([\ A P Q, [\ \text{unitary } P; \ \text{unitary } Q ] \implies t P \land \land t Q \vdash t (P \land \land Q) \] \implies \)
\( sub\: distr\: ib \ pconj \) \((Embed \ t)\)
by \((auto \ simp\: wp\: eval)\)

lemma \( sdpa\: repeat\): 
-fixes \( a::'s \) \( \text{proq} \)
- assumes \( sdpa\: sub\: distr\: ib \ pconj \ a \)
and \textit{hwp}: healthy \((wp \ a)\) and \textit{hwlp}: nearly-healthy \((wp \ a)\)

shows \textit{sub-distrib-pconj} \((\text{repeat } n \ a) \ (\text{is } ?X \ n)\)

\textbf{proof}(\textit{induct } n)

show \( ?X \ 0 \) by \((\text{simp add: sdp-Skip})\)

fix \( n \) assume \( \text{IH}: ?X \ n \)

show \( ?X \ (\text{Suc } n) \)

proof \((\text{rule sub-distrib-pconjI, simp add: wp-eval})\)

fix \( P :: \text{'}a \Rightarrow \text{'}s \) and \( Q :: \text{'}s \Rightarrow \text{real} \)

assume \( uP :: \text{unitary } P \) and \( uQ :: \text{unitary } Q \)

from \( \text{assms} \) have \( \text{hwlp a: nearly-healthy } (wp \ (\text{repeat } n \ a)) \)

and \( \text{hwp a: healthy } (wp \ (\text{repeat } n \ a)) \)

by \((\text{auto intro: healthy-intros})\)

from \( uP \) and \( \text{hwpa have unitary } (wp \ (\text{repeat } n \ a) \ P) \) by \((\text{blast})\)

moreover from \( uQ \) and \( \text{hwpa have unitary } (wp \ (\text{repeat } n \ a) \ Q) \) by \((\text{blast})\)

ultimately

have \( wp \ (wp \ (\text{repeat } n \ a) \ (P \ &\& \ wp \ (\text{repeat } n \ a) \ Q)) \)

using \( \text{sdpa by} (\text{blast}) \)

also \{

from \( \text{hwpa have nearly-healthy } (wp \ (\text{repeat } n \ a)) \) by \((\text{rule healthy-intros})\)

with \( uP \) have \( \text{sound } (wp \ (\text{repeat } n \ a) \ P) \) by \((\text{auto})\)

moreover from \( \text{hwpa have sound } (wp \ (\text{repeat } n \ a) \ Q) \)

by \((\text{auto intro: healthy-intros})\)

ultimately have \( \text{sound } (wp \ (\text{repeat } n \ a) \ P \ &\& \ wp \ (\text{repeat } n \ a) \ Q) \)

by \((\text{rule exp-conj-sound})\)

moreover \{

from \( uP \ uQ \) have \( \text{sound } (P \ &\& \ Q) \) by \((\text{auto intro: exp-conj-sound})\)

with \( \text{hwp have sound } (wp \ (\text{repeat } n \ a) \ (P \ &\& \ Q)) \)

by \((\text{auto intro: healthy-intros})\)

\}

moreover from \( uP \ uQ \) \textit{IH}

have \( wp \ (\text{repeat } n \ a) \ P \ &\& \ wp \ (\text{repeat } n \ a) \ Q \)

by \((\text{blast})\)

ultimately

have \( wp \ (wp \ (\text{repeat } n \ a) \ (P \ &\& \ wp \ (\text{repeat } n \ a) \ Q)) \)

by \((\text{rule mono-transD[OF healthy- monoD, OF hwp]})) \)

\}

finally show \( wp \ (wp \ (\text{repeat } n \ a) \ P \ &\& \ wp \ (\text{repeat } n \ a) \ Q) \)

by \((\text{rule exp-conj-sound})\)

ultimately

have \( wp \ a \ (wp \ (\text{repeat } n \ a) \ (P \ &\& \ wp \ (\text{repeat } n \ a) \ Q)) \)

by \((\text{rule mono-transD[OF healthy- monoD, OF hwp]})) \)

\}

\textbf{qed}

\textbf{lemma} \( \text{sdp-SetPC}: \)

\textbf{fixes} \( p : \ 'a \Rightarrow \ 's \text{ prog} \)

\textbf{assumes} \( \text{sdp: } \forall s \ a. \ a \in \text{ supp } (P \ s) \implies \text{ sub-distrib-pconj } (p \ a) \)

and \( \text{fin: } \forall s. \ \text{finite } (\text{ supp } (P \ s)) \)

and \( \text{nnp: } \forall s \ a. \ 0 \leq P \ s \ a \)

and \( \text{sub: } \forall s. \ \text{setsum } (P \ s) \ (\text{ supp } (P \ s)) \leq 1 \)
4.7. WELL-DEFINED PROGRAMS.

shows sub-distrib-pconj (SetPC p P)
proof (rule sub-distrib-pconj1, 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 \((\lambda s. \sum a:\in\text{supp} (P s). P s a * wp (p a) Q s) \&\&
            (\lambda s. \sum a:\in\text{supp} (P s). P s a * wp (p a) R s)) s =
            \((\sum a:\in\text{supp} (P s). P s a * wp (p a) Q s) + \sum a:\in\text{supp} (P s). P s a * wp (p a) R s) \oplus 1\)
    by (simp add: exp-conj-def pconj-def)
  also have \(... = (\sum a:\in\text{supp} (P s). P s a * (wp (p a) Q s + wp (p a) R s)) \oplus 1\)
    by (simp add: setsum.distrib field-simps)
  also from sub
  have \(...) ≤ (\sum a:\in\text{supp} (P s). P s a * (wp (p a) Q s + wp (p a) R s)) \oplus P s a
    by (rule tminus-right-antimono)
  also from fin
  have \(...) ≤ (\sum a:\in\text{supp} (P s). P s a * (wp (p a) Q s + wp (p a) R s) \oplus P s a)
    by (rule tminus-setsum-mono)
  also from nmp
  have \(...) = (\sum a:\in\text{supp} (P s). P s a * (wp (p a) Q s + wp (p a) R s \oplus 1))
    by (simp add: tminus-left-distrib)
  also have \(...) = (\sum a:\in\text{supp} (P s). P s a * (wp (p a) Q \&\& wp (p a) R) s)
    by (simp add: pconj-def exp-conj-def)
  also {\}
    from sdp uQ uR
    have \(\lambda a. a \in\text{supp} (P s) \Longrightarrow wp (p a) Q \&\& wp (p a) R \iff wp (p a) (Q \&\& R)\)
      by (blast intro: sub-distrib-pconjD)
    with nmp
    have \((\sum a:\in\text{supp} (P s). P s a * (wp (p a) Q \&\& wp (p a) R) s) \leq
            (\sum a:\in\text{supp} (P s). P s a * (wp (p a) (Q \&\& R)) s)\)
      by (blast intro: setsum-mono mult-left-mono)
  } finally show \((\lambda s. \sum a:\in\text{supp} (P s). P s a * wp (p a) Q s) \&\&
              (\lambda s. \sum a:\in\text{supp} (P s). P s a * wp (p a) R s)) s ≤
              \((\sum a:\in\text{supp} (P s). P s a * wp (p a) (Q \&\& R) s)\).
qed

lemma sdp-SetDC:
  fixes p::'a ⇒ 's prog
  assumes sdp: \(\forall s. a \in S s \Longrightarrow \text{sub-distrib-pconj} (p a)\)
    and hwp: \(\forall s. a \in S s \Longrightarrow \text{healthy} (wp (p a))\)
    and hwlp: \(\forall s. a \in S s \Longrightarrow \text{nearly-healthy} (wp (p a))\)
    and nc: \(\forall s. S s \neq \{\}\)
  shows sub-distrib-pconj (SetDC p S)
proof (rule sub-distrib-pconj1, rule le-funI)
  fix P::'s ⇒ real and Q::'s ⇒ real and s::'s
  assume uP: unitary P and uQ: unitary Q
from \textit{uP hwlp}

have \(\forall x. \ x \in (\lambda a. \ \text{wlp} (p \ a) \ P) \implies \text{unitary} \ x\) 
by(auto)

hence \(\forall y. \ y \in (\lambda a. \ \text{wlp} (p \ a) \ P) \implies 0 \leq y\) 
by(auto)

hence \(\forall a. \ a \in S \implies \text{wlp} (\text{SetDC} p S) \ P \ s \leq \text{wlp} (p \ a) \ P \ s\)

unfolding \text{wp-eval} by(intro \text{cInf-lower bdd-below1, auto})

moreover {
from \textit{uQ hwp} have \(\forall a. \ a \in S \implies 0 \leq wp (p \ a) \ Q \ s\) 
by(blast)

hence \(\forall a. \ a \in S \implies wp (\text{SetDC} p S) \ Q \ s \leq wp (p \ a) \ Q \ s\)

unfolding \text{wp-eval} by(intro \text{cInf-lower bdd-below1, auto})
}

ultimately

have \(\forall a. \ a \in S \implies wp (\text{SetDC} p S) \ P \ s \ + \ wp (\text{SetDC} p S) \ Q \ s \leq I\)

by(auto intro:tminus-left-mono add-mono)

also have \(\forall a. \ wp (p \ a) \ P \ s \ + \ wp (p \ a) \ Q \ s \ = \ (wp (p \ a) \ P \ \&\& wp (p \ a) \ Q) \ s\)

by(simp add:exp-conj-def pconj-def)

also from \textit{sdp uP uQ}

have \(\forall a. \ a \in S \implies ... \ a \leq wp (p \ a) (P \ \&\& Q) \ s\)

by(blast)

also have \(\forall a. \ ... \ a \ = wp (p \ a) (\lambda s. \ P \ s \ + \ Q \ s \leq I) \ s\)

by(simp add:exp-conj-def pconj-def)

finally

show \(wp (\text{SetDC} p S) \ P \ \&\& wp (\text{SetDC} p S) \ Q \ s \leq wp (\text{SetDC} p S) (P \ \&\& Q) \ s\)

unfolding exp-conj-def pconj-def wp-eval

using ne by(blast intro!:cInf-greatest)

qed

\textbf{lemma} \textit{sdp-Bind:}

\(\exists s. \ \text{sub-distrib-pconj} (p \ (f \ s)) \implies \text{sub-distrib-pconj} (\text{Bind} f \ p)\)

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.

\textbf{lemma} \textit{sdp-loop:}

\textbf{fixes} \textit{body:’s prog}

\textbf{assumes} \textit{sdp-body: sub-distrib-pconj body}

and \textit{hwlp: nearly-healthy (wp body)}

and \textit{hwp: healthy (wp body)}

\textbf{shows} \textit{sub-distrib-pconj (do G \ → \ body od)}

\textbf{proof}(\textit{rule sub-distrib-pconj1, rule loop-induct(OF hwp hwlp)})

\textbf{fix} \textit{P Q::’s expect and S::’s trans × ’s trans \ set}

\textbf{assume} \textit{uP: unitary P and uQ: unitary Q}

and \textit{fstl: \ \forall x \in S. \ feasible (fst x)}

and \textit{usnd: \ \forall x \in S. \ unitary Q \ → \ unitary (snd x Q)}

and \textit{IH: \ \forall x \in S. \ snd x P \ \&\& \ \exists x Q \ F \ \exists x (P \ \&\& Q)
4.7. WELL-DEFINED PROGRAMS.

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 \( \forall s. 0 ≤ t (P & & Q) s \) 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 \( \forall s. t (P & & Q) s ≤ Sup-trans (fst ' S) (P & & Q) s \)
by(auto intro:exp-conj-unitary)
}
finally have nn-rhs: \( \forall s. 0 ≤ Sup-trans (fst ' S) (P & & Q) s \).

have \( \forall R. Inf-utrans (snd ' S) P & & R ⊢ Sup-trans (fst ' S) (P & & Q) \implies R ≤ ?f \)
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 \neg (\text{Inf-utrans} \cdot S) \: P \land R \: s \leq \text{Sup-trans} \cdot S \: P \land Q \: s

by(simp add:exp-conj-def)

thus \neg (\text{Inf-utrans} \cdot S) \: P \land R \: \vdash \text{Sup-trans} \cdot S \: P \land Q

by(auto)

qed

moreover have \forall \ t \in \text{fst} \cdot S. \: \text{Inf-utrans} \cdot S \: P \land Q \: \vdash \text{Sup-trans} \cdot S \: P \land Q

proof

fix \ t assume \ tin: \ t \in \text{fst} \cdot S
then obtain \ x where \ xin: \ x \in S \ and \ fx: \ t = \text{fst} \ x\ by(auto)

from \ xin have \text{snd} \ x \in \text{snd} \cdot S\ by(auto)
with \ uP usnd have \text{Inf-utrans} \cdot S \: P \vdash \text{snd} \ x \: P

by(auto intro:triple-utos-transD[OF \text{Inf-utrans-lower}])

hence \text{Inf-utrans} \cdot S \: P \land \text{fst} \ x \: Q \vdash \text{snd} \ x \: P \land \text{fst} \ x \: Q

by(auto intro:entails-frame)

also from \ xin \text{IH} have ... \vdash \text{fst} \ x \: (P \land Q)

by(auto)

also from \ xin \text{fffst} \exp-conj-unitary[\text{OF} \ uP \ uQ]

have ... \vdash \text{Sup-trans} \cdot S \: P \land Q

by(auto intro:triple-utos-transD[OF \text{Sup-trans-upper}])

finally show \text{Inf-utrans} \cdot S \: P \land Q \: \vdash \text{Sup-trans} \cdot S \: P \land Q

by(simp add:fx)

qed

ultimately have \bt: \ \forall \ t \in \text{fst} \cdot S. \: t \: Q \vdash \ ?f\ by(blast)

have \text{Sup-trans} \cdot S \: Q = \text{Sup-exp} \ \{ t \: Q \mid \ t \in \text{fst} \cdot S \}

by(simp add:Sup-trans-def)

also have ... \vdash \ ?f

proof(rule \text{Sup-exp-least})

from \ bt show \ \forall \ R \in \{ t \: Q \mid \ t \in \text{fst} \cdot S \}. \: R \vdash \ ?f\ by(blast)

from \ ne obtain \ t where \ tin: \ t \in \text{fst} \cdot S\ by(auto)

with \ \text{fffst} \ uQ have \text{unitary} \cdot (t \: Q)\ by(auto)

hence \ \lambda s. \: 0 \vdash \: t \: Q\ by(auto)

also from \ tin \bt have ... \vdash \ ?f\ by(auto)

finally show \text{nneg} \cdot (\lambda s. \: 1 + \text{Sup-trans} \cdot S \: (P \land Q)\: s - \text{Inf-utrans} \cdot S \: P \: s)

by(auto)

qed

finally have \text{Inf-utrans} \cdot S \: P \land \text{Sup-trans} \cdot S \: P \land \ ?f

by(auto intro:entails-frame)

also from \ \text{nn-rhs} have ... \vdash \text{Sup-trans} \cdot S \: (P \land Q)

by(simp add:exp-conj-def pconj-def)

finally show \ ?\text{thesis}.

qed

next
4.7. WELL-DEFINED PROGRAMS.

fix $P$:’s expect and $t$:’s trans
assume $uP$: unitary $P$ and $uQ$: unitary $Q$

and $ft$: feasible $t$
and $uw$: unitary $Q \implies$ unitary ($uQ$)
and $IH$: $uP \land Q \vdash t$ ($P \land Q$)

show $WLP$ ($body :: Embed u \ast \ G \ast \!!$ $Skip$) $P \land Q$

wp ($body :: Embed t \ast \ G \ast \!!$ $Skip$) $Q \vdash$
wp ($body :: Embed t \ast \ G \ast \!!$ $Skip$) ($P \land Q$)

proof (rule le-fun1, simp add: wp-eval exp-conj-def pconj-def)

fix $s$’s

have $\langle G \rangle s \ast \! wp body (uP) s + (1 - \langle G \rangle s) \ast P s +$
\begin{align*}
& (\langle G \rangle s \ast \! wp body (tQ) s + (1 - \langle G \rangle s) \ast Q s) \circ i = \\
& (1 - \langle G \rangle s) \ast P s + (1 - \langle G \rangle s) \ast Q s \circ (1 - \langle G \rangle s))
\end{align*}
by (simp add: ac-simps)

also have $\ldots \leq$
\begin{align*}
& (\langle G \rangle s \ast \! wp body (uP) s + \langle G \rangle s \ast \! wp body (tQ) s \circ \langle G \rangle s) + \\
& (1 - \langle G \rangle s) \ast P s + (1 - \langle G \rangle s) \ast Q s \circ (1 - \langle G \rangle s))
\end{align*}
by (rule tminus-add-mono)

also have $\ldots =$
\begin{align*}
& \langle G \rangle s \ast (wp body (uP) s + \! wp body (tQ) s \circ i) + \\
& (1 - \langle G \rangle s) \ast (P s + Q s \circ i)
\end{align*}
by (simp add: tminus-left-distrib distrib-left)

also \{
from $uP \land uQ \land ft \land uu$

have $\! wp body (uP) \land Q \land \! wp body (tQ) \vdash \! wp body (uP) \land Q \land tQ$
by (auto intro: sub-distrib-pconjD[OF wp-body])

also from $IH$ unitary-sound[OF $uP$] unitary-sound[OF $uQ$] $ft$
unitary-sound[OF uu[OF $uP$]]

have $\ldots \leq \! wp body (t(P \land Q))$
by (blast intro: mono-transD[OF healthy-monoD, OF $uP$] exp-conj-sound)

finally have $\! wp body (uP) s + \! wp body (tQ) s \circ i \leq$
\begin{align*}
& \! wp body (t(\lambda s. P s + Q s \circ i)) s
\end{align*}
by (auto simp: exp-conj-def pconj-def)

hence $\langle G \rangle s \ast (wp body (uP) s + \! wp body (tQ) s \circ i) +$
\begin{align*}
& (1 - \langle G \rangle s) \ast (P s + Q s \circ i) \leq \\
& \langle G \rangle s \ast \! wp body (t(\lambda s. P s + Q s \circ i)) s + \\
& (1 - \langle G \rangle s) \ast (P s + Q s \circ i)
\end{align*}
by (auto intro: add-right-mono mult-left-mono)
\}

finally

show $\langle G \rangle s \ast \! wp body (uP) s + (1 - \langle G \rangle s) \ast P s +$
\begin{align*}
& (\langle G \rangle s \ast \! wp body (tQ) s + (1 - \langle G \rangle s) \ast Q s) \circ i \leq \\
& \langle G \rangle s \ast \! wp body (t(\lambda s. P s + Q s \circ i)) s + \\
& (1 - \langle G \rangle s) \ast (P s + Q s \circ i)
\end{align*}
qed

next

fix $P$:’s expect and $t'$ $u$:’s trans
assume unitary $P$ unitary $Q$
equiv-trans \ t \ t' \ equiv-utrans \ u \ u'
\ u \ P \land \ t \ Q \vdash \ t' (P \land \land \ Q)
thus \ u' \ P \land \ t' \ Q \vdash \ t' (P \land \land \ 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 \Rightarrow \ bool
where
well-def prog \equiv \ healthy (wp prog) \land \ nearly-healthy (wlp prog)
\land \ wp-under-wlp prog \land \ sub-distrib-pconj prog
\land \ sublinear (wp prog) \land \ 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) ] \Rightarrow
well-def prog
unfolding well-def-def by(simp)

lemma well-def-wp-healthy[dest]:
well-def prog \Rightarrow \ healthy (wp prog)
unfolding well-def-def by(simp)

lemma well-def-wlp-nearly-healthy[dest]:
well-def prog \Rightarrow \ nearly-healthy (wlp prog)
unfolding well-def-def by(simp)

lemma well-def-wp-under[dest]:
well-def prog \Rightarrow \ wp-under-wlp prog
unfolding well-def-def by(simp)

lemma well-def-sdp[dest]:
well-def prog \Rightarrow \ sub-distrib-pconj prog
unfolding well-def-def by(simp)

lemma well-def-wp-sublinear[dest]:
well-def prog \Rightarrow \ sublinear (wp prog)
unfolding well-def-def by(simp)

lemma well-def-wp-cts[dest]:
4.7. WELL-DEFINED PROGRAMS.

