pGCL for Isabelle

David Cock

April 20, 2020
# Contents

1 Overview

2 Introduction to pGCL
   2.1 Language Primitives
      2.1.1 The Basics
      2.1.2 Assertion and Annotation
      2.1.3 Probability
      2.1.4 Nondeterminism
      2.1.5 Properties of Expectations
   2.2 Loops
      2.2.1 Guaranteed Termination
      2.2.2 Probabilistic Termination
   2.3 The Monty Hall Problem
      2.3.1 The State Space
      2.3.2 The Game
      2.3.3 A Brute Force Solution
      2.3.4 A Modular Approach

3 Semantic Structures
   3.1 Expectations
      3.1.1 Bounded Functions
      3.1.2 Non-Negative Functions
      3.1.3 Sound Expectations
      3.1.4 Unitary expectations
      3.1.5 Standard Expectations
      3.1.6 Entailment
      3.1.7 Expectation Conjunction
      3.1.8 Rules Involving Conjunction
      3.1.9 Rules Involving Entailment and Conjunction Together
   3.2 Expectation Transformers
      3.2.1 Comparing Transformers
      3.2.2 Healthy Transformers
      3.2.3 Sublinearity
CONTENTS

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 .......... 123
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 Maximality ........................................ 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 .............................. 177
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 .................................. 190
4.9.6 The Algebra of Data Refinement ............ 193
4.9.7 Structural Rules for Correspondence ....... 200
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:

```plaintext
datatype coin = Heads | Tails

record coins =
  a :: coin
  b :: coin
```

The primitive state operation is \(\text{Apply}\), which takes a state transformer as an argument, constructs the pGCL equivalent. Thus \(\text{Apply} \ (a\text{-update} \ (\lambda s. \text{Heads}))\) sets the value of coin \(a\) to \(\text{Heads}\). 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 \(\text{Apply} \ (a\text{-update} \ (\lambda s. \text{Heads}))\) (note that the syntax translation involved does not apply to Latex output, and thus this lemma appears trivial):

```plaintext
lemma
  Apply (\lambda s. s(\{a := \text{Heads}\})) = (a := (\lambda s. \text{Heads}))
by(simp)
```

We can treat the record’s fields as the names of \textit{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 \(\text{Apply} \ (\lambda s. s(\{a := b \ s\}))\), which updates \(a\) with the current value of \(b\). If we wish to formally
establish that the previous statement is correct i.e. that in the final state, \( a \) really will have whatever value \( b \) had in the initial state, we must first introduce the assertion language.

### 2.1.2 Assertion and Annotation

Assertions in pGCL are real-valued functions of the state, which are often interpreted as a probability distribution over possible outcomes. These functions are termed expectations, for reasons which shortly be clear. Initially, however, we need only consider standard expectations: those derived from a binary predicate. A predicate \( P: s \rightarrow \text{bool} \) is embedded as \( \langle P \rangle: s \rightarrow \text{real} \), such that \( P s \rightarrow \langle P \rangle s = 1 \land \neg P s \rightarrow \langle P \rangle s = 0 \).

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 \( \langle P \rangle \vdash \langle Q \rangle \), \( \langle ?P \rangle \vdash \langle ?Q \rangle \); \( ?P \ ?s \implies \ ?Q \ ?s \). Thus a valid specification of \( \text{Apply} (\lambda s. s (| a := b s |)) \) is:

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

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:

\[
\langle \lambda s::\text{real. } s \neq 0 \rangle \vdash \wp (\text{Apply ((*) 2) :: Apply (\lambda s. s / s)}) \langle \lambda s. \text{s = 1} \rangle
\]

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 \( x \oplus y \) 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

type 's ⇒ real. The following annotation states that the probability of heads is exactly 1/2:

definition
flip-a :: real ⇒ coins prog
where
flip-a p = a := (λ-. Heads) (λs. p)⊕ a := (λ-. Tails)

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

Sufficiently small problems can be handled by the simplifier, by symbolic evaluation.
by(simp add:wp-eval o-def)

2.1.4 Nondeterminism

We can also under-specify a program, using the nondeterministic choice operator, x ∩ 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
λs. 1/3 ⊢ wp (flip-a (2/3) ∩ flip-a (1/3)) «λs. a s = Heads»
unfolding flip-a-def
by(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
flip-b :: real ⇒ coins prog
where
flip-b p = b := (λ-. Heads) (λs. p)⊕ b := (λ-. Tails)

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

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

definition Puniform :: 'a set ⇒ ('a ⇒ real)
where Puniform S = (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 := (λ-. v)) red
  by(simp add:wp-eval supp-Puniform sum.atLeast-Suc-atMost Puniform-in)

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 sum.atLeast-Suc-atMost Puniform-in)

end

2.2 Loops

theory LoopExamples imports ../pGCL begin

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

2.2.1 Guaranteed Termination

We start with a completely classical loop, to show that standard techniques apply. Here, we have a program that simply decrements a counter until it hits zero:
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 ≤ (0 :: '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.
CHAPTER 2. INTRODUCTION TO PGCL

\[
\begin{align*}
\text{show } & \forall n. \forall \lambda x. \text{nat}(\max x 0) = \text{Suc } n \land \forall \text{inv-counts } \vdash \wp ?p \forall \lambda x. \text{nat}(\max x 0) = n \rangle \\
\text{unfolding } & \text{inv-count-def} \\
\text{by } & (pvcg, \\
\text{auto simp: } & o-def \text{ exp-conj-std-split}[\text{symmetric}] \\
\text{intro: } & \text{implies-entails}) \\
\text{qed}
\end{align*}
\]

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.

\[
\begin{align*}
\text{type-synonym } & \text{coin } = \text{bool} \\
\text{definition } & \text{Heads } = \text{True} \\
\text{definition } & \text{Tails } = \text{False} \\
\text{definition } & \text{flip } :: \text{coin prog} \\
\text{where } & \text{flip } = \text{Apply } (\lambda - . \text{Heads}) (\lambda s. \frac{1}{2}) \oplus \text{Apply } (\lambda - . \text{Tails}) \\
\end{align*}
\]

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

\[
\begin{align*}
\text{definition } & \text{wait-for-heads } :: \text{coin prog} \\
\text{where } & \text{wait-for-heads } = \text{do } (\neg \text{Heads}) \rightarrow \text{flip } \text{od} \\
\end{align*}
\]

Nonetheless, we can show termination.

\[
\begin{align*}
\text{lemma } & \text{wait-for-heads-term:} \\
\text{\lambda s. } & 1 \vdash \wp \text{ wait-for-heads } (\lambda s. \text{1}) \\
\text{unfolding } & \text{wait-for-heads-def} \\
\end{align*}
\]

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.

\[
\begin{align*}
\text{proof}(\text{rule termination-0-1}) \\
\text{show } & \text{well-def flip} \\
\text{unfolding } & \text{flip-def} \\
\text{by } & (\text{auto intro:wd-intros}) \\
\end{align*}
\]

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

\[
\begin{align*}
\text{show } & \text{maximal } (\wp \text{ flip}) \\
\text{unfolding } & \text{flip-def by } (\text{auto intro:max-intros}) \\
\end{align*}
\]

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

The Monty Hall Problem

theory Monty imports ../pGCL begin

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

The setting is a game show: There is a prize hidden behind one of three doors, and the contestant is invited to choose one. Once the guess is made, the host than opens one of the remaining two doors, revealing a goat and showing that the prize is elsewhere. The 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 ⇒ bool
  where player-wins g ≡ guess g = prize g
CHAPTER 2. INTRODUCTION TO PGCL

Invariants

We prove explicitly that only valid doors are ever chosen.