well-def prog \implies 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

**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 ] ] \implies 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 ] ] \implies well-def (a \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 ] ] \implies well-def (a \sqcap 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:
[ [ \forall x \in S s \implies well-def (a x); \forall s. S s \neq {}];
\forall s. finite (S s) ] \implies well-def (SetDC a S)
by (iprover intro: well-defI healthy-wp-SetDC nearly-healthy-wlp-SetDC
    wp-under-wlp-SetDC sdp-SetDC sublinear-wp-SetDC cts-wp-SetDC)
informal

nonempty-witness
dest:wd-dests)

**lemma** wd-SetPC:
\[
\{ \forall x. x \in (\text{supp} (p \ s)) \implies \text{well-def} (a \ x) ; \ \forall s. \text{unitary} (p \ s) ; \ \forall s. \text{finite} (\text{supp} (p \ s)) \;: \; \forall s. \text{setsum} (p \ s) (\text{supp} (p \ s)) \leq 1 \} \implies \text{well-def} (\text{SetPC} a \ p)
\]
by (iprover intro: well-defI healthy-wp-SetPC nearly-healthy-wlp-SetPC
\quad wp-under-wlp-SetPC sdp-SetPC sublinear-wp-SetPC cts-wp-SetPC
dest:wd-dests unitary-sound sound-nneg)

**lemma** wd-Embed:
fixes \( t : s \text{ trans} \)
assumes \( \text{ht}: \text{healthy} \ t \text{ and st}: \text{sublinear} \ t \text{ and ct}: \text{bd-cts} \ t \)
shows \( \text{well-def} (\text{Embed} \ t) \)
proof (intro well-defI)
\quad from \( \text{ht} \) show \( \text{healthy} \ (\text{wp} (\text{Embed} \ t)) \text{ nearly-healthy} \ (\text{wlp} (\text{Embed} \ t)) \)
\quad by (simp add: wp-def wp-def Embed-def healthy-nearly-healthy)+
\quad from \( \text{st} \) show \( \text{sublinear} \ (\text{wp} (\text{Embed} \ t)) \)
\quad by (simp add: wp-def wp-def Embed-def)
\quad show \( \text{wp-under-wlp} \ (\text{Embed} \ t) \)
\quad by (simp add: wp-under-wlp-def wp-evl)
\quad show \( \text{sub-distrib-pconj} (\text{Embed} \ t) \)
\quad by (rule sub-distrib-pconjI, auto intro: le-funI [OF sublinearD [OF st, where a=1 and b=1 and c=1, simplified]])
\quad simp: exp-conj-def pconj-def wp-def wp-def Embed-def)
\quad from \( \text{ct} \) show \( \text{bd-cts} \ (\text{wp} (\text{Embed} \ t)) \)
\quad by (simp add: wp-def Embed-def)
qed

**lemma** wd-repeat:
well-def \( a \implies \text{well-def} \ (\text{repeat} \ n \ a) \)
by (blast intro: healthy-wp-repeat nearly-healthy-wlp-repeat
\quad wp-under-wlp-repeat sdp-repeat sublinear-wp-repeat cts-wp-repeat)

**lemma** wd-Bind:
\[
\{ \forall s. \text{well-def} (a (f s)) \} \implies \text{well-def} (\text{Bind} f a)
\]
by (blast intro: healthy-wp-Bind nearly-healthy-wlp-Bind
\quad wp-under-wlp-Bind sdp-Bind sublinear-wp-Bind cts-wp-Bind)

**lemma** wd-loop:
well-def \( \text{body} \implies \text{well-def} \ (\text{do} \ G \rightarrow \text{body} \ od) \)
by (blast intro: healthy-wp-loop nearly-healthy-wlp-loop
\quad wp-under-wlp-loop sdp-loop sublinear-wp-loop cts-wp-loop)

**lemmas** wd-intros =
wd-Abort \( \text{wd-Skip} \text{ wd-Apply} \)
wd-Embed \( \text{wd-Seq} \text{ wd-PC} \)
wd-DC \( \text{wd-SetPC} \text{ wd-SetDC} \)
wd-Bind \( \text{wd-repeat} \text{ wd-loop} \)
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**

\[ wp-inv :: (\forall s. 's \Rightarrow s \Rightarrow 's prog \Rightarrow ('s \Rightarrow real) \Rightarrow bool) \]

**where**

\[ wp-inv G body I \iff (\forall s. 'G' s \Rightarrow 'I' s \leq wp body I s) \]

**lemma** *wp-invI*:

\[ \forall I. (\forall s. 'G' s \Rightarrow 'I' s \leq wp body I s) \implies wp-inv G body I \]

**by** (*simp add*: *wp-inv-def*)

**definition**

\[ wlp-inv :: (\forall s. 's \Rightarrow bool) \Rightarrow 's prog \Rightarrow ('s \Rightarrow real) \Rightarrow bool \]

**where**

\[ wlp-inv G body I \iff (\forall s. 'G' s \Rightarrow 'I' s \leq wlp body I s) \]

**lemma** *wlp-invI*:

\[ \forall I. (\forall s. 'G' s \Rightarrow 'I' s \leq wlp body I s) \implies wlp-inv G body I \]

**by** (*simp add*: *wlp-inv-def*)

**lemma** *wlp-invD*:

\[ wlp-inv G body I = \implies 'G' s \Rightarrow 'I' s \leq wlp body I s \]

**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*)

\[ fix s show ('G' \&\& 'I') s \leq wp body 'I' s \]

**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 \leq wp body «I» s

by (simp add: wp-inv-def)

finally show ?thesis.

qed

4.8.2 Partial Correctness


lemma wlp-Loop:

assumes wd: well-def body
and uI: unitary I
and inv: wlp-inv G body I

shows I \leq wp do G \rightarrow body od (\lambda s. «N G» s \* I s)
(is I \leq wp do G \rightarrow body od ?P)

proof

let ?f Q s = «G» s \* wlp body Q s + «N G» s \* ?P s

have I \sqsubseteq \sqsubseteq gfp-exp ?f

proof (rule gfp-exp-upperbound [OF - uI])

have I = (\lambda s. («G» s + «N G» s) \* I s) by (simp add: negate-embed)

also have ... = (\lambda s. «G» s \* I s + «N G» s \* I s)

by (simp add: algebra-simps)

also have ... = (\lambda s. «G» s \* («G» s \* I s) + «N G» s \* («N G» s \* I s))

by (simp add: embed-bool-idem algebra-simps)

also have ... \sqsubseteq (\lambda s. «G» s \* wlp body I s + «N G» s \* ?P s)

using inv by (auto dest: wlp-invD intro: add-mono mult-left-mono)

finally show I \sqsubseteq (\lambda s. «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 ... = wp do G \rightarrow 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: wp-inv G body I
and unit: unitary I
shows \( I \land \land \wp (do G \rightarrow body od) (\lambda s. 1) \vdash wp (do G \rightarrow body od) (\lambda s. \langle N G \rangle s * 1 s) \)

(is \( I \land \land ?T \vdash wp ?loop ?X \))

proof –

We first appeal to the liberal loop rule:

from assms have \( I \land \land ?T \vdash wp ?loop ?X \land \land ?T \)

by(blast intro:exp-conj-mono-left wp-Loop)

Next, by sub-conjunctivity:

also {
  from wd have sdp-loop: sub-distrib-pconj (do G \rightarrow body od)
  by(blast intro:sdp-intros)

  from wd unit have wp ?loop ?X \land \land ?T \vdash wp ?loop (?X \land \land (\lambda s. 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 \rightarrow body od) P =
  (\lambda s. \langle N G \rangle s * P s + \langle G \rangle s * wp body (wp (do G \rightarrow body od) P) s)

unfolding wp-eval

proof –

let \(?X t = wp (body ;; Embed t \triangleq G \oplus Skip)\)

have equiv-trans (lfp-trans \(?X\))
  (wp (body ;; Embed (lfp-trans \(?X\)) \triangleq G \oplus Skip))

proof(intro lfp-trans-unfold)

fix t::'s trans and P::'s expect

assume st: \( Q. \) sound \( Q \implies sound \) \((t Q)\)
  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 \( P \implies sound \) \((t P)\))
  (\( P. \) sound \( P \implies sound \) \((u P)\))

with h show le-trans (wp (body ;; Embed t \triangleq G \oplus Skip))
  (wp (body ;; Embed u \triangleq G \oplus Skip))

by(prover intro:wp-loop-step-mono)
CHAPTER 4. THE PGCL LANGUAGE

next
let \( ?v = \lambda P \, s. \text{bound-of} \, P \)
from h show le-trans (wp (body ;; Embed \(?v \circ G \oplus \text{Skip}\)) \(?v\)
  by (intro le-transI, simp add: wp-eval lfp-loop-fp [unfolded negate-embed])
fix \( P :: \text{’s expect} \)
assume sound \( P \) thus sound \( (?v \, P) \) by (auto)
qed
also have equiv-trans ...
  \((\lambda \, P \, s. \, \langle N \, G > \, s \, * \, P \, s + \langle G > \, s \, * \, wp \, body \, (wp \, (Embed \, (\text{lfp-trans \, ?X}) \, P) \, s))\)
by (rule equiv-transI, simp add: wp-eval algebra-simps negate-embed)
finally show lfp-trans \(?X \, P\) =
  \((\lambda \, s. \, \langle N \, G > \, s \, * \, P \, s + \langle G > \, s \, * \, wp \, body \, (\text{lfp-trans \, ?X \, P}) \, s)\)
using \( sP \) unfolding wp-eval by (blast)
qed

lemma wp-loop-nguard:
[ \[ \text{healthy \, (wp \, body)}; \, \text{sound \, P}; \, \neg \, G \, s \] \implies wp \, do \, G \rightarrow body \, od \, P \, s = P \, s \]
by (subst wp-loop-unfold, simp-all)

lemma wp-loop-guard:
[ \[ \text{healthy \, (wp \, body)}; \, \text{sound \, P}; \, G \, s \] \implies wp \, do \, G \rightarrow body \, od \, P \, s = wp \, (body ;; \, do \, G \rightarrow 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 \( \sqcap \) and \( \sqcup \) as the meet and join operators, respectively. We also take advantage of the algebraic structure to establish a framework 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 :: \( \text{’s prog} \Rightarrow \text{’s prog} \Rightarrow \text{bool} \) (infix \( \sqsubseteq \, 70 \))
where
\( \text{prog} \sqsubseteq \text{prog’} \equiv \forall \, P. \, \text{sound} \, P \rightarrow wp \, \text{prog} \, P \sqsupseteq wp \, \text{prog’} \, P \)

lemma refinesI[intro]:
4.9. THE ALGEBRA OF PGCL

\[ \bigwedge P. \text{sound } P \implies \wp \text{prog } P \vdash \wp \text{prog}^\prime P \implies \text{prog} \sqsubseteq \text{prog}^\prime \]

**unfolding** refines-def by(simp)

**lemma** refinesD[dest]:
\[ \big[ \text{prog} \sqsubseteq \text{prog}^\prime; \text{sound } P \big] \implies \wp \text{prog } P \vdash \wp \text{prog}^\prime P \]

**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**

\[ \text{pequiv} :: \text{`prog } \Rightarrow \text{`prog } \Rightarrow \text{bool} \text{ (infix } \simeq 70) \]

**where**

\[ \text{prog} \simeq \text{prog}^\prime = \forall P. \text{sound } P \implies \wp \text{prog } P = \wp \text{prog}^\prime P \]

**lemma** pequivI[intro]:
\[ \big[ \bigwedge P. \text{sound } P \implies \wp \text{prog } P = \wp \text{prog}^\prime P \big] \implies \text{prog} \simeq \text{prog}^\prime \]

**unfolding** pequiv-def by(simp)

**lemma** pequivD[dest,simp]:
\[ \big[ \text{prog} \simeq \text{prog}^\prime; \text{sound } P \big] \implies \wp \text{prog } P = \wp \text{prog}^\prime P \]

**unfolding** pequiv-def by(simp)

**lemma** pequiv-equiv-trans:
\[ a \simeq b \iff \text{equiv-trans } (\wp a) (\wp b) \]

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 separately

**lemma** DC-comm[ac-simps]:
\[ a \bigcap b = b \bigcap a \]

**unfolding** DC-def by(simp add:ac-simps)

**lemma** DC-assoc[ac-simps]:
\[ a \bigcap (b \bigcap c) = (a \bigcap b) \bigcap c \]

**unfolding** DC-def by(simp add:ac-simps)

**lemma** DC-idem:
\[ a \bigcap a = a \]

**unfolding** DC-def by(simp)

**lemma** AC-comm[ac-simps]:
\[ a \bigcup b = b \bigcup a \]
unfolding AC-def by(simp add:ac-simps)

lemma AC-assoc[ac-simps]:
  \((a \biguplus (b \biguplus c)) = (a \biguplus b) \biguplus c\)
unfolding AC-def by(simp add:ac-simps)

lemma AC-idem:
  \(a \biguplus a = a\)
unfolding AC-def by(simp)

lemma PC-quasi-comm:
  \(a \ominus p \oplus b = b (\lambda s. t - p s) \ominus a\)
unfolding PC-def by(simp add:algebra-simps)

lemma PC-idem:
  \(a \ominus a = a\)
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 \Rightarrow \text{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:
  \((a \bigcap b) \sqsubseteq a\)
by(auto intro!:refinesI simp:wp-eval)

lemma right-refines-DC:
  \((a \bigcap b) \sqsubseteq b\)
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 \bigcap 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 \bigcap c) P
    by(auto simp:wp-eval intro:min.boundedI)
qed

lemma DC-mono:
4.9. THE ALGEBRA OF PGCL

fixes $a$: prog
assumes $r a b$: $a \sqsubseteq b$ and $r c d$: $c \sqsubseteq d$
shows $(a \sqbigcup c) \sqsubseteq (b \sqbigcup d)$

proof (rule refinesI, unfold wp-eval, rule le-funI)
  fix $P$: real and $s$:
assume $s P$: sound $P$
with assms have $\forall s. \operatorname{wp} a P s \leq \operatorname{wp} b P s$ and $\forall s. \operatorname{wp} c P s \leq \operatorname{wp} d P s$
  by (auto)
thus $\min (\operatorname{wp} a P s) (\operatorname{wp} c P s) \leq \min (\operatorname{wp} b P s) (\operatorname{wp} d P s)$
  by (auto)
qed

Laws relating angelic choice and refinement

lemma left-refines-AC:
  $a \sqsubseteq (a \sqbigcup b)$
by (auto intro!: refinesI simp: wp-eval)

lemma right-refines-AC:
  $b \sqsubseteq (a \sqbigcup b)$
by (auto intro!: refinesI simp: wp-eval)

lemma AC-refines:
  fixes $a$: prog and $b$ and $c$
assumes $r a c$: $a \sqsubseteq c$ and $r b c$: $b \sqsubseteq c$
shows $(a \sqbigcup b) \sqsubseteq c$

proof
  fix $P$: real assume $s P$: sound $P$
  with assms have $\forall s. \operatorname{wp} a P s \leq \operatorname{wp} c P s$
    and $\forall s. \operatorname{wp} b P s \leq \operatorname{wp} c P s$
    by (auto dest: refinesD)
  thus $\forall s. \operatorname{wp} (a \sqbigcup b) P \Rightarrow \operatorname{wp} c P$
    unfolding wp-eval by (auto)
qed

lemma AC-mono:
  fixes $a$: prog
assumes $r a b$: $a \sqsubseteq b$ and $r c d$: $c \sqsubseteq d$
shows $(a \sqbigcup c) \sqsubseteq (b \sqbigcup d)$

proof (rule refinesI, unfold wp-eval, rule le-funI)
  fix $P$: real and $s$:
assume $s P$: sound $P$
with assms have $\forall s. \operatorname{wp} a P s \leq \operatorname{wp} b P s$ and $\forall s. \operatorname{wp} c P s \leq \operatorname{wp} d P s$
  by (auto)
thus $\max (\operatorname{wp} a P s) (\operatorname{wp} c P s) \leq \max (\operatorname{wp} b P s) (\operatorname{wp} d P s)$
  by (auto)
qed
Laws depending on the arithmetic of \( a \oplus b \) and \( a \sqcap b \) together