\textbf{definition} \texttt{inv-prize} :: \texttt{game} \Rightarrow \texttt{bool}
\textbf{where} \texttt{inv-prize} \texttt{g} \equiv \texttt{prize g} \in \{1, 2, 3\}

\textbf{definition} \texttt{inv-clue} :: \texttt{game} \Rightarrow \texttt{bool}
\textbf{where} \texttt{inv-clue} \texttt{g} \equiv \texttt{clue g} \in \{1, 2, 3\}

\textbf{definition} \texttt{inv-guess} :: \texttt{game} \Rightarrow \texttt{bool}
\textbf{where} \texttt{inv-guess} \texttt{g} \equiv \texttt{guess g} \in \{1, 2, 3\}

\subsection{2.3.2 The Game}

Hide the prize behind door \(D\).

\textbf{definition} \texttt{hide-behind} :: \texttt{nat} \Rightarrow \texttt{game prog}
\textbf{where} \texttt{hide-behind} \texttt{D} \equiv \texttt{Apply (prize-update (\lambda x. D))}

Choose door \(D\).

\textbf{definition} \texttt{guess-behind} :: \texttt{nat} \Rightarrow \texttt{game prog}
\textbf{where} \texttt{guess-behind} \texttt{D} \equiv \texttt{Apply (guess-update (\lambda x. D))}

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

\textbf{definition} \texttt{open-door} :: \texttt{nat} \Rightarrow \texttt{game prog}
\textbf{where} \texttt{open-door} \texttt{D} \equiv \texttt{Apply (clue-update (\lambda x. D))}

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

\textbf{definition} \texttt{hide-prize} :: \texttt{game prog}
\textbf{where} \texttt{hide-prize} \equiv \texttt{hide-behind 1 \cap hide-behind 2 \cap hide-behind 3}

Guess uniformly at random.

\textbf{definition} \texttt{make-guess} :: \texttt{game prog}
\textbf{where} \texttt{make-guess} \equiv \texttt{guess-behind 1 (\lambda s. 1/3)} \oplus \texttt{guess-behind 2 (\lambda s. 1/2)} \oplus \texttt{guess-behind 3}

Open one of the two doors that \textit{doesn’t} hide the prize.

\textbf{definition} \texttt{reveal} :: \texttt{game prog}
\textbf{where} \texttt{reveal} \equiv \prod \texttt{d} \in (\lambda s. \{1, 2, 3\} - \{\texttt{prize s, guess s}\}). \texttt{open-door} \texttt{d}

Switch your guess to the other unopened door.

\textbf{definition} \texttt{switch-guess} :: \texttt{game prog}
\textbf{where} \texttt{switch-guess} \equiv \prod \texttt{d} \in (\lambda s. \{1, 2, 3\} - \{\texttt{clue s, guess s}\}). \texttt{guess-behind} \texttt{d}

The complete game, either with or without switching guesses.

\textbf{definition} \texttt{monty} :: \texttt{bool} \Rightarrow \texttt{game prog}
2.3. THE MONTY HALL PROBLEM

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.

\textbf{lemma eval-win[simp]:}

\[
p = g \implies \langle \text{player-wins} \rangle (s | \text{prize} := p, \text{guess} := g, \text{clue} := c |) = 1
\]

\textbf{by (simp add: embed-bool-def player-wins-def)}

\textbf{lemma eval-loss[simp]:}

\[
p \neq g \implies \langle \text{player-wins} \rangle (s | \text{prize} := p, \text{guess} := g, \text{clue} := c |) = 0
\]

\textbf{by (simp add: embed-bool-def player-wins-def)}

If they stick to their guns, the player wins with \( p = \frac{1}{3} \).

\textbf{lemma wp-monty-noswitch:}

\[
(\lambda s. \frac{1}{3}) = \text{wp} (\text{monty False}) \langle \text{player-wins} \rangle
\]

\textbf{unfolding monty-def hide-prize-def make-guess-def reveal-def}

\textbf{hide-behind-def guess-behind-def open-door-def}

\textbf{switch-guess-def}

\textbf{by (simp add: wp-eval insert-Diff-if o-def)}

\textbf{lemma swap-upd:}

\[
s | \text{prize} := p, \text{clue} := c, \text{guess} := g | =
\]

\[
s | \text{prize} := p, \text{guess} := g, \text{clue} := c |
\]

\textbf{by (simp)}

If they switch, they win with \( p = \frac{2}{3} \). 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.

\textbf{lemma wp-monty-switch-bruteforce:}

\[
(\lambda s. 2/3) = \text{wp} (\text{monty True}) \langle \text{player-wins} \rangle
\]

\textbf{unfolding monty-def hide-prize-def make-guess-def reveal-def}

\textbf{hide-behind-def guess-behind-def open-door-def}

\textbf{switch-guess-def}

— Note that this is getting slow

\textbf{by (simp add: wp-eval insert-Diff-if swap-upd o-def cong del: INF-cong-simp)}

2.3.4 A Modular Approach

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

**Healthiness**

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

**lemma** *wd-hide-prize*:

**well-def** *hide-prize*

**unfolding** *hide-prize-def* *hide-behind-def*

**by** (*simp add:* *wd-intros*)

**lemma** *wd-make-guess*:

**well-def** *make-guess*

**unfolding** *make-guess-def* *guess-behind-def*

**by** (*simp add:* *wd-intros*)

**lemma** *wd-reveal*:

**well-def** *reveal*

**proof** –

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

**have** \( \forall s. \{1, 2, 3\} - \{\text{prize } s, \text{guess } s\} \neq \{\} \)

**by** (*auto simp:* *insert-Diff-if*)

**thus** ?thesis

**unfolding** *reveal-def* *open-door-def*

**by** (*intro* *wd-intros*, *auto*)

**qed**

**lemma** *wd-switch-guess*:

**well-def** *switch-guess*

**proof** –

**have** \( \forall s. \{1, 2, 3\} - \{\text{clue } s, \text{guess } s\} \neq \{\} \)

**by** (*auto simp:* *insert-Diff-if*)

**thus** ?thesis

**unfolding** *switch-guess-def* *guess-behind-def*

**by** (*intro* *wd-intros*, *auto*)

**qed**

**lemmas** *monty-healthy* =

*wd-switch-guess* *wd-reveal* *wd-make-guess* *wd-hide-prize*

**Annotations**

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

hide-prize chooses a valid door.

**lemma** wp-hide-prize:

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

**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:

\[
(\lambda s. 2/3 * \langle \lambda g. \text{inv-prize g} \rangle s) \vdash
\]

**unfolding** make-guess-def guess-behind-def wp-eval o-def

by (auto simp: embed-bool-def inv-prize-def inv-guess-def)

**lemma** last-one:

assumes \(a \neq b\) and \(a \in \{1::nat,2,3\}\) and \(b \in \{1,2,3\}\)

shows \(\exists!c. \{1,2,3\} - \{b,a\} = \{c\}\)

**apply** (simp add: insert-Diff-if)

**using** assms by (auto intro: assms)

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

**lemma** wp-reveal:

\[
\langle \lambda g. \text{guess g} \neq \text{prize g} \land \text{inv-prize g} \land \text{inv-guess g} \rangle \vdash
\]

**unfolding** inv-prize-def inv-guess-def

by (force dest: last-one elim: ex1E)

ultimately show \(1 \leq \text{wp reveal }\langle \text{prize} \rangle\)

**proof** (rule use-premise, rule well-def-wp-healthy[OF wd-reveal], clarify)

fix \(s\)

**assume** \(\text{guess s} \neq \text{prize s}\)

and \(\text{inv-prize s}\)

and \(\text{inv-guess s}\)

moreover then obtain \(c\)

**where** singleton: \(\{\text{Suc 0,2,3}\} - \{\text{prize s, guess s}\} = \{c\}\)

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 \(1 \leq \text{wp reveal }\langle \text{prize} \rangle\)

**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:
[ guess g ≠ prize g; clue g ≠ prize g; clue g ≠ guess g; inv-prize g; inv-guess g; inv-clue g ]
{1, 2, 3} = {guess g, prize g, clue g}
unfolding inv-prize-def inv-guess-def inv-clue-def
apply (rule set-eqI)
apply (rule iffI)
apply (clarify)
apply (metis (full-types) empty-iff insert-iff)
apply (metis insert-iff)
done

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

lemma wp-switch-guess:
«λg. guess g ≠ prize g ∧ clue g ≠ prize g ∧ clue g ≠ guess g ∧ inv-prize g ∧ inv-guess g ∧ inv-clue g» ⊢ ⊢ wp switch-guess «player-wins»
proof (rule use-premise, safe)
from wd-switch-guess show healthy (wp switch-guess) by (auto)

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

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

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

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

**Using the VCG**

lemmas 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:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a → b</th>
<th>x</th>
<th>y</th>
<th>x ≤ y</th>
</tr>
</thead>
<tbody>
<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

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 :: real $\Rightarrow$ (’a $\Rightarrow$ real) $\Rightarrow$ bool
where
 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-byI[intro]:
 $[ \ \forall x. \ P \ x \leq b \ ] \ \Rightarrow \ bounded-by \ b \ P$
 by (simp add:bounded-by-def)

lemma bounded-byI2[intro]:
 $P \leq (\lambda s. \ b) \ \Rightarrow \ bounded-by \ b \ P$
 by (blast dest:le-funD)

lemma bounded-byD[dest]:
 bounded-by $b \ P \ \Rightarrow \ P \ x \leq b$
 by (simp add:bounded-by-def)

lemma bounded-byD2[dest]:
 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 :: (’a $\Rightarrow$ real) $\Rightarrow$ bool
where
 bounded $P \equiv (\exists b. \ bounded-by \ b \ P)$

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

definition bound-of :: (’a $\Rightarrow$ real) $\Rightarrow$ real
where
 bound-of $P \equiv \text{Sup} \ (P \cdot \text{UNIV})$

lemma bounded-bdd-above[intro]:
 assumes $bP$: bounded $P$
 shows bdd-above (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:Inf-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 \( P::'a \Rightarrow \text{real} \)
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]:
\( \text{bounded} P \implies P x \leq \text{bound-of} P \)
by (blast intro:bounded-byD)

lemma bounded-by-mono:
\( \text{bounded-by} a P; a \leq b \) \implies \( \text{bounded-by} b P \)
unfolding bounded-def by (blast intro:order-trans)

lemma bounded-by-imp-bounded[intro]:
\( \text{bounded-by} b P \implies \text{bounded} P \)
unfolding bounded-def by (blast)

This is occasionally easier to apply:

lemma bounded-by-bound-of-alt:
\( \text{bounded} P; \text{bound-of} P = a \) \implies \( \text{bounded-by} a P \)
by (blast)

lemma bounded-const[simp]:
\( \text{bounded} (\lambda x. c) \)
by (blast)

lemma bounded-by-const[intro]:
c \leq b \implies \text{bounded-by} b (\lambda x. c)
by (blast)

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

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

lemma le-bound-of[intro]:
  \( \forall x. \text{bounded } f \Rightarrow f x \leq \text{bound-of } f \)
  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 \iff (\forall x. 0 \leq P x)

lemma nneg1[intro]:
  \[ \forall x. 0 \leq P x \Rightarrow nneg P \]
  by (simp add:nneg-def)

lemma nneg12[intro]:
  (\lambda s. 0) \leq P \Rightarrow nneg P
  by (blast dest:le-funD)

lemma nnegD[dest]:
  nneg P \Rightarrow 0 \leq P x
  by (simp add:nneg-def)

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

lemma nneg-bdd-below[intro]:
  nneg P \Rightarrow \text{bdd-below \ (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]:

\[ \text{nneg} \ P \implies \text{nneg} \ (P \circ f) \]

*by* (force)

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

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

*by* (blast intro:order-trans)

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

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

*by* (blast intro:order-trans)

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

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

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

*by* (blast intro:order-trans)

### 3.1.3 Sound Expectations

**definition** sound :: ('s => real) => bool

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

Combining \textit{nneg} and \textit{Expectations.bounded}, we have \textit{sound} expectations. We set up the classical reasoner and the simplifier, such that showing soundness, 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:

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

*by* (simp add: sound-def)

**lemma** soundI2[intro]:

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

*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[dest]:

*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]:
assumes $sP$: 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 $\forall s. P s + Q s \leq \text{bound-of } P + \text{bound-of } Q$
  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:
assumes $sP$: sound $P$ and $sQ$: sound $Q$
shows sound ($\lambda s. P s \ast Q s$)
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 $\forall s. P s \ast Q s \leq \text{bound-of } P \ast \text{bound-of } Q$
  using $sP$ and $sQ$ by(blast intro:mult-mono)
thus bounded-by ($\text{bound-of } P \ast \text{bound-of } Q$) ($\lambda s. P s \ast Q s$) by(blast)
from $sP$ and $sQ$ show $\text{nneg}$ ($\lambda s. P s \ast Q s$)
  by(blast intro:mult-nonneg-nonneg)
qed

lemma div-sound:
assumes $sP$: sound $P$ and cpos: $0 < c$
shows sound ($\lambda s. P s / c$)
proof
from $sP$ and 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:
assumes $sP$: sound $P$ and nnc: $0 \leq c$
3.1. EXPECTATIONS

shows sound (\(\lambda s. P s \odot c\))
proof (rule soundI)
  from \(s P\) 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 \(\text{nneq}\) (\(\lambda s. P s \odot c\)) by (blast)
qed