**Lemma DC-refines-PC:**
assumes unit: unitary \( p \)
shows \((a \sqcap b) \subseteq (a \oplus b)\)
proof(rule refinesI, unfold wp-eval, rule le-funI)
fix \( s \) and \( P::'a \Rightarrow real \) assume sound: sound \( P \)
from unit have nn-p: \( 0 \leq p \) s by(blast)
from unit have \( p \) s \( \leq 1 \) by(blast)
hence nn-np: \( 0 \leq 1 - p \) s by(simp)
show \( \min \ (wp \ a \ P \ s) \ (wp \ b \ P \ s) \leq p \) s \( \) wp a P s + \( (1 - p) \) s wp b P s
proof(cases wp a P s \( \leq wp \ b \ P \ s) \)
  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 \( \leq (1 - p) \) s wp b P s
      by(rule mult-left-mono)
    hence \( p \) s \( \) wp a P s + \( (1 - p) \) s wp a P s \( \leq \)
      \( p \) s \( \) wp a P s + \( (1 - p) \) s wp b P s
      by(rule add-left-mono)
  }
finally show \( wp \ a \ P \ s \leq p \) s \( \) wp a P s + \( (1 - p) \) s wp b P s .
next
case False
then have \( le: wp \ b \ P \ s \leq wp \ a \ P \ s \) 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 \( \leq p \) s wp a P s
    by(rule mult-left-mono)
  hence \( p \) s \( \) wp b P s + \( (1 - p) \) s wp b P s \( \leq \)
    \( p \) s \( \) wp a P s + \( (1 - p) \) s wp b P s
    by(rule add-right-mono)
  }
finally show \( wp \ b \ P \ s \leq p \) s \( \) wp a P s + \( (1 - p) \) s wp b P s .
qed
qed

Laws depending on the arithmetic of \( a \oplus b \) and \( a \sqcup b \) together

**Lemma PC-refines-AC:**
assumes unit: unitary \( p \)
shows \((a \oplus b) \subseteq (a \sqcup b)\)
proof(rule refinesI, unfold wp-eval, rule le-funI)
fix \( s \) and \( P::'a \Rightarrow real \) assume sound: sound \( P \)
from unit have \( p s \leq 1 \) by (blast)
hence \( \text{nn-np}: 0 \leq 1 - p s \) by (simp)

show \( p s \ast \text{wp a P s} + (1 - p s) \ast \text{wp b P s} \leq \max (\text{wp a P s}) (\text{wp b P s}) \)
proof (cases \( \text{wp a P s} \leq \text{wp b P s} \))
case True note leab = this
with unit nn-np have \( p s \ast \text{wp a P s} + (1 - p s) \ast \text{wp b P s} \leq p s \ast \text{wp b P s} + (1 - p s) \ast \text{wp b P s} \)
by (auto intro: add-mono mult-left-mono)
also have \( \ldots = \text{wp b P s} \)
by (auto simp: field-simps)
also from leab have \( \ldots = \max (\text{wp a P s}) (\text{wp b P s}) \)
by (auto)
finally show \( \text{thesis} \).
next
case False note leba = this
with unit nn-np have \( p s \ast \text{wp a P s} + (1 - p s) \ast \text{wp b P s} \leq p s \ast \text{wp a P s} + (1 - p s) \ast \text{wp a P s} \)
by (auto intro: add-mono mult-left-mono)
also have \( \ldots = \text{wp a P s} \)
by (auto simp: field-simps)
also from leba have \( \ldots = \max (\text{wp a P s}) (\text{wp b P s}) \)
by (auto)
finally show \( \text{thesis} \).
qed
qed

Laws depending on the arithmetic of \( a \uplus b \) and \( a \cap b \) together

lemma DC-refines-AC:
\[
(a \cap b) \subseteq (a \uplus b)
\]
by (auto intro!: refinesI simp: wp-eval)

Laws Involving Refinement and Equivalence

lemma pr-trans[trans]:
fixes A :: 'a prog
assumes prAB: A \subseteq B
and prBC: B \subseteq C
shows A \subseteq C
proof
fix P :: 'a \Rightarrow real assume sP: sound P
with prAB have wp A P \vdash wp B P by (blast)
also from sP and prBC have \( \ldots \vdash \text{wp C P} \) by (blast)
finally show wp A P \vdash \ldots .
**CHAPTER 4. THE PGCL LANGUAGE**

**qed**

**lemma** `pequiv-refl[intro,simp]`:
\[ a \simeq a \]
by(auto)

**lemma** `pequiv-comm[ac-simps]`:
\[ a \simeq b \iff b \simeq a \]
unfolding `pequiv-def`
by(rule iffI, safe, simp-all)

**lemma** `pequiv-pr[dest]`:
\[ a \simeq b \Rightarrow a \sqsubseteq b \]
by(auto)

**lemma** `pequiv-trans[intro,trans]`:
\[ [ a \simeq b ; b \sqsubseteq c ] \Rightarrow a \simeq c \]
unfolding `pequiv-def` by(auto intro:order-trans)

**lemma** `pequiv-pr-trans[intro,trans]`:
\[ [ a \sqsubseteq b ; b \simeq c ] \Rightarrow a \sqsubseteq c \]
unfolding `pequiv-def` by(simp)

**Refinement induces equivalence by antisymmetry:**

**lemma** `pequiv-antisym`:
\[ [ a \sqsubseteq b ; b \sqsubseteq a ] \Rightarrow a \simeq b \]
by(auto intro:antisym)

**lemma** `pequiv-DC`:
\[ [ a \simeq c ; b \simeq d ] \Rightarrow (a \sqcap b) \simeq (c \sqcap d) \]
by(auto intro:DC-mono pequiv-antisym simp:ac-simps)

**lemma** `pequiv-AC`:
\[ [ a \simeq c ; b \simeq d ] \Rightarrow (a \sqcup b) \simeq (c \sqcup d) \]
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)`
4.9. THE ALGEBRA OF PGCL

and wa: well-def a
and wb: well-def b
and dr: a ⊑ b
shows a ≃ b

Proof by contradiction.

proof (rule pequivI, rule contrapos-pp)
from wb have feasible (wp b) by (auto)
with wb have s: sub: sub-add (wp b)
  by (auto dest: sublinear-subadd [OF well-def-wp-sublinear])
fix P::'s ⇒ real assume sP: sound P

Assume that a and b are not equivalent:
assume ne: wp a P ≠ wp b P

Find a point at which they differ. As a ⊑ b, wp b P s must by strictly greater than wp a P s here:
hence ∃ s. wp a P s < wp b P s
proof (rule contrapos-np)
  assume ~(∃ s. wp a P s < wp b P s)
  hence ∀ s. wp b P s ≤ wp a P s by (auto simp: not-less)
  hence wp b P ⊢ wp a P by (auto)
  moreover from sP dr have wp a P ⊢ 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 = λ s. bound-of P − P s
have sP: sound ?Pc
proof (rule soundI)
  from sP have ∀ s. 0 ≤ P s by (auto)
  hence ∀ s. ?Pc s ≤ bound-of P by (auto)
  thus bounded ?Pc by (blast)
  from sP have ∃ s. P s ≤ bound-of P by (auto)
  hence ∀ s. 0 ≤ ?Pc s by (auto simp: sign-simps)
  thus nneg ?Pc by (auto)
qed

We then show that wp b violates feasibility, and thus healthiness.
from sP have 0 ≤ bound-of P by (auto)
with da have bound-of P = wp a (λ s. bound-of P) s
  by (simp add: maximalD determ-maximalD)
also have ... = wp a (?Pc s + P s) s
  by (simp)
also from da sP sPc have ... = wp a (?Pc s + wp a P s)
  by (subst additiveD [OF determ-additiveD], simp-all add: sP sPc)
also from sPc dr have ... ≤ wp b ?Pc s + wp a P s
  by (auto)
4.9.4 The Algebraic Structure of Refinement

Well-defined programs form a half-bounded semilattice under refinement, where Abort is bottom, and $a \sqcup b$ is $\text{inf}$. 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 $\forall s \cdot \text{program} = \forall s \cdot \text{program} / \partial \lambda a b. a \simeq b \land \text{well-def} a \land \text{well-def} b$

proof (rule part-equivpI)

have Skip $\simeq$ Skip and well-def Skip by (auto intro: wd-intros)
thus $\exists x. x \simeq x \land \text{well-def} x \land \text{well-def} x$ by (blast)
show symp $(\lambda a b. a \simeq b \land \text{well-def} a \land \text{well-def} b)$
proof (rule sympI, safe)
fix $a$: $\forall s \cdot \text{prog}$ and $b$
assume $a \simeq b$

hence equiv-trans $(\forall p a) (\forall p b)$
by (simp add: pequiv-equiv-trans)
thus $b \simeq a$ by (simp add: ac-simps pequiv-equiv-trans)
qed

show transp $(\lambda a b. a \simeq b \land \text{well-def} a \land \text{well-def} b)$
by (rule transpI, safe, rule pequiv-trans)

qed

instantiation program :: (type) semilattice-inf begin

lift-definition

less-eq-program :: $\forall s \cdot \text{program} \Rightarrow \forall s \cdot \text{program} \Rightarrow \text{bool}$ is refinest

proof (safe)
fix $a$: $\forall s \cdot \text{prog}$ and $b c d$

also from less have ...
by (auto)

also from sab $s P \text{ Pc}$ have ...
by (blast)

finally have $\neg \wp b (\lambda s. \text{ bound-of P} \ s \leq \text{bound-of P})$
by (simp)

thus $\neg \text{bounded-by} (\text{bound-of P}) (\wp b (\lambda s. \text{bound-of P}))$
by (auto)

next

However,

fix $P$: $\forall s \Rightarrow \text{real}$ assume $s P$: sound $P$

hence $\neg \text{neq} (\lambda s. \text{bound-of P})$ by (auto)

moreover have $\text{bounded-by} (\text{bound-of P}) (\lambda s. \text{bound-of P})$ by (auto)

ultimately

show $\text{bounded-by} (\text{bound-of P}) (\wp b (\lambda s. \text{bound-of P}))$
using $\text{wb}$ 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 $a \sqcup b$ is $\text{inf}$. 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 $\forall s \cdot \text{program} =\forall s \cdot \text{program} / \partial \lambda a b. a \simeq b \land \text{well-def} a \land \text{well-def} b$

proof (rule part-equivpI)

have Skip $\simeq$ Skip and well-def Skip by (auto intro: wd-intros)
thus $\exists x. x \simeq x \land \text{well-def} x \land \text{well-def} x$ by (blast)
show symp $(\lambda a b. a \simeq b \land \text{well-def} a \land \text{well-def} b)$
proof (rule sympI, safe)
fix $a$: $\forall s \cdot \text{prog}$ and $b$
assume $a \simeq b$

hence equiv-trans $(\forall p a) (\forall p b)$
by (simp add: pequiv-equiv-trans)
thus $b \simeq a$ by (simp add: ac-simps pequiv-equiv-trans)
qed

show transp $(\lambda a b. a \simeq b \land \text{well-def} a \land \text{well-def} b)$
by (rule transpI, safe, rule pequiv-trans)

qed

instantiation program :: (type) semilattice-inf begin

lift-definition

less-eq-program :: $\forall s \cdot \text{program} \Rightarrow \forall s \cdot \text{program} \Rightarrow \text{bool}$ is refinest

proof (safe)
fix $a$: $\forall s \cdot \text{prog}$ and $b c d$

also from less have ...
by (auto)

also from sab $s P \text{ Pc}$ have ...
by (blast)

finally have $\neg \wp b (\lambda s. \text{ bound-of P} \ s \leq \text{bound-of P})$
by (simp)

thus $\neg \text{bounded-by} (\text{bound-of P}) (\wp b (\lambda s. \text{bound-of P}))$
by (auto)

next

However,

fix $P$: $\forall s \Rightarrow \text{real}$ assume $s P$: sound $P$

hence $\neg \text{neq} (\lambda s. \text{bound-of P})$ by (auto)

moreover have $\text{bounded-by} (\text{bound-of P}) (\lambda s. \text{bound-of P})$ by (auto)

ultimately

show $\text{bounded-by} (\text{bound-of P}) (\wp b (\lambda s. \text{bound-of P}))$
using $\text{wb}$ by (auto dest: well-def-wp-healthy)

qed
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 $a \sqsubseteq c$.

qed

lift-definition

less-program :: 'a$\Rightarrow 'a$ program \Rightarrow 'a program \Rightarrow bool
is $\lambda a$ $b$. $a \sqsubseteq b \land \neg b \sqsubseteq a$

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$\Rightarrow 'a$ 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 \((a \sqcap c) \simeq (b \sqcap d)\) by (rule pequiv-DC)

next
fix a c::'s prog
assume well-def a well-def c
thus well-def \((a \sqcap c)\) by (rule wd-intros)

next
fix a c::'s prog
assume well-def a well-def c
thus well-def \((a \sqcap c)\) by (rule wd-intros)

qed

instance

proof
fix x y::'a program
show \((x < y) = (x \leq y \land \neg y \leq x)\)
  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: \(\forall a b::'s prog. [a \simeq b; determ (wp a)] \rightarrow determ (wp b)\)

proof (intro determI additiveI maximalI)
fix a b::'s prog and P::'s \Rightarrow real
and $Q$'s $\Rightarrow$ real and $s$'s
assume $da$: $\text{determ (wp a)}$
assume $sP$: $\text{sound P}$ and $sQ$: $\text{sound Q}$
and eq: $a \simeq b$
hence $\text{wp b (\lambda s. P s + Q s) s =}$
$\text{wp a (\lambda s. P s + Q s) s}$
by($\text{simp add: sound-intros}$)
also from $da$ $sP sQ$
have $... = \text{wp a P s + wp a Q s}$
by($\text{simp add: additiveD determ-additiveD}$)
also from eq $sP sQ$
also from $da$ $sP sQ$
have $... = \text{wp b P s + wp b Q s}$
by($\text{simp add: pequivD}$)
finally show $\text{wp b (\lambda s. P s + Q s) s = wp b P s + wp b Q s}$.
next
fix $a b$'s prog and $c$::real
assume $a \simeq b$ hence $b \simeq a$ by($\text{simp add: ac-simps}$)
moreover assume $nn$: $0 \leq c$
ultimately have $\text{wp b (\lambda -. c) = wp a (\lambda -. c)}$
by($\text{simp add: pequivD const-sound}$)
also from $da$ $nn$ have $... = (\lambda -. c)$
by($\text{simp add: determ-maximalD maximalD}$)
finally show $\text{wp b (\lambda -. c) = (\lambda -. c)}$.
qed

definition $pdeterm ::$'s program $\Rightarrow$ bool
is $\lambda a. \text{determ (wp a)}$
proof(safe)
fix $a b$'s prog
assume $a \simeq b$ and $\text{determ (wp a)}$
thus $\text{determ (wp b)}$ by($\text{rule eq-det}$)
next
fix $a b$'s prog
assume $a \simeq b$ hence $b \simeq a$ by($\text{simp add: ac-simps}$)
moreover assume $\text{determ (wp b)}$
ultimately show $\text{determ (wp a)}$ by($\text{rule eq-det}$)
qed

lemma $\text{determ-maximal:}$
\begin{align*}
[ \text{pdeterm a; a \leq x } ] & \Rightarrow a = x \\
\text{by(transfer, auto intro:refines-determ)}
\end{align*}

4.9.5 Data Refinement

A projective data refinement construction for pGCL. By projective, we mean
that the abstract state is always a function ($\varphi$) of the concrete state. Re-
finement may be predicated ($G$) on the state.
definition
drefines :: ('b ⇒ 'a) ⇒ ('b ⇒ bool) ⇒ 'a prog ⇒ 'b prog ⇒ bool
where
drefines ϕ G A B ≡ ∀ P Q. (unitary P ∧ unitary Q ∧ (P ⊢ wp A Q)) →→
(«G» && (P o ϕ) ⊢ wp B (Q o ϕ))

lemma drefinesD[dest]:
[ drefines ϕ G A B; unitary P; unitary Q; P ⊢ wp A Q ] →→
«G» && (P o ϕ) ⊢ wp B (Q o ϕ)
unfolding drefines-def by(blast)

We can alternatively use G as an assumption:

lemma drefinesD2:
assumes dr: drefines ϕ G A B
and uP: unitary P
and uQ: unitary Q
and wpA: P ⊢ wp A Q
and G: G s
shows (P o ϕ) s ≤ wp B (Q o ϕ) s
proof –
from uP have 0 ≤ (P o ϕ) s unfolding o-def by(blast)
with G have (P o ϕ) s = («G» && (P o ϕ)) s
  by(simp add:exp-conj-def)
also from assms have ... ≤ wp B (Q o ϕ) s by(blast)
finally show (P o ϕ) s ≤ ...
qed

This additional form is sometimes useful:

lemma drefinesD3:
assumes dr: drefines ϕ G a b
and G: G s
and uQ: unitary Q
and wa: well-def a
shows wp a Q (ϕ s) ≤ wp b (Q o ϕ) 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 ⊢ ?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]:
[ ∀ P Q. [ unitary P; unitary Q; P ⊢ wp A Q ] →→
«G» && (P o ϕ) ⊢ wp B (Q o ϕ) ] →→
drefines ϕ G A B

unfolding \texttt{drefines-def} by (\texttt{blast})

Use G as an assumption, when showing refinement:

\textbf{lemma \texttt{drefinesI2}:}
\begin{itemize}
  \item \texttt{fixes} \( A::'a \text{ prog} \)
  \item and \( B::'b \text{ prog} \)
  \item and \( \varphi::'b \Rightarrow 'a \)
  \item and \( G::'b \Rightarrow \text{ bool} \)
  \item \texttt{assumes} \( wB: \text{ well-def } B \)
  \item \texttt{withAs:} \( \land P \ Q \ s. \ [ \{ \text{unitary } P; \text{ unitary } Q; \ G \ s; P \vdash \vdash \text{wp } A \ Q \} \implies (P \circ \varphi) \ s \leq \text{wp } B (Q \circ \varphi) \ s \)
\end{itemize}

\texttt{shows \texttt{drefines } \varphi \ G A B}

\texttt{proof}
\begin{itemize}
  \item \texttt{fix} \( P \text{ and } Q \)
  \item \texttt{assume} \( uP: \text{ unitary } P \)
  \item and \( uQ: \text{ unitary } Q \)
  \item and \( wpA: P \vdash \vdash \text{wp } A \ Q \)
\end{itemize}

\text{hence} \( \land s. G \ s \implies (P \circ \varphi) \ s \leq \text{wp } B (Q \circ \varphi) \ s \)

\texttt{using \texttt{withAs by (blast)}}

\texttt{moreover}
\begin{itemize}
  \item \texttt{from} \( uQ \text{ have} \text{ unitary } (Q \circ \varphi) \)
  \item \texttt{unfolding \texttt{o-def by (blast)}}
\end{itemize}

\texttt{moreover}
\begin{itemize}
  \item \texttt{from} \( uP \text{ have} \text{ unitary } (P \circ \varphi) \)
  \item \texttt{unfolding \texttt{o-def by (blast)}}
\end{itemize}

\texttt{ultimately}
\begin{itemize}
  \item \texttt{show} \( «G» \& \& (P \circ \varphi) \vdash \vdash \text{wp } B (Q \circ \varphi) \)
  \item \texttt{using} \( wB \text{ by (blast intro: entails-pconj-assumption)} \)
\end{itemize}

\texttt{qed}

\textbf{lemma \texttt{dr-strengthen-guard}:}
\begin{itemize}
  \item \texttt{fixes} \( a::'s \text{ prog and } b::'t \text{ prog} \)
  \item \texttt{assumes} \( fg: \land s. F \ s \implies G \ s \)
  \item and \( drab: \text{drefines } \varphi \ G a b \)
\end{itemize}

\texttt{shows \texttt{drefines } \varphi \ F a b}

\texttt{proof(intro drefinesI)}
\begin{itemize}
  \item \texttt{fix} \( P \ Q::'s \text{ expect} \)
  \item \texttt{assume} \( uP: \text{ unitary } P \text{ and } uQ: \text{ unitary } Q \)
  \item and \( wp: P \vdash \vdash \text{wp } a \ Q \)
\end{itemize}

\texttt{from} \( fg \text{ have} \land s. «F» \ s \leq «G» \ s \text{ by (simp add: embed-bool-def)} \)

\texttt{hence} \( «F» \& \& (P \circ \varphi) \vdash («G» \& \& (P \circ \varphi)) \text{ by (auto intro: pconj-monot le-funI simp: exp-conj-def)} \)

\texttt{also from} \( drab \ uP \ uQ \text{ wp have} \ ... \vdash \text{wp } b (Q \circ \varphi) \text{ by (auto)} \)

\texttt{finally show} \( «F» \& \& (P \circ \varphi) \vdash \vdash \text{wp } b (Q \circ \varphi) \).

\texttt{qed}

Probabilistic correspondence, \texttt{pcorres}, is equality on distribution transform-
ers, modulo a guard. It is the analogue, for data refinement, of program
equivalence for program refinement.

**definition**

\[
\text{pcorres} ::= (\text{‘b} \Rightarrow \text{‘a}) \Rightarrow (\text{‘b} \Rightarrow \text{bool}) \Rightarrow \text{‘a prog} \Rightarrow \text{‘b prog} \Rightarrow \text{bool}
\]

**where**

\[
\text{pcorres} \varphi \ G \ A \ B \mapsto
\left(\forall Q. \ \text{unitary} Q \implies \langle \text{‘G} \rangle \ & \ (\text{wp} \ A \ Q \circ \varphi) = \langle \text{‘G} \rangle \ & \ \text{wp} \ B \ (Q \circ \varphi)\right)
\]

**lemma** pcorresI:

\[
\left[ \left[ \forall Q. \ \text{unitary} Q \implies \langle \text{‘G} \rangle \ & \ (\text{wp} \ A \ Q \circ \varphi) = \langle \text{‘G} \rangle \ & \ \text{wp} \ B \ (Q \circ \varphi) \right] \implies
\text{pcorres} \varphi \ G \ A \ B
\]

by (simp add: pcorres-def)

Often easier to use, as it allows one to assume the precondition.

**lemma** pcorresI2[intro]:

\[
\text{fixes} \ A::\text{‘a prog} \ \text{and} \ B::\text{‘b prog}
\]

assumes

\[
\text{with}\ G: \left[ \forall Q. \ \text{unitary} Q; \ G s \right] \implies \text{wp} \ A \ Q \circ \varphi \iff \text{wp} \ B \ (Q \circ \varphi)
\]

and \ wA: \ well-def \ A

and \ wB: \ well-def \ B

shows \ pcorres \varphi \ G \ A \ B

**proof** (rule pcorresI, rule ext)

fix \ Q::\text{‘a} \ \text{and} \ s::\text{‘b}

assume \ uQ: \ \text{unitary} \ Q

hence \ uQ: \ \text{unitary} \ (Q \circ \varphi) \ by(auto)

show \ \langle \text{‘G} \rangle \ & \ (\text{wp} \ A \ Q \circ \varphi) \ s = \langle \text{‘G} \rangle \ & \ \text{wp} \ B \ (Q \circ \varphi) \ s

**proof**(cases \ G \ s)

- **case** True 
  note this
  moreover
  from well-def-wp-healthy[OF \ wA] uQ have \ 0 \leq \ \text{wp} \ A \ Q \circ \varphi \ s \ by(blast)
  moreover
  from well-def-wp-healthy[OF \ wB] uQ have \ 0 \leq \ \text{wp} \ B \ (Q \circ \varphi) \ 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 \ \text{wp} \ A \ Q \circ \varphi \ s \leq \ 1 \ by(blast)
  moreover
  from well-def-wp-healthy[OF \ wB] uQ have \ \text{wp} \ B \ (Q \circ \varphi) \ s \leq \ 1
    by(blast dest:healthy-bounded-byD intro:sound-nneg)
  ultimately show \ ?thesis \ by(simp add: exp-conj-def)

qed

**lemma** pcorresD:

\[
\left[ \text{pcorres} \varphi \ G \ A \ B; \ \text{unitary} \ Q \right] \implies \langle \text{‘G} \rangle \ & \ (\text{wp} \ A \ Q \circ \varphi) = \langle \text{‘G} \rangle \ & \ \text{wp} \ B \ (Q \circ \varphi)
\]

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`
  - `uQ`: unitary `Q`
  - `wA`: well-def `A` and `wB`: well-def `B`
  - `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 ≤ 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 `sound Q` by(auto)
- Hence `sound (Q o ϕ)` by(auto intro:sound-intros)
  - With well-def-wp-healthy[OF `wB`] have `0 ≤ 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 ⊑ b` and `wb`: well-def `b`
- **Shows** `drefines (λs. s) G a b`

**Proof**

- `(intro drefinesI2 `wb`, simp add:o-def)
- Fix `P::′s ⇒ real` and `Q::′s ⇒ real` and `s::′s`
- Assume `sQ`: `unitary Q`
- Assume `P ⊢ wp a Q hence P s ≤ wp a Q s` by(auto)
- Also from `rab sQ` have `... ≤ wp b Q s` by(auto)
- Finally show `P s ≤ 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`
  - `drBC`: `drefines ϕ' G' B C`
  - `Gimp`: `∀s. G' s ⇒ G (ϕ' s)`
- **Shows** `drefines (ϕ o ϕ') G' A C`

**Proof** (rule `drefinesI`)
fix $P: \forall a \Rightarrow \text{real}$ and $Q: \forall a \Rightarrow \text{real}$ and $s: \forall a$
assume $uP$: unitary $P$ and $uQ$: unitary $Q$
and $wpA: P \vdash wp A Q$

have $\langle G' \rangle \& \& \langle G \circ \varphi' \rangle = \langle G' \rangle$
proof(rule ext, unfold exp-conj-def)
fix $x$
show $\langle G' \rangle x \& \& \langle G \circ \varphi' \rangle x = \langle G' \rangle x$ (is $?X$)
proof(cases $G' x$)
case False then show $?X$ by(simp)
next
case True moreover
with $Gimp$ have $(G \circ \varphi') x$ by(simp add:o-def)
ultimately
show $?X$ by(simp)
qed

with $uP$
have $\langle G' \rangle \& \& (P \circ (\varphi \circ \varphi')) = \langle G' \rangle \& \& ((\langle G \rangle \& \& (P \circ \varphi)) \circ \varphi')$
by(simp add:exp-conj-assoc o-assoc)
also {
from $uP$ $uQ$ $wpA$ and $drAB$
have $\langle G \rangle \& \& (P \circ \varphi) \vdash wp B (Q \circ \varphi)$
by(blast intro:drefinesD)
with $drBC$ and $uP$ $uQ$
have $\langle G' \rangle \& \& ((\langle G \rangle \& \& (P \circ \varphi)) \circ \varphi') \vdash wp C ((Q \circ \varphi) \circ \varphi')$
by(blast intro:unitary-intros drefinesD)
}
finally
show $\langle G' \rangle \& \& (P \circ (\varphi \circ \varphi')) \vdash wp C (Q \circ (\varphi \circ \varphi'))$
by(simp add:o-assoc)
qed

Data refinement composes with program refinement:

lemma pr-dr-trans[trans]:
assumes $prAB: A \sqsubseteq B$
and $drBC$: drefines $\varphi$ $G B C$
shows drefines $\varphi$ $G A C$
proof(rule drefinesI)
fix $P$ and $Q$
assume $uP$: unitary $P$
and $uQ$: unitary $Q$
and $wpA: P \vdash wp A Q$
4.9. THE ALGEBRA OF PGCL

note \( wpA \)
also from \( uQ \) and \( prAB \) have \( wp A Q \vdash wp B Q \) by \((\text{blast})\)
finally have \( P \vdash wp B Q \).
with \( uP uQ \)
show \( \langle G \rangle \& (P o \varphi) \vdash wp C (Q o \varphi) \) by \((\text{blast intro:drefinesD})\)
qed

lemma \( dr-pr-trans[trans] \):
assumes \( drAB: \text{drefines} \varphi G A B \)
assumes \( prBC: B \subseteq C \)
shows \( \text{drefines} \varphi G A C \)
proof (rule drefinesI)
fix \( P \) and \( Q \)
assume \( uP: \text{unitary} P \)
and \( uQ: \text{unitary} Q \)
and \( wpA: P \vdash wp A Q \)
with \( drAB \) have \( \langle G \rangle \& (P o \varphi) \vdash wp B (Q o \varphi) \) by \((\text{blast intro:drefinesD})\)
also from \( uQ \)
have \( ... \vdash wp C (Q o \varphi) \) by \((\text{blast})\)
finally show \( \langle G \rangle \& (P o \varphi) \vdash ... \).
qed

If the projection \( \varphi \) commutes with the transformer, then data refinement is reflexive:

lemma \( dr-refl \):
assumes \( wa: \text{well-def} a \)
and \( \text{comm}: Q. \text{unitary} Q \longrightarrow wp a Q \varphi \vdash wp a (Q o \varphi) \)
shows \( \text{drefines} \varphi G a a \)
proof (intro drefinesI2 wa)
fix \( P \) and \( Q \) and \( s \)
assume \( wp: P \vdash wp a Q \)
assume \( uQ: \text{unitary} Q \)

have \( (P o \varphi) s = P (\varphi s) \) by \((\text{simp})\)
also from \( wp \) have \( ... \leq wp a Q (\varphi s) \) by \((\text{blast})\)
also \{ 
from \( \text{comm} uQ \) have \( wp a Q o \varphi \vdash wp a (Q o \varphi) \) by \((\text{blast})\)

hence \( wp a Q o \varphi) s \leq wp a (Q o \varphi) s \) by \((\text{blast})\)

hence \( wp a Q (\varphi s) \leq ... \) by \((\text{simp})\)
\}
finally show \( (P o \varphi) s \leq wp a (Q o \varphi) s \).
qed

Correspondence implies data refinement

lemma \( pcorres-drefine \):
assumes \( pcorres: \text{pcorres} \varphi G A C \)
and \( wC: \text{well-def} C \)
shows \( \text{drefines} \varphi G A C \)
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 refinement, « previous result, the second must be somewhere strictly larger than the first:

If the programs do not correspond, the terms must differ somewhere, and given the

4.9. THE ALGEBRA OF PGCL

197

The transformers themselves must differ at this point:

| 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 ne1: wp a Q o φ ≠ wp b (Q o φ) by(auto) |

From refinement, « G » &(& (wp a Q o φ) lies below « G » &(& wp b (Q o φ)).

| 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:pcconj-mono) |
| moreover from uQ ha have wp a Q (φ s) ≤ 1 by(auto dest:healthy-bounded-byD) |
| moreover from uQ ha have θ ≤ wp a Q (φ s) by(auto) |
| ultimately show « G » s .& (wp a Q o φ) s ≤ « G » s .& wp b (Q o φ) s by(simp add:pcconj-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 rle: ∃ 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 o φ)) s < («G» &(& wp b (Q o φ)) s) |
| thus («G» &(& wp b (Q o φ))) s ≤ («G» &(& wp a Q o φ)) s by(simp add:not-less) |

| qed |
| from this obtain s where less-s: («G» &(& (wp a Q o φ)) s < («G» &(& wp b (Q o φ)) s by(blast) |

The transformers themselves must differ at this point:
hence \( \text{larger: } \text{wp a } Q (\varphi s) < \text{wp b } (Q \circ \varphi) s \)

proof(cases \( G s \))

  case True
  moreover from \( ha uQ \) have \( \theta \leq \text{wp a } Q (\varphi s) \)
  by(blast)
  moreover from \( hb uQ \) have \( \theta \leq \text{wp b } (Q \circ \varphi) s \)
  by(blast)
  moreover note less-s
  ultimately show \( \text{thesis by(simp add:exp-conj-def) } \)

next
  case False
  moreover from \( ha uQ \) have \( \text{wp a } Q (\varphi s) \leq 1 \)
  by(blast)
  moreover {
    from \( uQ \) have \( \text{bounded-by 1 } (Q \circ \varphi) \)
    by(blast)
    moreover from \( \text{unitary-sound[OF } uQ] \)
    have \( \text{sound } (Q \circ \varphi) \) by(auto)
    ultimately have \( \text{wp b } (Q \circ \varphi) s \leq 1 \)
    using \( hb \) by(auto)
  }
  moreover note less-s
  ultimately show \( \text{thesis by(simp add:exp-conj-def) } \)

qed

from less-s have \( (\langle G \rangle \&\& (\text{wp a } Q \circ \varphi)) s \neq (\langle G \rangle \&\& \text{wp b } (Q \circ \varphi)) s \)
  by(force)

\( G \) must also hold, as otherwise both would be zero.

hence \( G-s: G s \)

proof(rule contrapos-np)

  assume \( nG: \neg G s \)
  moreover from \( ha uQ \) have \( \text{wp a } Q (\varphi s) \leq 1 \)
  by(blast)
  moreover {
    from \( uQ \) have \( \text{bounded-by 1 } (Q \circ \varphi) \)
    by(blast)
    moreover from \( \text{unitary-sound[OF } uQ] \)
    have \( \text{sound } (Q \circ \varphi) \) by(auto)
    ultimately have \( \text{wp b } (Q \circ \varphi) s \leq 1 \)
    using \( hb \) by(auto)
  }
  ultimately
  show \( (\langle G \rangle \&\& (\text{wp a } Q \circ \varphi)) s = (\langle G \rangle \&\& \text{wp b } (Q \circ \varphi)) s \)
  by(simp add:exp-conj-def)

qed

Take a carefully constructed expectation:

let \( \text{?Qc} = \lambda s. \text{bound-of } Q - Q s \)

have \( \text{bQc: bounded-by 1 } \text{?Qc} \)
4.9. THE ALGEBRA OF PGCL

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 \( \text{wp } a \ Q (\varphi \ s) + \text{wp } a \ ?Q_c (\varphi \ s) < \)
\( \text{wp } b \ (Q \ o \ \varphi) \ s + \text{wp } b \ (\ ?Q_c \ o \ \varphi) \ s . \)

\}

also from \( \text{sab} \) and unitary-sound \( \text{OF uQ} \) and \( sQc \)
have \( \text{wp } b \ (Q \ o \ \varphi) \ s + \text{wp } b \ (\ ?Q_c \ o \ \varphi) \ s \leq \)
\( \text{wp } b \ (\lambda s. \ (Q \ o \ \varphi) \ s + (\ ?Q_c \ o \ \varphi) \ s) \ s \)
by(blast)
also have \( ... = \text{wp } b \ (\lambda s. \ \text{bound-of } Q) \ s \)
by(simp)
finally
show \( \neg \text{feasible } (wp \ b) \)
proof(rule contrapos-pn)
assume \( \text{fb} : \text{feasible } (wp \ b) \)
have bounded-by (bound-of \( Q \)) \( (\lambda s. \ \text{bound-of } Q) \ by(blast) \)
hence bounded-by (bound-of \( Q \)) \( (wp \ b \ (\lambda s. \ \text{bound-of } Q)) \)
using \( uQ \ by(blast \ intro: \text{feasible-bounded D}[\text{OF } \text{fb}]) \)
hence \( wp \ b \ (\lambda s. \ \text{bound-of } Q) \ s \leq \text{bound-of } Q \ by(blast) \)
thus \( \neg \text{bound-of } Q < wp \ b \ (\lambda s. \ \text{bound-of } Q) \ s \ by(simp) \)
qed
qed