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

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

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

lemma sc-bound[simp]:
assumes \(s P: \text{sound } P\)
and \(\text{nnc}: 0 \leq c\)
shows \(c \ast \text{bound-of } P = \text{bound-of } (\lambda x. c \ast P x)\)
proof (cases \(c = 0\))
case True then show \(\text{?thesis}\) by (simp)
next
case False with \(\text{nnc}\) have \(c < c\) by (auto)
show \(\text{?thesis}\)
proof (rule antisym)
  from \(s P\) and \(\text{nnc}\) have 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 \(s P\) 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 \(\text{bound-of } (\lambda x. c \ast P x) \leq c \ast \text{bound-of } P\)
by (blast)  
qed  
qed

**Lemma sc-sound:**  
\[
\left[ \text{sound } P; \ 0 \leq c \right] \implies \text{sound } (\lambda s. c \ast P \ s)
\]
by (blast intro:mult-nonneg-nonneg)

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

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

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

**Lemma unit-mult[intro]:**  
assumes \( sP: \forall x \in S. \text{sound } P \ x \)
shows \( \text{bounded-by } 1 \ (\lambda s. \sum x \in S. P \ s) \)
proof (rule bounded-byI)
fix \( s \)
have \( P \ s \ast Q \ s \leq 1 \ast 1 \)
using \( \text{assms by(blast dest:bounded-by-mult)} \)
thus \( P \ s \ast Q \ s \leq 1 \) by (simp)
qed

**Lemma sum-sound:**  
assumes \( sP: \forall x \in S. \text{sound } P \ x \)
shows \( \text{sound } (\lambda s. \sum x \in S. P \ x \ s) \)
proof (rule soundI2)
from \( sP \) show \( \text{bounded-by } (\sum x \in S. \text{bound-of } (P \ x)) \ (\lambda s. \sum x \in S. P \ x \ s) \)
by (auto intro!:sum-mono)
from \( sP \) show \( \text{nneg } (\lambda s. \sum x \in S. P \ x \ s) \)
by (auto intro!:sum-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 unitaryI[intro]:
  [ sound P; bounded-by 1 P ] ⇒ unitary P
  by(simp add:unitary-def)

lemma unitaryI2:
  [ 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.
**CHAPTER 3. SEMANTIC STRUCTURES**

**lemma** embed-bool-idem:

\[ «P» s \ast \ «P» s = «P» s \]

by (simp add:embed-bool-def)

**lemma** eval-embed-true[simp]:

\[ P s \Longrightarrow «P» s = 1 \]

by (simp add:embed-bool-def)

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

\[ \neg P s \Longrightarrow «P» s = 0 \]

by (simp add:embed-bool-def)

**lemma** embed-ge-0[simp,intro]:

\[ 0 \leq «G» s \]

by (simp add:embed-bool-def)

**lemma** embed-le-1[simp,intro]:

\[ «G» s \leq 1 \]

by (simp add:embed-bool-def)

**lemma** embed-le-1-alt[simp,intro]:

\[ 0 \leq 1 - «G» s \]

by (subst add-le-cancel-right[where c=«G» s, symmetric], simp)

**lemma** expect-1-I:

\[ P x \Longrightarrow 1 \leq «P» x \]

by (simp)

**lemma** standard-sound[intro,simp]:

sound «P»

by (blast)

**lemma** embed-o[simp]:

\[ «P» o f = «P o f» \]

unfolding embed-bool-def o-def by (simp)

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

**definition** negate :: ('s \Rightarrow bool) \Rightarrow 's \Rightarrow bool (N)

where negate P = (\lambda s. \neg P s)

**lemma** negateI:

\[ \neg P s \Longrightarrow N P s \]

by (simp add:negate-def)

**lemma** embed-split:

\[ f s = «P» s \ast f s + \neg N P* s \ast f s \]

by (simp add:negate-def embed-bool-def)
3.1. EXPECTATIONS

lemma negate-embed:
«NNP» s = 1 - «PP» s
by (simp add:embed-bool-def negate-def)

lemma eval-nembed-true[simp]:
P s ⇒ «NNP» s = 0
by (simp add:embed-bool-def negate-def)

lemma eval-nembed-false[simp]:
¬P s ⇒ «NNP» s = 1
by (simp add:embed-bool-def negate-def)

lemma negate-Not[simp]:
NNNot = (λx. x)
by(simp add:negate-def)

lemma negate-negate[simp]:
NN(NNP) = P
by(simp add:negate-def)

lemma embed-bool-cancel:
«G» s * «NNG» 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 ⊢ Q ≡ P ≤ Q

lemma entailsI[intro]:
[∀s. P s ≤ Q s] ⇒ P ⊢ Q
by(simp add:le-funI)

lemma entailsD[dest]:
P ⊢ Q ⇒ P s ≤ Q s
by(simp add:le-funD)

lemma eq-entails[intro]:
P = Q ⇒ P ⊢ Q
by(blast)

lemma entails-trans[trans]:
[ P ⊢ Q; Q ⊢ R ] ⇒ P ⊢ 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:
\[ \forall s. \ P s \Rightarrow Q s \ \Rightarrow \ \langle P \rangle \cdot \langle Q \rangle \]
by (rule entailsI, case-tac P s, simp-all)

lemma entails-implies:
\[ \forall s. \ \langle P \rangle \cdot \langle Q \rangle; \ P s \ \Rightarrow \ Q s \]
by (rule ccontr, drule-tac s=s in entailsD, simp)

3.1.7 Expectation Conjunction

definition pconj :: real ⇒ real ⇒ real (infixl .& 71)
where
p .& q ≡ p + q ⊖ 1

definition exp-conj :: (′s ⇒ real) ⇒ (′s ⇒ real) ⇒ (′s ⇒ real) (infixl && 71)
where a && b ≡ λs. (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]:
\[ b \leq 1 \Rightarrow 0 .& b = 0 \]
by (simp add: pconj-def tminus-def)

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

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

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

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

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

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

\[ a \& (b \& c) = (a \& b) \& c \]

**lemma** pconj-mono:

\[ [a \leq b; c \leq d] \Rightarrow a \& c \leq b \& d \]

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

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

\[ 0 \leq a \& b \]

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

**lemma** min-pconj:

\((\min a b) \& (\min c d) \leq \min (a \& c) (b \& d)\)

**by**(cases \(a \leq b\),

\(cases c \leq d,\)

\(simp-all add:min.absorb1 min.absorb2 pconj-mono]),

\(cases c \leq d,\)

\(simp-all add:min.absorb1 min.absorb2 pconj-mono))\)

**lemma** pconj-less-one[simp]:

\[ a + b < 1 \Rightarrow a \& b = 0 \]

**unfolding** pconj-def **by**(simp)

**lemma** pconj-ge-one[simp]:

\[ 1 \leq a + b \Rightarrow a \& b = a + b - 1 \]