4.9.7 Structural Rules for Correspondence

lemma \( \text{pcorres-Skip} : \)
\( \text{pcorres } \varphi \ G \ \text{Skip} \ \text{Skip} \)
by(simp add:pcorres-def wp-eval)

Correspondence composes over sequential composition.

lemma \( \text{pcorres-Seq} : \)
fixes \( A : ‘b \ \text{prog} \) and \( B : ‘c \ \text{prog} \)
and \( C : ‘b \ \text{prog} \) and \( D : ‘c \ \text{prog} \)
and \( \varphi : ‘c \Rightarrow ‘b \)
assumes \( \text{pcAB} : \text{pcorres } \varphi \ G \ A \ B \)
and \( \text{pcCD} : \text{pcorres } \varphi \ H \ C \ D \)
and \( \text{wA} : \text{well-def } A \) and \( \text{wB} : \text{well-def } B \)
and \( \text{wC} : \text{well-def } C \) and \( \text{wD} : \text{well-def } D \)
and \( \text{p3p2} : \forall Q. \ \text{unitary } Q \Longrightarrow «I» \ &\& \ wp \ B \ Q = \ wp \ B \ («H» \ &\& \ Q) \)
and \( \text{p1p3} : \forall \ s. \ G \ s \Longrightarrow I \ s \)
shows \( \text{pcorres } \varphi \ G \ (A;C) \ (B;D) \)
proof(rule pcorresI)
fix \( Q : ‘b \Rightarrow \text{real} \)
assume \( uQ : \text{unitary } Q \)
with \( \text{well-def-wp-healthy}[\text{OF } wC] \) have \( uCQ : \text{unitary } (wp \ C \ Q) \ by(auto) \)
from \( uQ \) well-def-wp-healthy[\text{OF } wD] have \( uDQ : \text{unitary } (wp \ D \ (Q \ o \ \varphi)) \)
by(auto dest:unitary-comp)

have \( \text{p3p1 : } \forall \ R, S. \ [ \text{unitary } R; \ \text{unitary } S; \ «I» \ &\& \ R = «I» \ &\& \ S ] \Longrightarrow \)
\( «G» \ &\& \ R = «G» \ &\& \ S \]
proof(rule ext)
4.9. THE ALGEBRA OF PGCL

fix $R :: 'c \Rightarrow \text{real}$ and $S :: 'c \Rightarrow \text{real}$ and $s :: 'c$

assume $a3: \langle I \rangle \& \& R = \langle I \rangle \& \& S$
and $uR: \text{unitary } R$ and $uS: \text{unitary } S$
show $(G \& \& R) s = (G \& \& S) s$

proof (simp add: exp-conj-def, cases $G s$)

  case False
  note this
moreover from $uR$ have $R s \leq 1$ by (blast)
moreover from $uS$ have $S s \leq 1$ by (blast)
ultimately show $\langle G \rangle s . \& R s = \langle G \rangle s . \& S s$
  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 $\langle G \rangle s . \& R s = \langle G \rangle s . \& S s$
  by (simp)
qed

4.9.8 Structural Rules for Data Refinement

lemma dr-Skip:
  fixes $\varphi :: 'c \Rightarrow 'b$
  shows drefines $\varphi \ G \ 	ext{Skip} \ \text{Skip}$
proof (intro drefinesI2 wd-intros)

show $\langle G \rangle \& \& (\text{wp } (A ;; C) Q \circ \varphi) = \langle G \rangle \& \& \text{wp } (B ;; D) (Q \circ \varphi)$
proof (simp add: wp-eval)
  from $uCQ$ pcAB have $\langle G \rangle \& \& (\text{wp } A (\text{wp } C Q) \circ \varphi) =$
    $\langle G \rangle \& \& \text{wp } B ((\text{wp } C Q) \circ \varphi)$
    by (auto dest: pcorresD)
  also have $\langle G \rangle \& \& \text{wp } B ((\text{wp } C Q) \circ \varphi) =$
    $\langle G \rangle \& \& \text{wp } B (\text{wp } D (Q \circ \varphi))$
  proof (rule $p3p1$
    from $uCQ$ well-def-wp-healthy[OF $wB$] show unitary (wp $B (\text{wp } C Q \circ \varphi))$
    by (auto intro: unitary-comp)
    from $uDQ$ well-def-wp-healthy[OF $wB$] show unitary (wp $B (\text{wp } D (Q \circ \varphi))$
    by (auto)
    from $uQ$ have $\langle H \rangle \& \& (\text{wp } C Q \circ \varphi) = \langle H \rangle \& \& \text{wp } D (Q \circ \varphi)$
      by (blast intro: pcorresD[OF $pcCD$])
    thus $\langle I \rangle \& \& \text{wp } B (\text{wp } C Q \circ \varphi) = \langle I \rangle \& \& \text{wp } B (\text{wp } D (Q \circ \varphi))$
      by (simp add: $p3p2$ $uCQ$ $uDQ$)
qed
finally show $\langle G \rangle \& \& (\text{wp } A (\text{wp } C Q) \circ \varphi) = \langle G \rangle \& \& \text{wp } B (\text{wp } D (Q \circ \varphi))$
  qed
qed
\textbf{CHAPTER 4. THE PGCL LANGUAGE}

\begin{verbatim}
fix \(P::'b \Rightarrow \text{real and } Q::'b \Rightarrow \text{real and } s::'c\)
assume \(P \vdash \text{wp Skip } Q\)
hence \((P \circ \varphi) \ s \leq \text{wp Skip } Q \ (\varphi \ s)\) \ by\(\text{(simp, blast)}\)
thus \((P \circ \varphi) \ s \leq \text{wp Skip } (Q \circ \varphi) \ s\) \ by\(\text{(simp add:wp-eval)}\)
\textbf{qed}

\textbf{lemma \(\text{dr-Abort:}\)}
fixes \(\varphi::'c \Rightarrow 'b\)
shows \(\text{drefines } \varphi \ G \ \text{Abort} \ \text{Abort}\)
\textbf{proof}(\text{intro \text{drefinesI2 \ wd-intros})}
fix \(P::'b \Rightarrow \text{real and } Q::'b \Rightarrow \text{real and } s::'c\)
assume \(P \vdash \text{wp } \text{Abort } Q\)
hence \(\text{(P } \circ \varphi) \ s \leq \text{wp } \text{Abort } Q \ (\varphi \ s)\) \ by\(\text{(auto)}\)
thus \((\text{P } \circ \varphi) \ s \leq \text{wp } \text{Abort } (Q \circ \varphi) \ s\) \ by\(\text{(simp add:wp-eval)}\)
\textbf{qed}

\textbf{lemma \(\text{dr-Apply:}\)}
fixes \(\varphi::'c \Rightarrow 'b\)
assumes \(\text{commutes: } f \circ \varphi = \varphi \circ g\)
shows \(\text{drefines } \varphi \ G \ (\text{Apply } f) \ (\text{Apply } g)\)
\textbf{proof}(\text{intro \text{drefinesI2 \ wd-intros})}
fix \(P::'b \Rightarrow \text{real and } Q::'b \Rightarrow \text{real and } s::'c\)
assume \(\text{wp: } P \vdash \text{wp } (\text{Apply } f) \ Q\)
hence \(P \vdash (Q \circ f)\) \ by\(\text{(simp add:wp-eval)}\)
hence \(P \ (\varphi \ s) \leq (Q \circ f) \ (\varphi \ s)\) \ by\(\text{(blast)}\)
also have \... = \(Q (f \circ \varphi) \ s\) \ by\(\text{(simp)}\)
also with \(\text{commutes}\)
have \... = \((Q \circ \varphi) \ o \ g) \ s\) \ by\(\text{(simp)}\)
also have \... = \(\text{wp } (\text{Apply } g) \ (Q \circ \varphi) \ s\)
\quad by\(\text{(simp add:wp-eval)}\)
finally show \((P \circ \varphi) \ s \leq \text{wp } (\text{Apply } g) \ (Q \circ \varphi) \ s\) \ by\(\text{(simp)}\)
\textbf{qed}

\textbf{lemma \(\text{dr-Seq:}\)}
assumes \(\text{drAB: } \text{drefines } \varphi \ P \ A \ B\)
\quad \text{and } \(\text{drBC: } \text{drefines } \varphi \ Q \ C \ D\)
\quad \text{and wpB: } «P» \vdash \text{wp } B \ «Q»
\quad \text{and wB: } \text{well-def } B
\quad \text{and wC: } \text{well-def } C
\quad \text{and wD: } \text{well-def } D
shows \(\text{drefines } \varphi \ P \ (A::C) \ (B::D)\)
\textbf{proof}
fix \(R \text{ and } S\)
assume \(\text{uR: } \text{unitary } R \text{ and } \text{uS: } \text{unitary } S\)
\quad \text{and wpAC: } R \vdash \text{wp } (A::C) \ S
from \(\text{uR}\)
have \(«P» \&\& (R \circ \varphi) = «P» \&\& («P» \&\& (R \circ \varphi))\)
\end{verbatim}
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 \; \psi) \vdash wp\; B\; (wp\; C\; o\; \psi)
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 \(«Q» \& \& («Q» \& \& (R \; o \; \psi)) \vdash wp\; B\; («Q» \& \& (wp\; C\; o\; \psi))
by (auto intro!: entails-combine dest!: unitary-sound)
}
also 
from uS well-def-wp-healthy \([OF\; wC]\)

have \(«Q» \& \& (wp\; C\; o\; \psi) \vdash wp\; D\; (S\; o\; \psi)
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\; o\; \psi)) \vdash wp\; B\; (wp\; D\; (S\; o\; \psi))
by (blast intro!: mono-transD)
}

finally

show \(«P» \& \& (R \; o \; \psi) \vdash wp\; (B;;\; D)\; (S\; o\; \psi)
unfolding wp-eval o-def.

qed

lemma dr-repeat:
fixes \(\psi :: \;'a \Rightarrow \;'b
assumes dr-ab: drefines \(\psi \; G\; a\; b
and Gpr: \(«G» \vdash wp\; b\; «G»
and wa: \; well-def \; a
and wb: \; well-def \; b
shows drefines \(\psi \; 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:**

\[(\lambda s. 0) \vdash wp\text{ Abort }Q\]

**Unfolding** \(wp\text{-eval}\) **by** \((simp)\)

**Lemma wlp-Abort:**

\[(\lambda s. 1) \vdash wlp\text{ Abort }Q\]

**Unfolding** \(wp\text{-eval}\) **by** \((simp)\)

**Lemma wp-Skip:**

\(P \vdash wp\text{ Skip }P\)

**Unfolding** \(wp\text{-eval}\) **by** \((blast)\)

**Lemma wlp-Skip:**

\(P \vdash wlp\text{ Skip }P\)

**Unfolding** \(wp\text{-eval}\) **by** \((blast)\)

**Lemma wp-Apply:**

\(Q \circ f \vdash wp\text{ (Apply }f)\text{ }Q\)

**Unfolding** \(wp\text{-eval}\) **by** \((simp)\)

**Lemma wlp-Apply:**

\(Q \circ f \vdash wlp\text{ (Apply }f)\text{ }Q\)

**Unfolding** \(wp\text{-eval}\) **by** \((simp)\)

**Lemma wp-Seq:**

**Assumes**

\(\text{ent-a: } P \vdash wp\text{ a }Q\)

\(\text{ent-b: } Q \vdash wp\text{ b }R\)

\(\text{wa: well-def a}\)

\(\text{wb: well-def b}\)

\(\text{s-Q: sound }Q\)

\(\text{s-R: sound }R\)

**Shows**

\(P \vdash wp\text{ (a ;; b) }R\)

**Proof**

- **Note** \(ha = \text{well-def-wp-healthy[OF wa]}\)
- **Note** \(hb = \text{well-def-wp-healthy[OF wb]}\)
- **Note** \(ent-a\)
- **Also from** \(ent-b\) \(ha\) \(hb\) \(s-Q\) \(s-R\) **Have** \(wp\text{ a Q }\vdash wp\text{ a (wp b R)}\)
  **by** \((\text{blast intro:healthy-monoD2})\)
- **Finally show** ?thesis **by** \((simp add:wp\text{-eval})\)

**Qed**

**Lemma wlp-Seq:**
assumes \( \text{ent-a}: P \vdash \text{wlp} \ a \ Q \)
\( \text{and} \ \text{ent-b}: Q \vdash \text{wlp} \ b \ R \)
\( \text{and} \ \text{wa}: \ \text{well-def} \ a \)
\( \text{and} \ \text{wb}: \ \text{well-def} \ b \)
\( \text{and} \ u-Q: \ \text{unitary} \ Q \)
\( \text{and} \ u-R: \ \text{unitary} \ R \)
shows \( P \vdash \text{wlp} (a ;; b) R \)

proof
- note \( ha = \text{well-def-wlp-nearly-healthy}[OF \ wa] \)
- note \( hb = \text{well-def-wlp-nearly-healthy}[OF \ wb] \)
- note \( \text{ent-a} \)
also from \( \text{ent-b} \ ha \ hb \ u-Q \ u-R \) have \( \text{wlp} \ a \ Q \vdash \text{wlp} \ a \ (\text{wlp} \ b \ R) \)
  by \((\text{blast intro:nearly-healthy-monoD}[OF \ ha])\)
finally show ?thesis by \((\text{simp add:wp-eval})\)

qed

lemma \( \text{wp-PC}: \)
\( (\lambda s. P s * \text{wp} \ a \ Q s + (1 - P s) * \text{wp} \ b \ Q s) \vdash \text{wp} \ (a \oplus b) Q \)
by \((\text{simp add:wp-eval})\)

lemma \( \text{wlp-PC}: \)
\( (\lambda s. P s * \text{wlp} \ a \ Q s + (1 - P s) * \text{wlp} \ b \ Q s) \vdash \text{wlp} \ (a \oplus b) Q \)
by \((\text{simp add:wp-eval})\)

A simpler rule for when the probability does not depend on the state.

lemma \( \text{PC-fixed}: \)
assumes \( \text{wpa}: P \vdash a \ ab \ R \)
\( \text{and} \ \text{wpb}: Q \vdash b \ ab \ R \)
\( \text{and} \ np: 0 \leq p \) and \( bp: p \leq 1 \)
shows \( (\lambda s. p * P s + (1 - p) * Q s) \vdash (a (\lambda s. p) \oplus b) \ ab \ R \)
unfolding \( \text{PC-def} \)
proof \((\text{rule le-funI})\)
fix \( s \)
from \( \text{wpa} \) and \( np \) have \( p * P s \leq p * a \ ab \ R \)
  by \((\text{auto intro:mult-left-mono})\)
moreover {
  from \( bp \) have \( 0 \leq 1 - p \) by \((\text{simp})\)
  with \( \text{wpb} \) have \( (1 - p) * Q s \leq (1 - p) * b \ ab \ R \)
  by \((\text{auto intro:mult-left-mono})\)
}
ultimately show \( p * P s + (1 - p) * Q s \leq p * a \ ab \ R \) s + (1 - p) * b \ ab \ R \)
  by \((\text{rule add-mono})\)

qed

lemma \( \text{wp-PC-fixed}: \)
\[ [ P \vdash \text{wp} \ a \ R; Q \vdash \text{wp} \ b \ R; 0 \leq p; p \leq 1 ] \implies \]
\( (\lambda s. p * P s + (1 - p) * Q s) \vdash \text{wp} \ (a (\lambda s. p) \oplus b) R \)
by \((\text{simp add:wp-def PC-fixed})\)
lemma \textbf{wlp-PC-fixed}:
\[
\begin{align*}
\[ P \vdash \text{wlp } a \ R; Q \vdash \text{wlp } b \ R; 0 \leq p; p \leq 1 \] \implies \\
(\lambda s. \ p \cdot P s + (1 - p) \cdot Q s) \vdash \text{wlp } (a \ (\lambda s. \ p) \oplus b) \ R \\
\hspace{1em}\text{by}(\text{simp add: wlp-def PC-fixed})
\end{align*}
\]

lemma \textbf{wp-DC}:
\[
\begin{align*}
(\lambda s. \ \text{min } (wp a Q s) \ (wp b Q s)) \vdash wp \ (a \sqcap b) \ Q \\
\hspace{1em}\text{unfolding wp-eval by}(\text{simp})
\end{align*}
\]

lemma \textbf{wlp-DC}:
\[
\begin{align*}
(\lambda s. \ \text{min } (wlp a Q s) \ (wlp b Q s)) \vdash wlp \ (a \sqcap b) \ Q \\
\hspace{1em}\text{unfolding wlp-eval by}(\text{simp})
\end{align*}
\]