**unfolding** pconj-def **by**(simp)

**lemma** pconj-idem[simp]:

\[ «P» s \& «P» s = «P» s \]

**unfolding** pconj-def **by**(cases \(P s\), simp-all)

3.1.8 Rules Involving Conjunction.

**lemma** exp-conj-mono-left:

\[ P \vdash Q \Rightarrow P \& & R \vdash Q \& & R \]

**unfolding** exp-conj-def pconj-def

**by**(auto intro:tminus-left-mono add-right-mono)

**lemma** exp-conj-mono-right:

\[ Q \vdash R \Rightarrow P \& & Q \vdash P \& & R \]

**unfolding** exp-conj-def pconj-def

**by**(auto intro:tminus-left-mono add-left-mono)

**lemma** exp-conj-comm[ac-simps]:

\[ a \& & b = b \& & a \]

**by**(simp add:exp-conj-def ac-simps)

**lemma** exp-conj-bounded-by[intro,simp]:

assumes \(bP: \text{bounded-by } 1 P\)
and \( bQ \); bounded-by 1 \( Q \)
shows bounded-by 1 \((P \land Q)\)
proof (rule bounded-byI, unfold exp-conj-def pconj-def)
fix \( x \)
from \( bP \) have \( P x \leq 1 \) by (blast)
moreover from \( bQ \) have \( Q x \leq 1 \) by (blast)
ultimately have \( P x + Q x \leq 2 \) by (auto)
thus \( P x + Q x \land 1 \leq 1 \)
unfolding tminus-def by (simp)
qed

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

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

lemma exp-conj-top-left [simp]:
sound \( P \Longrightarrow \langle \lambda s. \text{True} \rangle \land P = P \)
unfolding exp-conj-def by (force)

lemma exp-conj-top-right [simp]:
sound \( P \Longrightarrow P \land \langle \lambda s. \text{True} \rangle = P \)
unfolding exp-conj-def by (force)

lemma exp-conj-idem [simp]:
\( \langle P \rangle \land \langle P \rangle = \langle P \rangle \)
unfolding exp-conj-def
by (rule ext, cases \( P s \), simp-all)

lemma exp-conj-nneg [intro, simp]:
\((\langle s \rangle. 0) \leq P \land Q \)
unfolding exp-conj-def
by (blast intro: le-funI)
3.1. EXPECTATIONS

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

lemma \textit{exp-conj-rzero[simp]}:
bounded-by \( 1 P \implies P \&\& (\lambda s. 0) = (\lambda s. 0) \)
unfolding \textit{exp-conj-def} by (force)

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

lemma \textit{exp-conj-std-split}:
\( \langle \lambda s. P s \&\& Q s \rangle = \langle P \rangle \&\& \langle Q \rangle \)
unfolding \textit{exp-conj-def} \textit{embed-bool-def} \textit{pconj-def}
by (auto)

3.1.9 Rules Involving Entailment and Conjunction Together

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

lemma \textit{entails-frame}:
assumes \textit{ePR}: \( P \vdash R \)
and \textit{eQS}: \( Q \vdash S \)
shows \( P \&\& Q \vdash R \&\& S \)
proof (rule le-funI)
fix \( s \)
from ePR have \( P \leq R \) by (blast) 
morerover from eQS have \( Q \leq S \) by (blast) 
ultimately have \( P + Q \leq R + S \) by (rule add-mono) 
hence \( (P \&\& Q) \leq (R \&\& S) \) 
unfolding exp-conj-def pconj-def 
qed 

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

lemma pentails-cases: 
assumes \( PQe: \forall x. P x \vdash Q x \) 
and exhaust: \( \forall s. \exists x. P (x s) s = 1 \) 
and framed: \( \forall x. P x \&\& \Lambda R x \vdash Q x \&\& S \) 
and \( sR: \text{sound } R \) and \( sS: \text{sound } S \) 
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) 
morerover \{ 
hence \( 1 = P x s \) by (simp) 
also from \( PQe \) have \( P x s \leq Q x s \) by (blast dest:le-funD) 
finally have \( Q x s = 1 \) 
using \( bQ \) by (blast intro:antisym) 
\} 
morerover note le-funD[OF framed[where \( x=x \)], where \( x=s \)] 
morerover from \( sR \) have \( 0 \leq R s \) by (blast) 
morerover 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]: 
\[
[ 0 \leq c; c \leq 1 ] \Rightarrow \text{unitary } (\lambda s. c)
\] 
by (auto) 

lemma unitary-mult:
3.2. EXPECTATION TRANSFORMERS

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

lemma exp-conj-unitary:
  [ unitary P; unitary Q ] ⇒ unitary (P && Q)
  by (intro unitaryI2 nnegI2, auto)

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

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

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

end

3.2 Expectation Transformers

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

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

The set of healthy transformers is the universe into which we place our semantic interpretation of pGCL programs. In its standard presentation, the healthiness condition for pGCL programs is sublinearity, for demonic programs, and superlinearity for angelic programs. We extract a minimal core property, consisting of monotonicity, feasibility and scaling to form our healthiness property, which holds across all programs. The additional components of sublinearity are broken out separately, and shown later. The two reasons for this are firstly to avoid the effort of establishing sub-(super-)linearity globally, and to allow us to define primitives whose sublinearity, and indeed healthiness, depend on context.
Consider again the automaton of Figure 3.1. Here, the effect of executing the automaton from its initial state (a) until it reaches some final state (b or c) is to transform the expectation on final states (P), into one on initial states, giving the expected value of the function on termination. Here, the transformation is linear: \( P_{\text{prior}}(a) = 0.7 \times P_{\text{post}}(b) + 0.3 \times P_{\text{post}}(c) \), 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

![Diagram of a diverging automaton]

Figure 3.3: A diverging automaton.

arguments in Section 4.11.

Having reached $a$, the automaton will proceed to $b$ with probability $0.5 \times (1/(0.5 + 0.3)) = 0.625$, 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, $P_{\text{prior}}(e) = 0.3125 \times P_{\text{post}}(b) + 0.1875 \times P_{\text{post}}(c)$. 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 $\cap$ and $\sqcup$, which are themselves feasible. We are thus able to define both least and greatest fixed points on this set, and thus give semantics to recursive programs built from feasible components.
3.2.1 Comparing Transformers

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

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

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

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

lemma le-transI[intro]:
[ [ P. sound P ⇒ t P ≤ u P ] ] ⇒ le-trans t u
by(simp add:le-trans-def)

lemma le-utransI[intro]:
[ [ P. unitary P ⇒ t P ≤ u P ] ] ⇒ le-utrans t u
by(simp add:le-utrans-def)

lemma le-transD[dest]:
[ le-trans t u; sound P ] ⇒ t P ≤ u P
by(simp add:le-trans-def)

lemma le-utransD[dest]:
[ le-utrans t u; unitary P ] ⇒ t P ≤ u P
by(simp add:le-utrans-def)

lemma le-trans-trans[trans]:
[ le-trans x y; le-trans y z ] ⇒ le-trans x z
unfolding le-trans-def by(blast dest:order-trans)

lemma le-utrans-trans[trans]:
[ le-utrans x y; le-utrans y z ] ⇒ le-utrans x z
unfolding le-utrans-def by(blast dest:order-trans)