Combining annotations for both branches:

lemma \textbf{DC-split}:
\[
\begin{align*}
\text{fixes } a::'s \ \text{prog and } b \\
\text{assumes } \text{wp}a: P \vdash a \ ab \ R \\
\hspace{1em}\text{and wp}b: Q \vdash b \ ab \ R \\
\text{shows } (\lambda s. \ \text{min } (P s) \ (Q s)) \vdash (a \sqcap b) \ ab \ R \\
\hspace{1em}\text{unfolding DC-def}
\end{align*}
\]

proof \textbf{(rule le-funI)}
\[
\begin{align*}
\text{fix } s \\
\text{from wp}a \ \text{wp}b \\
\text{have } P s \leq a \ ab \ R s \ \text{and } Q s \leq b \ ab \ R s \ \text{by}(auto) \\
\hspace{1em}\text{thus } \text{min } (P s) \ (Q s) \leq \text{min } (a \ ab \ R s) \ (b \ ab \ R s) \ \text{by}(auto) \\
\text{qed}
\end{align*}
\]

lemma \textbf{wp-DC-split}:
\[
\begin{align*}
\[ P \vdash wp \ \text{prog } R; Q \vdash wp \ \text{prog}' \ R \] \implies \\
(\lambda s. \ \text{min } (P s) \ (Q s)) \vdash wp \ (\text{prog } \sqcap \ \text{prog}') \ R \\
\hspace{1em}\text{by}(\text{simp add: wp-def DC-split})
\end{align*}
\]

lemma \textbf{wlp-DC-split}:
\[
\begin{align*}
\[ P \vdash wlp \ \text{prog } R; Q \vdash wlp \ \text{prog}' \ R \] \implies \\
(\lambda s. \ \text{min } (P s) \ (Q s)) \vdash wlp \ (\text{prog } \sqcap \ \text{prog}') \ R \\
\hspace{1em}\text{by}(\text{simp add: wlp-def DC-split})
\end{align*}
\]

lemma \textbf{wp-DC-split-same}:
\[
\begin{align*}
\[ P \vdash wp \ \text{prog } Q; P \vdash wp \ \text{prog}' \ Q \] \implies P \vdash wp \ (\text{prog } \sqcap \ \text{prog}') \ Q \\
\hspace{1em}\text{unfolding wp-eval by}(\text{blast intro:min.boundedI})
\end{align*}
\]

lemma \textbf{wlp-DC-split-same}:
\[
\begin{align*}
\[ P \vdash wlp \ \text{prog } Q; P \vdash wlp \ \text{prog}' \ Q \] \implies P \vdash wlp \ (\text{prog } \sqcap \ \text{prog}') \ Q \\
\hspace{1em}\text{unfolding wp-eval by}(\text{blast intro:min.boundedI})
\end{align*}
\]

lemma \textbf{SetPC-split}:
\[
\begin{align*}
\text{fixes } f::'x \Rightarrow 'y \ \text{prog} \\
\hspace{1em}\text{and } p::'y \Rightarrow 'x \Rightarrow \text{real}
\end{align*}
\]
4.10. STRUCTURED REASONING

assumes rec: $\forall x. x \in \text{supp} (p s) \Rightarrow P x \vdash f x ab Q$

and $\text{nnp}: \forall s. \text{nneg} (p s)$

shows $(\forall s. \sum x \in \text{supp} (p s), p s x * P x s) \vdash \text{SetPC f p ab Q}$

unfolding $\text{SetPC-def}$

proof (rule le-funI)

fix $s$

from $\text{rec}$ have $\forall x. x \in \text{supp} (p s) \Rightarrow P x s \leq f x ab Q s$ by (blast)

moreover from $\text{nnp}$ have $\forall x. 0 \leq p s x$ by (blast)

ultimately have $\forall x. x \in \text{supp} (p s) \Rightarrow p s x * P x s \leq p s x * f x ab Q s$

by (blast intro: mult-left-mono)

thus $(\sum x \in \text{supp} (p s), p s x * P x s) \leq (\sum x \in \text{supp} (p s), p s x * f x ab Q s)$

by (rule setsum mono)

qed

lemma $\text{wp-SetPC-split}$:

$[ \forall x. x \in \text{supp} (p s) \Rightarrow P x \vdash wp (f x) Q; \forall s. \text{nneg} (p s) ] \Rightarrow$

$(\forall s. \sum x \in \text{supp} (p s), p s x * P x s) \vdash wp (\text{SetPC f p}) Q$

by (simp add: wp-def SetPC-split)

lemma $\text{wlp-SetPC-split}$:

$[ \forall x. x \in \text{supp} (p s) \Rightarrow P x \vdash wlp (f x) Q; \forall s. \text{nneg} (p s) ] \Rightarrow$

$(\forall s. \sum x \in \text{supp} (p s), p s x * P x s) \vdash wlp (\text{SetPC f p}) Q$

by (simp add: wp-def SetPC-split)

lemma $\text{wp-SetDC-split}$:

$[ \forall s. x. x \in S s \Rightarrow P x \vdash wp (f x) Q; \forall s. S s \neq \{\} ] \Rightarrow$

$P \vdash wp (\text{SetDC f S}) Q$

by (rule le-funI, unfold wp-eval, blast intro!: cInf-greatest)

lemma $\text{wlp-SetDC-split}$:

$[ \forall s. x. x \in S s \Rightarrow P x \vdash wlp (f x) Q; \forall s. S s \neq \{\} ] \Rightarrow$

$P \vdash wlp (\text{SetDC f S}) Q$

by (rule le-funI, unfold wp-eval, blast intro!: cInf-greatest)

lemma $\text{wp-SetDC}$:

assumes $wp$: $\forall s. x. x \in S s \Rightarrow P x \vdash wp (f x) Q$

and $\text{nc}$: $\forall s. S s \neq \{\}$

and $\text{sP}$: $\forall x. \text{sound} (P x)$

shows $\forall s. \text{inf} ((\forall x. P x s) \cdot S s) \vdash wp (\text{SetDC f S}) Q$

using assms by (intro le-funI, simp add: wp-eval del: Inf-image-eq, blast intro!: cInf mono)

lemma $\text{wlp-SetDC}$:

assumes $wp$: $\forall s. x. x \in S s \Rightarrow P x \vdash wlp (f x) Q$

and $\text{nc}$: $\forall s. S s \neq \{\}$

and $\text{sP}$: $\forall x. \text{sound} (P x)$

shows $\forall s. \text{inf} ((\forall x. P x s) \cdot S s) \vdash wlp (\text{SetDC f S}) Q$

using assms by (intro le-funI, simp add: wp-eval del: Inf-image-eq, blast intro!: cInf mono)

lemma $\text{wp-Embed}$:
\[ P \vdash t \quad \Rightarrow \quad P \vdash \text{wp} (\text{Embed} \ t) \quad Q \]
\[ \text{by}(\text{simp add: wp-def Embed-def}) \]

**Lemma wp-Embed:**
\[ P \vdash t \quad \Rightarrow \quad P \vdash \text{wp} (\text{Embed} \ t) \quad Q \]
\[ \text{by}(\text{simp add: wp-def Embed-def}) \]

**Lemma wp-Bind:**
\[ \forall s. \quad P \ s \leq \text{wp} (a (f s)) \quad Q \ s \quad \Rightarrow \quad P \vdash \text{wp} (\text{Bind} \ f \ a) \quad Q \]
\[ \text{by}(\text{auto simp add: wp-def Bind-def}) \]

**Lemma wlp-Bind:**
\[ \forall s. \quad P \ s \leq \text{wlp} (a (f s)) \quad Q \ s \quad \Rightarrow \quad P \vdash \text{wlp} (\text{Bind} \ f \ a) \quad Q \]
\[ \text{by}(\text{auto simp add: wlp-def Bind-def}) \]

**Lemma wp-repeat:**
\[ P \vdash \text{wp} \ a \ Q; \ Q \vdash \text{wp} (\text{repeat} \ n \ a) \ R; \quad \text{well-def} \ a; \ \text{sound} \ Q; \ \text{sound} \ R \quad \Rightarrow \quad P \vdash \text{wp} (\text{repeat} \ (\text{Suc} \ n) \ a) \ R \]
\[ \text{by}(\text{auto intro: wp-Seq wd-intros}) \]

**Lemma wlp-repeat:**
\[ P \vdash \text{wlp} \ a \ Q; \ Q \vdash \text{wlp} (\text{repeat} \ n \ a) \ R; \quad \text{well-def} \ a; \ \text{unitary} \ Q; \ \text{unitary} \ R \quad \Rightarrow \quad P \vdash \text{wlp} (\text{repeat} \ (\text{Suc} \ n) \ a) \ R \]
\[ \text{by}(\text{auto intro: wlp-Seq wd-intros}) \]

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=**

\[ \text{entails-strengthen-post}[\text{where} \ \ t= \text{wp} \ a \ \ \text{for} \ a] \]

**Lemma wp-strengthen-post:**
\[ P \vdash \text{wp} \ a \ Q \quad \Rightarrow \quad \text{nearly-healthy} \ (\text{wp} \ a) \quad \Rightarrow \quad \text{unitary} \ R \quad \Rightarrow \quad Q \vdash \text{unitary} \]
\[ P \vdash \text{wp} \ a \ R \]
\[ \text{by}(\text{blast intro: entails-trans}) \]

**Lemmas wp-weaken-pre=**

\[ \text{entails-weaken-pre}[\text{where} \ \ t= \text{wp} \ a \ \ \text{for} \ a] \]

**Lemmas wlp-weaken-pre=**

\[ \text{entails-weaken-pre}[\text{where} \ \ t= \text{wlp} \ a \ \ \text{for} \ a] \]

**Lemmas wp-scale=**

\[ \text{entails-scale}[\text{where} \ \ t= \text{wp} \ a \ \ \text{for} \ a, \ \text{OF} \ - \ \text{well-def-wp-healthy}] \]
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:
\[ [a \sqsubseteq b; P \triangleright wp a Q] \implies P \vdash wp b Q \]
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 ⇒ real) ⇒ ′a prog ⇒ (′a ⇒ real) ⇒ bool (⌈-⌉ - ⌈-⌉p)
where
wp-valid P prog Q ≡ P \vdash \triangleright wp prog Q

lemma wp-validI:
P \triangleright wp prog Q \implies \{P\}_p \triangleright \{Q\}_p
unfolding wp-valid-def by(assumption)

lemma wp-validD:
\{P\}_p \triangleright \{Q\}_p \implies P \triangleright wp prog Q
unfolding wp-valid-def by(assumption)

lemma valid-Seq:
\[ [\{P\}_a \{Q\}_b; \{Q\} b \{R\}_p; \text{well-def } a; \text{well-def } b; \text{sound } Q; \text{sound } R ] \implies \{P\}_a \{R\}_p \]
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

definition 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  
and \( ws :: 's \) well-def s  
and \( ts :: (\lambda s. 1) \)  
**assumes** \( mr :: \text{maximal} \ (wp \ r) \)

**shows** \( (\lambda s. 1) \vdash wp \ (r ;; s) \ (\lambda s. 1) \)

**proof**

- **note** \( hs = \text{well-def-wp-healthy[OF } ws] \)
- **have** \( wp \ s \ (\lambda s. 1) = (\lambda s. 1) \)
- **proof** (rule antisym)
  - **show** \( (\lambda s. 1) \vdash wp \ s \ (\lambda s. 1) \) by (rule \( ts \))
  - **have** bounded-by \( I \ (wp \ s \ (\lambda s. 1)) \) by (auto intro! healthy-bounded-byD [OF hs])
  - **thus** \( wp \ s \ (\lambda s. 1) \vdash (\lambda s. 1) \) 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** \( wb :: \text{well-def body} \)

**shows** \( « N \ G » \vdash wp \ do \ G \rightarrow od \ (\lambda s. 1) \)

**proof** (rule le-funI)

- **note** \( hb = \text{well-def-wp-healthy[OF } wb] \)
- **fix** \( s \)
- **show** \( « N \ G » \ s \leq wp \ do \ G \rightarrow od \ (\lambda s. 1) \ s \)
- **proof**(cases \( G \ s, \ simp-all add:wp-loop-nguard hb \))
  - **from** \( hb \) **have** sound \( (wp \ do \ G \rightarrow od \ (\lambda s. 1) \ s) \)
  - **by** (auto intro! healthy-sound[OF healthy-wp-loop])
  - **thus** \( 0 \leq wp \ do \ G \rightarrow od \ (\lambda s. 1) \ s \) **by** (auto)
- **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** \( m :: 's \Rightarrow \text{nat} \) and \( \text{body} :: 's \) prog  
**assumes** \( wb :: \text{well-def body} \)
and guard: \( \forall s. m \ s = 0 \rightarrow \neg G \ s \)
and variant:  \( \forall n. \langle \lambda s. m s = \text{Suc } n \rangle \vdash \text{wp } \langle \lambda s. m s = n \rangle \)
says \( \lambda s. 1 \vdash \text{wp } \text{do } G \longrightarrow \text{body od } (\lambda s. 1) \)

proof –

note \( \text{hb} = \text{well-def-wp-healthy}[\text{OF } \text{wb}] \)

have \( \forall n. (\forall s. m s = n \longrightarrow 1 \leq \text{wp } \text{do } G \longrightarrow \text{body od } (\lambda s. 1) \ s) \)

proof\((\text{induct-tac } n)\)

fix \( n \)

show \( \forall s. m s = 0 \longrightarrow 1 \leq \text{wp } \text{do } G \longrightarrow \text{body od } (\lambda s. 1) \ s \)

proof\((\text{clarify})\)

fix \( s \)

assume \( m s = 0 \)

with \( \text{guard} \) have \( \neg G s \) by\( (\text{blast}) \)

with \( \text{hb} \) show \( 1 \leq \text{wp } \text{do } G \longrightarrow \text{body od } (\lambda s. 1) \ s \)

by\( (\text{simpl add:wp-loop-nguard}) \)

qed

assume \( IH; \forall s. m s = n \longrightarrow 1 \leq \text{wp } \text{do } G \longrightarrow \text{body od } (\lambda s. 1) \ s \)

hence \( IH'; \forall s. m s = n \longrightarrow 1 \leq \text{wp } \text{do } G \longrightarrow \text{body od } \langle \lambda s. \text{True} \rangle s \)

by\( (\text{simpl add:embed-bool-def}) \)

have \( \forall s. m s = \text{Suc } n \longrightarrow 1 \leq \text{wp } \text{do } G \longrightarrow \text{body od } \langle \lambda s. \text{True} \rangle s \)

proof\((\text{intro fold-premise healthy-intros hb, rule le-funI})\)

fix \( s \)

show \( \langle \lambda s. m s = \text{Suc } n \rangle \ s \leq \text{wp } \text{do } G \longrightarrow \text{body od } \langle \lambda s. \text{True} \rangle s \)

proof\((\text{cases } G \ s)\)

case \( \text{False} \)

hence \( 1 = \langle \text{N' } G \rangle \ s \) by\( (\text{auto}) \)

also from \( \text{wb} \) have \( \ldots \leq \text{wp } \text{do } G \longrightarrow \text{body od } (\lambda s. 1) \ s \)

by\( (\text{rule le-funD}[\text{OF } \text{term-onestep}]) \)

finally show \( \text{thesis} \) by\( (\text{simpl add:embed-bool-def}) \)

next

case \( \text{True} \) note \( G = \text{this} \)

from \( IH' \) have \( \langle \lambda s. m s = n \rangle \vdash \text{wp } \text{do } G \longrightarrow \text{body od } \langle \lambda s. \text{True} \rangle \)

by\( (\text{blast intro:use-premise healthy-intros hb}) \)

with \( \text{variant } \text{wb} \)

have \( \langle \lambda s. m s = \text{Suc } n \rangle \vdash (\text{body } \text{;; } \text{do } G \longrightarrow \text{body od}) \langle \lambda s. \text{True} \rangle \)

by\( (\text{blast intro:wp-Seq ad-intros}) \)

hence \( \langle \lambda s. m s = \text{Suc } n \rangle \ s \leq (\text{body } \text{;; } \text{do } G \longrightarrow \text{body od}) \langle \lambda s. \text{True} \rangle s \)

by\( (\text{auto}) \)

also from \( \text{hb} \ G \) have \( \ldots = \text{wp } \text{do } G \longrightarrow \text{body od } \langle \lambda s. \text{True} \rangle s \)

by\( (\text{simpl add:wp-loop-guard}) \)

finally show \( \text{thesis} \) .

qed

qed

thus \( \forall s. m s = \text{Suc } n \longrightarrow 1 \leq \text{wp } \text{do } G \longrightarrow \text{body od } (\lambda s. 1) \ s \)

by\( (\text{simpl add:embed-bool-def}) \)

qed

thus \( \text{thesis} \) by\( (\text{auto}) \)

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: \( \forall s. m s = 0 \longrightarrow \neg G s \)
and variant: \( \forall n. (\lambda s. m s = \text{Suc } n) \&\& \langle I \rangle \vdash \text{wp } body \ (\lambda s. m s = n) \)
and inv: \( \text{wp-inv } G \text{ body } \langle I \rangle \)
shows \( \langle I \rangle \vdash \text{wp do } G \longrightarrow \text{body od } (\lambda s. \text{True}) \)