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

lemma le-utrans-refl[iff]:
le-utrans x x
by(simp add:le-utrans-def)
3.2. EXPECTATION TRANSFORMERS

lemma le-trans-le-utrans[dest]:
le-trans t u ⇒ le-utrans t u
unfolding le-trans-def le-utrans-def by(auto)

definition
l-trans :: 's trans ⇒ 's trans ⇒ bool
where
l-trans t u ←→ le-trans t u ∧ ¬ le-trans u t

Transformer equivalence is induced by comparison:

definition
equiv-trans :: 's trans ⇒ 's trans ⇒ bool
where
equiv-trans t u ←→ le-trans t u ∧ le-trans u t

definition
equiv-utrans :: 's trans ⇒ 's trans ⇒ bool
where
equiv-utrans t u ←→ le-utrans t u ∧ le-utrans u t

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

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

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

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

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

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

lemma le-trans-antisym:
[ le-trans x y; le-trans y x \] ⇒ equiv-trans x y
unfolding equiv-trans-def by(simp)

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

**unfolding** \text{equiv-utrans-def} \text{ by}(simp)

**lemma** \text{equiv-trans-comm}[\text{ac-simps}]:
\text{equiv-trans } t \ u \ \leftrightarrow \text{ equiv-trans } u \ t

**unfolding** \text{equiv-trans-def} \text{ by}(blast)

**lemma** \text{equiv-utrans-comm}[\text{ac-simps}]:
\text{equiv-utrans } t \ u \ \leftrightarrow \text{ equiv-utrans } u \ t

**unfolding** \text{equiv-utrans-def} \text{ by}(blast)

**lemma** \text{equiv-imp-le}[\text{intro}]:
\text{equiv-trans } t \ u \ \Rightarrow \text{ le-trans } t \ u

**unfolding** \text{equiv-trans-def} \text{ by}(clarify)

**lemma** \text{equiv-imp-le}[\text{intro}]:
\text{equiv-utrans } t \ u \ \Rightarrow \text{ le-utrans } t \ u

**unfolding** \text{equiv-utrans-def} \text{ by}(clarify)

**lemma** \text{equiv-imp-le-alt}:
\text{equiv-trans } t \ u \ \Rightarrow \text{ le-trans } u \ t

\text{by}(\text{force simp:ac-simps})

**lemma** \text{equiv-utrans-le-alt}:
\text{equiv-utrans } t \ u \ \Rightarrow \text{ le-utrans } u \ t

\text{by}(\text{force simp:ac-simps})

**lemma** \text{le-trans-equiv-rsp}[\text{simp}]:
\text{equiv-trans } t \ u \ \Rightarrow \text{ le-trans } t \ v \ \leftrightarrow \text{ le-trans } u \ v

**unfolding** \text{equiv-trans-def} \text{ by}(\text{blast intro:le-trans-trans})

**lemma** \text{le-utrans-equiv-rsp}[\text{simp}]:
\text{equiv-utrans } t \ u \ \Rightarrow \text{ le-utrans } t \ v \ \leftrightarrow \text{ le-utrans } u \ v

**unfolding** \text{equiv-utrans-def} \text{ by}(\text{blast intro:le-utrans-trans})

**lemma** \text{equiv-trans-le-trans}[\text{trans}]:
\[ \text{equiv-trans } t \ u; \text{ le-trans } u \ v \ \Rightarrow \text{ le-trans } t \ v \]

\text{by}(\text{simp})

**lemma** \text{equiv-utrans-le-utrans}[\text{trans}]:
\[ \text{equiv-utrans } t \ u; \text{ le-utrans } u \ v \ \Rightarrow \text{ le-utrans } t \ v \]

\text{by}(\text{simp})

**lemma** \text{le-trans-equiv-rsp-right}[\text{simp}]:
\text{equiv-trans } t \ u \ \Rightarrow \text{ le-trans } v \ t \ \leftrightarrow \text{ le-trans } v \ u

**unfolding** \text{equiv-trans-def} \text{ by}(\text{blast intro:le-trans-trans})

**lemma** \text{le-utrans-equiv-rsp-right}[\text{simp}]:
\text{equiv-utrans } t \ u \ \Rightarrow \text{ le-utrans } v \ t \ \leftrightarrow \text{ le-utrans } v \ u
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[trans]:
    assumes xy: equiv-trans x y
    and yz: equiv-trans y z
    shows equiv-trans x z
    proof (rule le-trans-antisym)
      from xy have le-trans x y by (blast)
      also from yz have le-trans y z by (blast)
      finally show le-trans x z .
    from yz have le-trans z y by (force simp: ac-simps)
    also from xy have le-trans y x by (force simp: ac-simps)
    finally show le-trans z x .
    qed

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

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

3.2.2 Healthy Transformers

Feasibility

definition feasible :: (('a ⇒ real) ⇒ (′a ⇒ real)) ⇒ bool
where feasible t = (∀ P b. bounded-by b P ∧ nneg P →
    bounded-by b (t P) ∧ nneg (t P))

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

**Lemma feasibleI[intro]:**

\[
\forall b P. [\text{bounded-by } b P; \text{nneg } P] \implies \text{bounded-by } b (t P);
\forall b P. [\text{bounded-by } b P; \text{nneg } P] \implies \text{nneg } (t P) \implies \text{feasible } t
\]

by (force simp:feasible-def)

**Lemma feasible-boundedD[dest]:**

\[
[\text{feasible } t; \text{bounded-by } b P; \text{nneg } P] \implies \text{bounded-by } b (t P)
\]

by (simp add: feasible-def)

**Lemma feasible-nnegD[dest]:**

\[
[\text{feasible } t; \text{bounded-by } b P; \text{nneg } P] \implies \text{nneg } (t P)
\]

by (simp add: feasible-def)

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

\[
[\text{feasible } t; \text{sound } P] \implies \text{sound } (t P)
\]

by (rule soundI, unfold sound-def, (blast)+)

**Lemma feasible-pr-0[simp]:**

\[
\text{fixes } t::('s \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real}
\]

assumes \( ft: \text{feasible } t \)

shows \( t(\lambda x. 0) = (\lambda x. 0) \)

**Proof** (rule ext, rule antisym)

fix \( s \)

have \( \text{bounded-by } 0 (\lambda::'s. 0::\text{real}) \) by (blast)

with \( ft \) have \( \text{bounded-by } 0 (t (\lambda .. 0)) \) by (blast)

thus \( t(\lambda::'s. 0::\text{real}) \leq 0 \) by (blast)

have \( \text{nneg } (\lambda::'s. 0::\text{real}) \) by (blast)

with \( ft \) have \( \text{nneg } (t (\lambda .. 0)) \) by (blast)

thus \( 0 \leq t (\lambda .. 0) \) by (blast)

qed

**Lemma feasible-id:**

\( \text{feasible } (\lambda x. x) \)

unfolding feasible-def by (blast)

**Lemma feasible-bounded-by[dest]:**

\[
[\text{feasible } t; \text{sound } P; \text{bounded-by } b P] \implies \text{bounded-by } b (t P)
\]

by (auto)

**Lemma feasible-fixes-top:**

\( \text{feasible } t \implies t (\lambda s. 1) \leq (\lambda s. (1::\text{real})) \)

by (drule bounded-byD2[OF feasible-bounded-by], auto)

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

assumes \( ft: \text{feasible } t \)

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 \((\text{bound-of } (\lambda s. 0 :: \text{real})) (\lambda s. 0)\) 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 \( ft: \text{feasible } t \) and \( uP: \text{unitary } P \)
  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
  \( \text{mono-trans} :: (('s \Rightarrow \text{real}) \Rightarrow ('s \Rightarrow \text{real})) \Rightarrow \text{bool} \)

where
  \( \text{mono-trans } t \equiv \forall P Q. \text{ (sound } P \land \text{ sound } Q \land P \leq Q) \rightarrow t P \leq t Q \)

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]:
  \[ \forall P Q. \text{ (sound } P \land \text{ sound } Q \land P \leq Q) \rightarrow t P \leq t Q \] \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)
CHAPTER 3. SEMANTIC STRUCTURES

Scaling

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

definition
  scaling :: (('s ⇒ real) ⇒ ('s ⇒ real)) ⇒ bool
where
  scaling t ≡ ∀ P c x. sound P ∧ 0 ≤ c −→ c * t P x = t (λx. c * P x) x

The scaling and feasibility properties together allow us to treat transformers as a complete lattice, when operating on bounded expectations. The action of a transformer on such a bounded expectation is completely determined by its action on unitary expectations (those bounded by 1): t P s = bound-of P * t (λs. P s / 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]:
  [ ∀ P c x. [ sound P; 0 ≤ c ] −→ c * t P x = t (λx. c * P x) x ] −→ scaling t
by(simp add:scaling-def)

lemma scalingD[dest]:
  [ scaling t; sound P; 0 ≤ c ] −→ c * t P x = t (λx. c * P x) x
by(simp add:scaling-def)

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

Healthiness

Healthy transformers are feasible and monotonic, and respect scaling

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

lemma healthyI[intro]:
  [ feasible t; mono-trans t; scaling t ] −→ healthy t
by(simp add:healthy-def)
lemmas healthy-parts = healthyI[OF feasibleI mono-transI scalingI]

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

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

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

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

lemma healthy-bounded-byD[dest]:
  [ healthy t; bounded-by b P; nneg P ] ⇒ bounded-by b (t P)
  by(blast)

lemma healthy-bounded-byD2:
  [ healthy t; bounded-by b P; sound P ] ⇒ bounded-by b (t P)
  by(blast)

lemma healthy-boundedD[dest,simp]:
  [ healthy t; sound P ] ⇒ bounded (t P)
  by(blast)

lemma healthy-nnegD[dest,simp]:
  [ healthy t; sound P ] ⇒ nneg (t P)
  by(blast intro:feasible-nnegD)

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

lemma healthy-sound[dest]:
  [ healthy t; sound P ] ⇒ sound (t P)
  by(rule soundI, blast, blast intro:feasible-nnegD)

lemma healthy-unitary[dest]:
  [ healthy t; unitary P ] ⇒ unitary (t P)
  by(blast intro:unitaryI dest:unitary-bound healthy-bounded-byD)

lemma healthy-id[simp,intro]:
  healthy id
  by(simp add:healthyI feasibleI mono-transI scalingI)
lemmas \textit{healthy-fixes-bot} = \textit{feasible-fixes-bot}\{OF \textit{healthy-feasible-D}\}

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

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

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

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

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

\textbf{lemma healthy-equivI}:
\begin{align*}
\textit{fixes} \ \textit{t} &\colon (\textquoteleft s \Rightarrow \textit{real}) \Rightarrow \textquoteleft s \Rightarrow \textit{real} \ \text{and} \ u \\
\text{assumes} &\quad \textit{equiv}: \ \textit{equiv-trans} \ \textit{t} \ \textit{u} \\
\text{and} &\quad \textit{healthy}: \ \textit{healthy} \ \textit{t} \\
\text{shows} &\quad \textit{healthy} \ \textit{u} \\
\text{proof} &\quad \text{have} \ \textit{le-t-u}: \ \textit{le-trans} \ \textit{t} \ \textit{u} \ \text{by}(\text{blast intro}:\textit{equiv}) \\
\text{have} &\quad \textit{le-u-t}: \ \textit{le-trans} \ \textit{u} \ \textit{t} \ \text{by}(\text{simp add}:\textit{equiv-imp-le} \ \textit{ac-simps} \ \textit{equiv}) \\
\text{from} &\quad \textit{equiv} \ \text{have} \ \textit{eq-u-t}: \ \textit{equiv-trans} \ \textit{u} \ \textit{t} \ \text{by}(\text{simp add}:\textit{ac-simps}) \\
\text{show} &\quad \textit{feasible} \ \textit{u} \\
\text{proof} &\quad \text{fix} \ b \ \text{and} \ P::\textquoteleft s \Rightarrow \textit{real} \\
\text{assume} &\quad bP: \ \textit{bounded-by} \ b \ \textit{P} \ \text{and} \ nP: \ \textit{nneg} \ \textit{P} \\
\text{hence} &\quad sP: \ \textit{sound} \ \textit{P} \ \text{by}(\text{blast}) \\
\text{with} &\quad \textit{healthy} \ \text{have} \ \forall s. \ 0 \leq t \ P \ s \ \text{by}(\text{blast}) \\
\text{also from} &\quad sP \ \text{and} \ \textit{le-t-u} \ \text{have} \ \forall s. \ ... \ s \leq u \ P \ s \ \text{by}(\text{blast}) \\
\text{finally show} &\quad \textit{nneg} \ (u \ P) \ \text{by}(\text{blast}) \\
\text{from} &\quad sP \ \text{and} \ \textit{le-u-t} \ \text{have} \ \forall s. \ u \ P \ s \leq t \ P \ s \ \text{by}(\text{blast}) \\
\text{also from} &\quad \textit{healthy} \ \text{and} \ sP \ \text{and} \ bP \ \text{have} \ \forall s. \ t \ P \ s \leq b \ \text{by}(\text{blast}) \\
\text{finally show} &\quad \textit{bounded-by} \ b \ (u \ P) \ \text{by}(\text{blast}) \\
\text{qed} \\
\text{show} &\quad \textit{mono-trans} u \\
\text{proof} &\quad \text{fix} \ P::\textquoteleft s \Rightarrow \textit{real} \ \text{and} \ Q::\textquoteleft s \Rightarrow \textit{real} \\
\text{assume} &\quad sP: \ \textit{sound} \ \textit{P} \ \text{and} \ sQ: \ \textit{sound} \ Q \\
\text{and} &\quad \textit{le}: \ P \vdash Q \\
\text{from} &\quad sP \ \text{and} \ \textit{le-u-t} \ \text{have} \ u \ P \vdash t \ P \ \text{by}(\text{blast}) \\
\text{also from} &\quad sP \ \text{and} \ sQ \ \text{and} \ \textit{le} \ \text{and} \ \textit{healthy} \ \text{have} \ t \ P \vdash Q \ \text{by}(\text{blast}) \\
\text{also from} &\quad sQ \ \text{and} \ \textit{le-t-u} \ \text{have} \ t \ Q \vdash u \ Q \ \text{by}(\text{blast}) \\
\text{finally show} &\quad u \ P \vdash u \ Q .
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:eqv-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)
with \( nc \) have \( \forall s. c \ast t P s \leq c \ast b \) by ( blast intro: mult-left-mono )
also { 
  from \( mP \) and \( bP \) have \( 0 \leq b \) by ( auto )
  with \( bc \) have \( c \ast b \leq 1 \ast b \) by ( blast intro: mult-right-mono )
  hence \( c \ast b \leq b \) by ( simp )
}
finally show bounded-by b ( \( \lambda s. c \ast t P s \) ) by ( blast )
from \( ht \) \( nnP \) \( bP \) have \( \forall s. 0 \leq t P s \) by ( blast )
with \( nc \) have \( \forall s. c \ast t P s \leq c \ast t Q s \) by ( rule mult-nonneg-nonneg )
thus \( \neg neg ( \lambda s. c \ast t P s ) \) by ( blast )
qed
show mono-trans ( \( \lambda P s. c \ast t P s \) )
proof
  fix \( P::'s \Rightarrow \) real and \( Q \)
  assume \( sP: sound P \) and \( sQ: sound Q \) and \( le: P \vdash Q \)
  with \( ht \) have \( \forall s. t P s \leq t Q s \) by ( auto intro: le-funD )
  with \( nc \) have \( \forall s. c \ast t P s \leq c \ast t Q s \)
  by ( blast intro: mult-left-mono )
  thus \( \lambda s. c \ast t P s \vdash \lambda s. c \ast t Q s \) by ( blast )
qed
from \( ht \) show scaling ( \( \lambda P s. c \ast t P s \) )
  by ( auto simp: scalingD healthy-scalingD \( ht \) )
qed