**proof**

- **note** hb = well-def-wp-healthy[OF wb]
- **note** scb = sublinear-sub-conj[OF well-def-wp-sublinear, OF wb]
- **have** \( \langle I \rangle \vdash \text{wp do } G \longrightarrow \text{body od } (\lambda s. \text{True}) \)
- **proof** (rule use-premise, intro healthy-intros hb)
  - **fix** s
  - **have** \( \forall n. (\forall s. m s = n \&\& I s \longrightarrow 1 \leq \text{wp do } G \longrightarrow \text{body od } (\lambda s. \text{True}) \ s) \)
  - **proof** (induct-tac n)
    - **fix** n
    - **show** \( \forall s. m s = 0 \&\& I s \longrightarrow 1 \leq \text{wp do } G \longrightarrow \text{body od } (\lambda s. \text{True}) \ s \)
    - **proof** (clarify)
      - **assume** m s = 0
      - **with** guard **have** \( \neg G s \) **by**(blast)
      - **with** hb **show** \( I \leq \text{wp do } G \longrightarrow \text{body od } (\lambda s. \text{True}) \ s \)
        - **by**(simp add:wp-loop-nguard)
    - qed
    - **assume** IH: \( \forall s. m s = n \&\& I s \longrightarrow 1 \leq \text{wp do } G \longrightarrow \text{body od } (\lambda s. \text{True}) \ s \)
    - **show** \( \forall s. m s = \text{Suc } n \&\& I s \longrightarrow 1 \leq \text{wp do } G \longrightarrow \text{body od } (\lambda s. \text{True}) \ s \)
      - **proof** (intro fold-premise healthy-intros hb le-funI)
        - **fix** s
        - **show** \( (\lambda s. m s = \text{Suc } n \&\& I s) \leq \text{wp do } G \longrightarrow \text{body od } (\lambda s. \text{True}) \ s \)
          - **proof** (cases G s)
            - **case** False with hb **show** \( \neg \text{thesis} \)
              - **by**(simp add:wp-loop-nguard)
            - **next**
              - **case** True **note** G = this
                - **have** \( (\lambda s. m s = \text{Suc } n) \&\& \langle I \rangle \&\& \langle G \rangle = (\lambda s. m s = \text{Suc } n) \&\& (\langle I \rangle \&\& (\langle G \rangle) \&\& \langle I \rangle) \&\& \langle G \rangle \)
                  - **by**(simp)
              - also **have** \( \... = (\lambda s. m s = \text{Suc } n) \&\& \langle I \rangle \&\& (\langle G \rangle \&\& \langle I \rangle) \)
                  - **by**(simp add:exp-conj-assoc exp-conj-unitary del:exp-conj-idem)
                - also **have** \( \... = (\lambda s. m s = \text{Suc } n) \&\& (\langle G \rangle \&\& \langle I \rangle) \)
                  - **by**(simp only:exp-conj-comm)
              - also \{ from inv hb **have** \( \langle G \rangle \&\& \langle I \rangle \vdash \text{wp body } \langle I \rangle \)
                  - **by**(rule wp-inv-stdD)
                - with variant
                  - **have** \( (\lambda s. m s = \text{Suc } n) \&\& \langle I \rangle \&\& (\langle G \rangle \&\& \langle I \rangle) \vdash \text{wp body } (\lambda s. m s = n) \&\& \text{wp body } \langle I \rangle \)
                    - **by**(rule entails-frame)
4.11. LOOP TERMINATION

also from scb
have \( \text{wp body } \langle \lambda s. m \ s = n \rangle \land \text{wp body } \langle I \rangle \vdash \text{wp body } (\langle \lambda s. m \ s = n \rangle \land \langle I \rangle) \)
by (blast)
finally have \( \langle \lambda s. m \ s = \text{Suc } n \rangle \land \langle I \rangle \vdash \text{wp body } (\langle \lambda s. m \ s = n \rangle \land \langle I \rangle) . \)
moreover {
from IH have \( \langle \lambda s. m \ s = n \rangle \land \langle I \rangle \vdash \text{wp do } G \rightarrow \text{body od } \langle \lambda s. \text{True} \rangle \)
by (blast intro:use-premise healthy-intros hb)
hence \( \langle \lambda s. m \ s = n \rangle \land \langle I \rangle \vdash \text{wp do } G \rightarrow \text{body od } \langle \lambda s. \text{True} \rangle \)
by (simp add:exp-conj-std-split)
}
ultimately
have \( \langle \lambda s. m \ s = \text{Suc } n \rangle \land \langle I \rangle \vdash \text{wp body } (\langle \lambda s. m \ s = n \rangle \land \langle I \rangle) \rightarrow \text{wp do } G \rightarrow \text{body od } \langle \lambda s. \text{True} \rangle \)
using \( \text{wb by (blast intro:wp-Seq wd-intros)} \)
hence \( \langle \lambda s. m \ s = n \rangle \land \langle I \rangle \vdash \text{wp body } (\langle \lambda s. m \ s = n \rangle \land \langle I \rangle) \rightarrow \text{wp do } G \rightarrow \text{body od } \langle \lambda s. \text{True} \rangle \)
by (auto simp:exp-conj-def)
also from \( \text{hb G have ... = wp do } G \rightarrow \text{body od } \langle \lambda s. \text{True} \rangle \)
by (simp add:wp-loop-guard)
finally show ?thesis .
qed

moreover assume \( I \) \( s \)
ultimately show \( 1 \leq \text{wp do } G \rightarrow \text{body od } \langle \lambda s. \text{True} \rangle \)
by (auto)
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: \( \text{well-def} \) body
— The loop terminates in one step with nonzero probability
and onestep: \( \lambda s. p \vdash \text{wp body } \langle \text{N } G \rangle \)
and nzp: \( 0 < p \)
— The body is maximal i.e. it terminates absolutely.
and mb: \( \text{maximal (wp body)} \)
shows \( \lambda s. 1 \vdash \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) \)
proof –

**note** \( hh = \text{well-def-wp-healthy} [\text{OF wb}] \)

**note** \( sh = \text{healthy-scalingD} [\text{OF hb}] \)

**note** \( sab = \text{sublinear-subadd} [\text{OF well-def-wp-sublinear}, \text{OF wb}, \text{OF healthy-feasibleD}, \text{OF hb}] \)

from \( hh \) have \( \text{hloop: healthy (wp do } G \rightarrow \text{body od)} \)

by (rule healthy-intros)

**hence** \( \text{swp: sound (wp do } G \rightarrow \text{body od (} \lambda s. 1)) \) by (blast)

\( p \) is no greater than 1, by feasibility.

from onestep have \( \lambda s. p \leq \text{wp body } \langle N G \rangle s \) by (auto)

also {
  from \( hh \) have \( \text{unitary (wp body } \langle N G \rangle) \) by (auto)
  **hence** \( \lambda s. \text{wp body } \langle N G \rangle s \leq 1 \) by (auto)
}

finally have \( \text{p1: } p \leq 1 . \)

This is the crux of the proof: that given a lower bound below 1, we can find another, higher one.

**have** new-bound: \( \lambda k. 0 \leq k \implies k \leq 1 \implies (\lambda s. k) \vdash \text{wp do } G \rightarrow \text{body od (} \lambda s. 1) \) \( \implies (\lambda s. p * (1 - k) + k) \vdash \text{wp do } G \rightarrow \text{body od (} \lambda s. 1) \)

**proof** (rule le-funI)

fix \( k \) \( s \)

**assume** \( X; \lambda s. k \vdash \text{wp do } G \rightarrow \text{body od (} \lambda s. 1) \)

and \( k0: 0 \leq k \) and \( k1: k \leq 1 \)

from \( k1 \) have \( \text{nz1k: } 0 \leq 1 - k \) 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** \( p * (1 - k) + k \leq \langle N G \rangle s + \langle G \rangle s * (p * (1 - k) + k) \)

by (cases G s, simp-all)

By the one-step termination assumption:

also from onestep \( \text{nz1k} \)

**have** \( ... \leq \langle N G \rangle s + \langle G \rangle s * (wp body \langle N G \rangle s * (1 - k) + k) \)

by (auto intro:add-left-mono add-right-mono mult-left-mono mult-right-mono)

By scaling:

also from \( \text{nz1k} \)

**have** \( ... = \langle N G \rangle s + \langle G \rangle s * (wp body (\lambda s. \langle N G \rangle s * (1 - k)) s + k) \)

by (simp add:right-scalingD [OF \( sb \)])

By the maximality (termination) of the loop body:
Lastly, by folding two loop iterations:

\[
\text{also from } \text{mb } k\theta \\
\text{have } \ldots = \langle N \ G \rangle s + \langle G \rangle s * (\text{wp body } (\lambda s. \langle N \ G \rangle s * (1 - k)) s + \text{wp body } (\lambda s. k) s) \\
\text{by}(\text{simp add:maximalD})
\]

By sub-additivity of the loop body:

\[
\text{also from } k\theta \ n \rightarrow 1k \\
\text{have } \ldots \leq \langle N \ G \rangle s + \langle G \rangle s * (\text{wp body } (\lambda s. \langle N \ G \rangle s * (1 - k) + k) s) \\
\text{by}(\text{auto intro: add-left-mono multi-left-mono sub-addD[OF sub] sound-intros}) \\
\text{also} \\
\text{have } \ldots = \langle N \ G \rangle s + \langle G \rangle s * (\text{wp body } (\lambda s. \langle N \ G \rangle s + \langle G \rangle s * k) s) \\
\text{by}(\text{simp add: negate-embed algebra-simps})
\]

By monotonicity of the loop body, and that \( k \) is a lower bound:

\[
\text{also from } k\theta \ bloop \ le-funD[OF X] \\
\text{have } \ldots \leq \langle N \ G \rangle s + \langle G \rangle s * (\text{wp body } (\lambda s. \langle N \ G \rangle s + \langle G \rangle s * \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s) s) \\
\text{by}(\text{iprover intro: add-left-mono multi-left-mono le-funI embed-ge-0} \\
\text{le-funD[OF mono-transD, OF healthy-monoD, OF hb]} \\
\text{sound-sum standard-sound sound-intros swp})
\]

Unrolling the loop once and simplifying:

\[
\text{also } \\
\text{have } \bigwedge s. \langle N \ G \rangle s + \langle G \rangle s * \text{wp body } (\text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s) s = \\
\langle N \ G \rangle s + \langle G \rangle s * (\langle N \ G \rangle s + \langle G \rangle s * \text{wp body } (\text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s) s) \\
(\lambda s. 1) s) \\
\text{by}(\text{simp only: distrib-left mult. assoc[symmetric] embed-bool-idem embed-bool-cancel}) \\
\text{also have } \bigwedge s. \ldots s = \langle N \ G \rangle s + \langle G \rangle s * \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s \\
\text{by}(\text{simp add: fun-cong[OF wp-loop-unfold[symmetric, where } P=\lambda s. 1, simplified, OF hb]]) \\
\text{finally have } X: \bigwedge s. \langle N \ G \rangle s + \langle G \rangle s * \text{wp body } (\text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s) s = \\
\langle N \ G \rangle s + \langle G \rangle s * \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s \\
\text{have } \langle N \ G \rangle s + \langle G \rangle s * (\text{wp body } (\lambda s. \langle N \ G \rangle s + \langle G \rangle s * \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s) s) = \\
(\lambda s. 1) s) s) \\
\text{by}(\text{simp only:X})
\]

Lastly, by folding two loop iterations:

\[
\text{also } \\
\text{have } \langle N \ G \rangle s + \langle G \rangle s * (\text{wp body } (\lambda s. \langle N \ G \rangle s + \langle G \rangle s * \text{wp body } (\text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s) s) s) = \\
\text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s \\
\text{by}(\text{simp add: wp-loop-unfold[OF hb, where } P=\lambda s. 1, simplified, symmetric]} \\
\text{fun-cong[OF wp-loop-unfold[OF hb, where } P=\lambda s. 1, simplified, symmetric]]
\]
finally show  \( p \ast (1-k) + k \leq \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s \).

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.

\[
\text{from } nzp \ \text{have inc: } \forall k. \ 0 \leq k \Rightarrow k < 1 \Rightarrow k < p \ast (1-k) + k
\]

\by(auto intro:mult-pos-pos)

The result follows by contradiction.

\begin{verbatim}
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 \( \neg (\forall s. \ 1 \leq \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s) \) \by(auto)
  then obtain \( s \) where \( \text{point: } \neg 1 \leq \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s \) \by(auto)

  let \( \forall k = \text{Inf } (\text{range } (\text{wp do } G \rightarrow \text{body od } (\lambda s. 1))) \)

  from hloop
  have Inflb: \( \forall s. \ \forall k \leq \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s \)
    \by(intro cInf-lower bdd-belowI, auto)
  also from point
  have wp do G \rightarrow \text{body od } (\lambda s. 1) s < 1 \by(auto)

  thus the least (infimum) probability of termination is strictly less than one.

  finally have \( k_1: \ ?k < 1 \).
  hence \( \forall k \leq 1 \) \by(auto)
  moreover from hloop
  have \( k_1: \ ?\forall k \leq 1 \) \by(intro cInf-greatest, auto)

  the infimum is, naturally, a lower bound.

  moreover from Inflb
  have \( (\lambda s. \ ?k) \vdash \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) \) \by(auto)

  ultimately

  we can therefore use the previous result to find a new bound, \ldots

  have \( \forall s. \ p \ast (1 - ?k) + ?k \leq \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s \)
    \by(blast intro:le-funD[OF new-bound])

  \ldots which is lower than the infimum, by minimality, \ldots

  hence \( p \ast (1 - ?k) + ?k \leq ?k \)
    \by(blast intro:cInf-greatest)

  \ldots yet also strictly greater than it.

  moreover from \( k_0 \ k_1 \)
  have \( ?k < p \ast (1 - ?k) + ?k \) \by(rule inc)

  we thus have a contradiction.

  ultimately show False \by(simp)
\end{verbatim}
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 preexpection 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
4.13 Miscellaneous Mathematics

theory Misc imports Real Multivariate-Analysis begin

lemma setsum-UNIV:
fixes S :: 'a::finite set
assumes complete: \( \forall x. x \notin S \implies f x = 0 \)
shows \( \text{setsum } f S = \text{setsum } f \text{ UNIV} \)

proof -
from complete have \( \text{setsum } f S = \text{setsum } f \text{ (UNIV } - S) + \text{setsum } f S \) by(simp)
also have \( ... = \text{setsum } f \text{ UNIV} \) by(auto intro: setsum.subset-diff[symmetric])
finally show \( \text{thesis} \).

qed

lemma cInf-mono:
fixes A :: 'a::conditionally-complete-lattice set
assumes lower: \( \forall b. b \in B \implies \exists a \in A. a \leq b \)
and bounded: \( \forall a. a \in A \implies c \leq a \)
and ne: \( B \neq \{\} \)
shows \( \text{Inf } A \leq \text{Inf } B \)

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 \( \text{Inf } A \leq a \) by(intro cInf-lower bdd-belowI, auto)
also note le
finally show \( \text{Inf } A \leq b \).

qed

lemma max-distrib:
fixes c::real
assumes nn: \( 0 \leq c \)
shows \( c \cdot \max a b = \max (c \cdot a) (c \cdot b) \)

proof(cases a \leq b)
case True
moreover with nn have \( c \cdot a \leq c \cdot b \) by(auto intro: mult-left-mono)
ultimately show \( \text{thesis} \) by(simp add: max.absorb2)
next
case False then have \( b \leq a \) by(auto)
moreover with nn have \( c \cdot b \leq c \cdot a \) by(auto intro: mult-left-mono)

219
ultimately show \(?thesis\) by(simp add:max.absorb1)

qed

lemma mult-div-mono-left:
  fixes \(c::\text{real}\)
  assumes nnc: \(0 \leq c\) and nzc: \(c \neq 0\)
  and \(\text{inv}::\text{inverse } c * a \leq b\)
  shows \(c * a \leq b\)
  proof --
  from nnc inv have \(c * a \leq (c * \text{inverse } c) * b\)
  by(auto simp:mult.assoc intro:mult-left-mono)
  also from nzc have ... = \(b\) by(simp)
  finally show \(c * a \leq b\).

qed

lemma mult-div-mono-right:
  fixes \(c::\text{real}\)
  assumes nnc: \(0 \leq c\) and nzc: \(c \neq 0\)
  and \(\text{inv}::\text{inverse } c * a \leq b\)
  shows \(a \leq c * b\)
  proof --
  from nzc have \(a = (c * \text{inverse } c) * a\) by(simp)
  also from nnc inv have \((c * \text{inverse } c) * a \leq c * b\)
  by(auto simp:mult.assoc intro:mult-left-mono)
  finally show \(a \leq c * b\).

qed

lemma min-distrib:
  fixes \(c::\text{real}\)
  assumes nnc: \(0 \leq c\)
  shows \(c * \min a b = \min (c * a) (c * b)\)
  proof(cases \(a \leq b\))
  case True moreover with nnc have \(c * a \leq c * b\)
  by(blast intro:mult-left-mono)
  ultimately show \(?thesis\) by(auto)
  next
  case False hence \(b \leq a\) by(auto)
  moreover with nnc have \(c * b \leq c * a\)
  by(blast intro:mult-left-mono)
  ultimately show \(?thesis\) by(simp add:min.absorb2)

qed

lemma nonempty-witness:
  \(S \neq \{\} \implies \exists x. x \in S\)
  by(blast)

lemma finite-set-least:
  fixes \(S::'a::\text{linorder set}\)
  assumes finite: \(\text{finite } S\)
and \( ne: S \neq \{ \} \)
shows \( \exists x \in S. \forall y \in S. x \leq y \)
proof
have \( S = \{ \} \lor (\exists x \in S. \forall y \in S. x \leq y) \)
proof (rule finite-induct, simp-all add:assms)
fix \( x::a \) and \( S::a \) set
assume \( IH: S = \{ \} \lor (\exists x \in S. \forall y \in S. x \leq y) \)
show \( (\forall y \in S. x \leq y) \lor (\exists x' \in S. x' \leq x \land (\forall y \in S. x' \leq y)) \)
proof (cases \( S = \{ \} \))

case \( \text{True} \) then show \( \text{thesis} \) by (auto)

next
case \( \text{False with } IH \) have \( \exists x \in S. \forall y \in S. x \leq y \) by (auto)
then obtain \( z \) where \( \text{zin: } z \in S \land \text{zmin: } \forall y \in S. z \leq y \) by (auto)
thus \( \text{thesis} \) by (cases \( z \leq x, \text{auto} \))
qed
qed

with \( ne \) show \( \text{thesis} \) by (auto)
qed
lemma \( c\text{Sup-add} \):
fixes \( c::\text{real} \)
assumes \( ne: S \neq \{ \} \)
and \( bS: \forall x. x \in S \rightarrow x \leq b \)
shows \( \text{Sup } S + c = \text{Sup } \{ x + c | x \in S \} \)
proof (rule antisym)
from \( ne \) \( bS \) show \( \text{Sup } \{ x + c | x \in S \} \leq \text{Sup } S + c \)
by (auto intro: cSup-least add-right-mono cSup-upper bdd-aboveI)

have \( \text{Sup } S \leq \text{Sup } \{ x + c | x \in S \} - c \)
proof (intro cSup-least \( ne \))
fix \( x \) assume \( \text{zin: } x \in S \)
from \( bS \) have \( \forall x. x \in S \rightarrow x + c \leq b + c \) by (auto intro: add-right-mono)

hence \( \text{bdd-aboveI } \{ x + c | x \in S \} \) by (intro bdd-aboveI, blast)
with \( \text{zin} \) have \( x + c \leq \text{Sup } \{ x + c | x \in S \} \) by (auto intro: cSup-upper)
thus \( x \leq \text{Sup } \{ x + c | x \in S \} - c \) by (auto)
qed
thus \( \text{Sup } S + c \leq \text{Sup } \{ x + c | x \in S \} \) by (auto)
qed

lemma \( c\text{Sup-mul} \):
fixes \( c::\text{real} \)
assumes \( ne: S \neq \{ \} \)
and \( bS: \forall x. x \in S \rightarrow x \leq b \)
and \( nnc: 0 \leq c \)
shows \( c \cdot \text{Sup } S = \text{Sup } \{ c \cdot x | x \in S \} \)
proof (cases)
assume \( c = 0 \)
moreover from \( ne \) have \( \exists x. x \in S \) by (auto)
ultimately show \( \text{thesis} \) by (simp)
next
assume cnz: \( \ c \neq 0 \)
show \(?thesis
proof (rule antisym)
from \( bS \) have \( baS \): bdd-above \( S \) by (intro bdd-aboveI, auto)
with \( ne \) nnc show \( \sup \ \{ c \cdot x \mid x \in S \} \leq c \cdot \sup \ S \)
by (blast intro: cSup-least mult-left-mono [OF cSup-upper])
have \( \sup S \leq \inverse c \cdot \sup \ \{ c \cdot x \mid x \in S \} \)
proof (intro cSup-least ne)
fix \( x \) assume \( \xin: x \in S \)
moreover from \( bS \) nnc have \( \forall x \cdot x \in S \implies c \cdot x \leq c \cdot b \) by (auto intro: mult-left-mono)
ultimately have \( c \cdot x \leq \sup \ \{ c \cdot x \mid x \in S \} \)
by (auto intro: cSup-upper bdd-aboveI)
moreover from \( nnc \) have \( 0 \leq \inverse c \) by (auto)
ultimately have \( \inverse c \cdot (c \cdot x) \leq \inverse c \cdot \sup \ \{ c \cdot x \mid x \in S \} \)
by (simp add: mult. assoc [symmetric])
qed
with \( nnc \) have \( c \cdot \sup S \leq \inverse c \cdot \sup \ \{ c \cdot x \mid x \in S \} \)
by (auto intro: mult-left-mono)
with \( cnz \) show \( c \cdot \sup S \leq \sup \ \{ c \cdot x \mid x \in S \} \)
by (simp add: mult. assoc [symmetric])
qed

lemma closure-contains-Sup:
fixes \( S :: \text{real set} \)
assumes \( neS: S \neq \{} \) and \( bS: \forall x \in S. x \leq B \)
shows \( \sup S \in \text{closure} S \)
proof
let \( ?T = \uminus ' S \)
from \( neS \) have \( \neT: ?T \neq \{} \) by (auto)
from \( bS \) have \( bT: \forall x \in ?T. -B \leq x \) by (auto)
thus \( bdd-below ?T \leq \inf \uminus ' S \) by (intro bdd-belowI, blast)

have \( \sup S = - \inf ?T \)
proof (rule antisym)
from \( neT \) \( bbT \)
have \( \\forall x. x \in S \implies \inf (\uminus ' S) \leq -x \)
by (blast intro: cInf-lower)
hence \( \\forall x. x \in S \implies -1 \cdot -x \leq -1 \cdot \inf (\uminus ' S) \)
by (rule mult-left-mono-neg, auto)
thus \( \\inf (\uminus ' S) \leq x \leq \inf (\uminus ' S) \)
by (simp)
with \( neS \) \( bS \) show \( \sup S \leq - \inf ?T \)
by (blast intro: cSup-least)
have \( \text{Sup } S \leq \text{Inf } ?T \)

proof (rule cInf-greatest[OF neT])

fix \( x \)
assume \( x \in \uminus ' S \)
then obtain \( y \) where \( \text{yin: } y \in S \) and \( \text{rwx: } x = -y \)(auto)

from \( \text{yin bS have } y \leq \text{Sup } S \)
by (intro cSup-upper bdd-belowI, auto)

hence \(-1 \times \text{Sup } S \leq -1 \times y \)
by (simp add: mult-left-mono-neg)

with \( \text{rwx show } - \text{Sup } S \leq x \) by (simp)

qed

hence \(-1 \times \text{Inf } ?T \leq -1 \times (- \text{Sup } S) \)
by (simp add: mult-left-mono-neg)

thus \(- \text{Inf } ?T \leq \text{Sup } S \) by (simp)

qed

also 

thm \( bT \)
from \( \text{neT bbT have } \text{Inf } ?T \in \text{closure } ?T \) by (rule closure-contains-Inf)

hence \(- \text{Inf } ?T \in \uminus ' \text{closure } ?T \) by (auto) 

}\}{

also 

have linear \( \uminus \) by (auto intro: linearI)

hence \( \uminus ' \text{closure } ?T \subseteq \text{closure } (\uminus ' ?T) \)
by (rule closure-linear-image) 

}\}{

also 

have \( \uminus ' ?T \subseteq S \) by (auto)

hence \( \text{closure } (\uminus ' ?T) \subseteq \text{closure } S \) by (rule closure-mono) 

}\}

finally show \( \text{Sup } S \in \text{closure } S \)

qed

lemma tendsto-min:
fixes \( x \ y :: \text{real} \)
assumes \( \text{ta: } a \dashv \dashv > x \)
and \( \text{tb: } b \dashv \dashv > y \)
shows \( (\lambda i. \text{min } (a \ i) \ (b \ i)) \dashv \dashv > \text{min } x \ y \)

proof (rule LIMSEQ-I, simp)

fix \( c :: \text{real} \) assume \( \text{pc: } 0 < e \)

from \( \text{ta pe obtain noa where balla: } \forall n \geq \text{noa}. \text{ abs } (a \ n - x) < e \)
by (auto dest: LIMSEQ-D)

from \( \text{tb pe obtain nob where ballb: } \forall n \geq \text{nob}. \text{ abs } (b \ n - y) < e \)
by (auto dest: LIMSEQ-D)

\{ 

fix \( n \)
assume \( \text{ge: max noa nob \leq n} \)

hence \( \text{gea: noa \leq n and geb: nob \leq n} \) by (auto)

have \( \text{abs } (\text{min } (a \ n) \ (b \ n) - \text{min } x \ y) < e \)
proof cases
  assume le: \( \text{lcm} (a \ n) (b \ n) \leq \text{lcm} x y \)
  show \( \exists \text{thesis} \)
proof cases
  assume \( a \ n \leq b \ n \)
  hence \( \text{rwmin} : \text{lcm} (a \ n) (b \ n) = a \ n \) by(auto)
  with \( \text{le} \) have \( a \ n \leq \text{lcm} x y \) by(simp)
  moreover from \( \text{gea balla} \) have \( \text{abs} (a \ n - x) < e \) by(simp)
  moreover have \( \text{lcm} x y \leq x \) by(auto)
  ultimately have \( \text{abs} (a \ n - \text{lcm} x y) < e \) by(auto)
  with \( \text{rwmin} \) show \( \text{abs} (\text{lcm} (a \ n) (b \ n) - \text{lcm} x y) < e \) by(simp)
next
  assume \( \neg a \ n \leq b \ n \)
  hence \( b \ n \leq a \ n \) by(auto)
  hence \( \text{rwmin} : \text{lcm} (a \ n) (b \ n) = b \ n \) by(auto)
  with \( \text{le} \) have \( b \ n \leq \text{lcm} x y \) by(simp)
  moreover from \( \text{geb ballb} \) have \( \text{abs} (b \ n - y) < e \) by(simp)
  moreover have \( \text{lcm} x y \leq y \) by(auto)
  ultimately have \( \text{abs} (b \ n - \text{lcm} x y) < e \) by(auto)
  with \( \text{rwmin} \) show \( \text{abs} (\text{lcm} (a \ n) (b \ n) - \text{lcm} x y) < e \) by(simp)
qed

next
  assume \( \neg \text{lcm} (a \ n) (b \ n) \leq \text{lcm} x y \)
  hence \( \text{le} : \text{lcm} x y \leq \text{lcm} (a \ n) (b \ n) \) by(auto)
  show \( \exists \text{thesis} \)
proof cases
  assume \( x \leq y \)
  hence \( \text{rwmin} : \text{lcm} x y = x \) by(auto)
  with \( \text{le} \) have \( x \leq \text{lcm} (a \ n) (b \ n) \) by(simp)
  moreover from \( \text{gea balla} \) have \( \text{abs} (a \ n - x) < e \) by(simp)
  moreover have \( \text{lcm} (a \ n) (b \ n) \leq a \ n \) by(auto)
  ultimately have \( \text{abs} (\text{lcm} (a \ n) (b \ n) - x) < e \) by(auto)
  with \( \text{rwmin} \) show \( \text{abs} (\text{lcm} (a \ n) (b \ n) - x) < e \) by(simp)
next
  assume \( \neg x \leq y \)
  hence \( y \leq x \) by(auto)
  hence \( \text{rwmin} : \text{lcm} x y = y \) by(auto)
  with \( \text{le} \) have \( y \leq \text{lcm} (a \ n) (b \ n) \) by(simp)
  moreover from \( \text{geb ballb} \) have \( \text{abs} (b \ n - y) < e \) by(simp)
  moreover have \( \text{lcm} (a \ n) (b \ n) \leq b \ n \) by(auto)
  ultimately have \( \text{abs} (\text{lcm} (a \ n) (b \ n) - y) < e \) by(auto)
  with \( \text{rwmin} \) show \( \text{abs} (\text{lcm} (a \ n) (b \ n) - y) < e \) by(simp)
qed

qed

\}\n
thus \( \exists \text{no.} \ \forall n \geq \text{no.} \ |\text{lcm} (a \ n) (b \ n) - \text{lcm} x y| < e \) by(blast)
qed

definition supp :: \((\text{'s} \Rightarrow \text{real}) \Rightarrow \text{'s set} \)
where \( \text{supp } f = \{ x. f x \neq 0 \} \)

**Definition** \( \text{dist-remove} :: (\cdot \Rightarrow \text{real}) \Rightarrow \cdot \Rightarrow \cdot \Rightarrow \text{real} \)

where \( \text{dist-remove } p x = (\lambda y. \text{if } y=x \text{ then } 0 \text{ else } p y / (1 - p x)) \)

**Lemma** \( \text{supp-dist-remove} \): 
\( p x \neq 0 \Rightarrow p x \neq 1 \Rightarrow \text{supp } (\text{dist-remove } p x) = \text{supp } p \setminus \{ x \} \)
by(auto simp:dist-remove-def supp-def)

**Lemma** \( \text{supp-empty} \):
\( \text{supp } f = \{ \} = \Rightarrow f x = 0 \)
by(simp add:supp-def)

**Lemma** \( \text{nsupp-zero} \):
\( x \notin \text{supp } f = \Rightarrow f x = 0 \)
by(simp add:supp-def)

**Lemma** \( \text{setsum-supp} \):
fixes \( f :: \cdot \Rightarrow \text{finite} \Rightarrow \text{real} \)
shows \( \text{setsum } f \ (\text{supp } f) = \text{setsum } f \ \text{UNIV} \)
**Proof**
- have \( \text{setsum } f \ (\text{UNIV} - \text{supp } f) = 0 \)
  by(simp add:supp-def)
- hence \( \text{setsum } f \ (\text{supp } f) = \text{setsum } f \ (\text{UNIV} - \text{supp } f) + \text{setsum } f \ (\text{supp } f) \)
  by(simp)
- also have \( \ldots = \text{setsum } f \ \text{UNIV} \)
  by(simp add:setsum.subset-diff[symmetric])
finally show \( \text{thesis} \).
**qed**

### 4.13.1 Truncated Subtraction

**Definition**
\( \text{tminus} :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real} \ \text{(infixl } \oplus 60) \)
where
\( x \oplus y = \text{max } (x - y) \ 0 \)

**Lemma** \( \text{minus-le-tminus}[intro,simp] \):
\( a - b \leq a \oplus b \)
**Unfolding** \( \text{tminus-def by(auto)} \)

**Lemma** \( \text{tminus-cancel-1} \):
\( 0 \leq a \Rightarrow a + 1 \oplus 1 = a \)
**Unfolding** \( \text{tminus-def by(simp)} \)

**Lemma** \( \text{tminus-zero-imp-le} \):
\( x \oplus y \leq 0 \Rightarrow x \leq y \)
**by(simp add:tminus-def)**
lemma tminus-zero[simp]:
\[0 \leq x \implies x \ominus 0 = x\]
by(simp add:tminus-def)

lemma tminus-left-mono:
\[a \leq b \implies a \ominus c \leq b \ominus c\]
unfolding tminus-def
by(case-tac a \leq c, simp-all)

lemma tminus-less:
\[[0 \leq a; 0 \leq b] \implies a \ominus b \leq a\]
unfolding tminus-def by(force)

lemma tminus-left-distrib:
assumes nna: \[0 \leq a\]
shows \[a \ast (b \ominus c) = a \ast b \ominus a \ast c\]
proof(cases b \leq c)
case True note le = this
hence \[a \ast \max(b - c) 0 = 0\] by(simp add:max.absorb2)
also {
  from nna le have \[a \ast b \leq a \ast c\] by(blast intro:mult-left-mono)
  hence \[0 = \max(a \ast b - a \ast c) 0\] by(simp add:max.absorb1)
}
finally show \?thesis by(simp add:tminus-def)
next
case False hence le: \[c \leq b\] by(auto)
hence \[a \ast \max(b - c) 0 = 0\] by(simp only:max.absorb1)
also {
  from nna le have \[a \ast c \leq a \ast b\] by(blast intro:mult-left-mono)
  hence \[a \ast (b - c) = \max(a \ast b - a \ast c) 0\] by(simp add:max.absorb1
field-simps)
}
finally show \?thesis by(simp add:tminus-def)
qed

lemma tminus-le[simp]:
\[b \leq a \implies a \ominus b = a - b\]
unfolding tminus-def by(simp)

lemma tminus-le-alt[simp]:
\[a \leq b \implies a \ominus b = 0\]
by(simp add:tminus-def)

lemma tminus-nle[simp]:
\[\neg b \leq a \implies a \ominus b = 0\]
unfolding tminus-def by(simp)

lemma tminus-add-mono:
\[(a+b) \ominus (c+d) \leq (a\ominus c) + (b\ominus d)\]
proof\((\text{cases } 0 \leq a - c)\)
  case True note \(pac = \text{this}\)
  show \(?\text{thesis}\)
  proof\((\text{cases } 0 \leq b - d)\)
    case True note \(pbd = \text{this}\)
    from \(pac\) and \(pbd\) have \((c + d) \leq (a + b)\) by\((\text{simp})\)
    with \(pac\) and \(pbd\) show \(?\text{thesis}\) by\((\text{simp})\)
  next
    case False with \(pac\) show \(?\text{thesis}\)
      by\((\text{cases } c + d \leq a + b, \text{auto})\)
  qed
next
  case False note \(nac = \text{this}\)
  show \(?\text{thesis}\)
  proof\((\text{cases } 0 \leq b - d)\)
    case True with \(nac\) show \(?\text{thesis}\)
      by\((\text{cases } c + d \leq a + b, \text{auto})\)
  next
    case False note \(nbd = \text{this}\)
    with \(nac\) have \(\neg(c + d) \leq (a + b)\) by\((\text{simp})\)
    with \(nac\) and \(nbd\) show \(?\text{thesis}\) by\((\text{simp})\)
  qed
qed

lemma \(\text{tminus-setsum-mono}:\)
  assumes \(fS: \text{finite } S\)
  shows \(\text{setsum } f \, S \ominus \text{setsum } g \, S \leq \text{setsum } (\lambda x. f \, x \ominus g \, x) \, S\)
    (is \(?X \, S)\)
proof\((\text{rule finite-induct})\)
  from \(fS\) show \(\text{finite } S\) .

  show \(?X \, \{\}\) by\((\text{simp})\)

  fix \(x\) and \(F\)
  assume \(fF: \text{finite } F\) and \(xniF: x \notin F\)
  and \(IH: \, ?X \, F\)
  have \(f \, x + \text{setsum } f \, F \ominus g \, x + \text{setsum } g \, F \leq\)
    \((f \, x \ominus g \, x) + (\text{setsum } f \, F \ominus \text{setsum } g \, F)\)
  by\((\text{rule tminus-add-mono})\)
  also from \(IH\) have \(\ldots \leq (f \, x \ominus g \, x) + (\text{sum } x \in F. f \, x \ominus g \, x)\)
    by\((\text{rule add-left-mono})\)
  finally show \(?X \, (\text{insert } x \, F)\)
    by\((\text{simp \, add: setsum.insert[OF } fF \, xniF])\)
qed

lemma \(\text{tminus-nneg}[\text{simp, intro}]:\)
  \(\theta \leq a \ominus b\)
  by\((\text{cases } b \leq a, \text{auto})\)
lemma tminus-right-antimono:
  assumes clb: c ≤ b
  shows a ⊕ b ≤ a ⊕ c
proof\(\text{cases } b ≤ a\)
  case True
    moreover with clb have c ≤ a by (auto)
  moreover note clb
  ultimately show \(?\text{thesis by (simp)}\)
next
  case False then show \(?\text{thesis by (simp)}\)
qed

lemma min-tminus-distrib:
  min a b ⊕ c = min (a ⊕ c) (b ⊕ c)
unfolding tminus-def by (auto)
Bibliography