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

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

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

definition
  nearly-healthy :: (('s => real) => ('s => real)) => bool
where
  nearly-healthy t \( \longleftrightarrow \) \( (\forall P. \) unitary \( P \) \( \rightarrow \) unitary \( t P \) \( ) \wedge \)
  \( (\forall P Q. \) unitary \( P \) \( \rightarrow \) unitary \( Q \) \( \rightarrow P \vdash Q \rightarrow t P \vdash t Q \) )

lemma nearly-healthy[intro]:
  \[ \wedge P. \) unitary \( P \Longrightarrow unitary \( t P \); \]
  \[ \wedge P Q. \) \( \) unitary \( P \); unitary \( Q \); P \( \vdash Q \] \( \) \( \rightarrow t P \vdash t Q \] \( \) \( \rightarrow \) nearly-healthy t
by ( simp add: nearly-healthy-def )

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

```
[ nearly-healthy t; P ⊢ Q; unitary P; unitary Q ] ⇒ t P ⊢ t Q
by(simp add:nearly-healthy-def)
```

**Lemma** nearly-healthy-unitaryD[dest]:

```
[ nearly-healthy t; unitary P ] ⇒ unitary (t P)
by(simp add:nearly-healthy-def)
```

**Lemma** healthy-nearly-healthy[dest]:

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

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

#### 3.2.3 Sublinearity

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

**Definition** sublinear ::

\( (('s ⇒ real) ⇒ ('s ⇒ real)) ⇒ bool \)

**Where**

```
sublinear t = (\forall a b c P Q s. (sound P ∧ sound Q ∧ 0 ≤ a ∧ 0 ≤ b ∧ 0 ≤ c)
⇒
    a * t P s + b * t Q s ⊓ c
≤ t (λs`. a * P s` + b * Q s` ⊓ c) s)
```

**Lemma** sublinearI[intro]:

```
[ \forall a b c P Q s. [ sound P; sound Q; 0 ≤ a; 0 ≤ b; 0 ≤ c ] ⇒
    a * t P s + b * t Q s ⊓ c ≤
    t (λs`. a * P s` + b * Q s` ⊓ c) s ] ⇒ sublinear t
by(simp add:sublinear-def)
```

**Lemma** sublinearD[dest]:

```
[ sublinear t; sound P; sound Q; 0 ≤ a; 0 ≤ b; 0 ≤ c ] ⇒
    a * t P s + b * t Q s ⊓ c ≤
    t (λs`. a * P s` + b * Q s` ⊓ c) s
by(simp add:sublinear-def)
```

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

definition sub-add ::
  (('s ⇒ real) ⇒ ('s ⇒ real)) ⇒ bool

where

  sub-add t ←→ (∀ P Q s. (sound P ∧ sound Q) →
              t P s + t Q s ≤ t (λs′. P s′ + Q s′) s)

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

Here P is an affine function (expectation) real ⇒ 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 ∩ 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 ∩ preserve convexity. A probabilistic choice will also preserve it. The preservation of convexity is a property of sub-additive transformers.
that respect scaling. Note the form of the definition of convexity:

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

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]:

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

by(simp add:sub-add-def)

lemma sub-addI2:

$$\begin{array}{c}
\forall P, Q. [\text{sound } P; \text{sound } Q] \Rightarrow \\
\lambda s. t P s + t Q s \vdash t (\lambda s. P s + Q s) s \Rightarrow \text{sub-add } t
\end{array}$$

by(auto)

lemma sub-addD[dest]:

$$\begin{array}{c}
\forall \text{sub-add } t; \text{sound } P; \text{sound } Q \Rightarrow \\
t P s + t Q s \leq t (\lambda s'. P s' + Q s') s
\end{array}$$

by(simp add:sub-add-def)

lemma equiv-sub-add:

$$\begin{array}{c}
\text{fixes } t :: (s \Rightarrow \text{real}) \Rightarrow \text{real} \\
\text{assumes } eq : \text{equiv-trans } t u \\
\text{and } sa : \text{sub-add } t \\
\text{shows } \text{sub-add } u
\end{array}$$

proof

fix $P :: s \Rightarrow \text{real}$ and $Q :: s \Rightarrow \text{real}$ and $s :: s$

assume $sP: \text{sound } P$ and $sQ: \text{sound } Q$

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 $\text{sound } (\lambda s. P s + Q s)$ by(auto)
with \( eq \) 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.

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 \( \theta \leq t P s \) and \( \theta \leq t Q s \) by\((auto)\)
  hence \( \theta \leq t P s + t Q s \) by\((auto)\)
  hence \( ... = ... \ominus \theta \) by\((simp)\)
  also from \( sP \) \( sQ \)
  have \( ... \leq t (\lambda s. P s + Q s \ominus \theta) s \)
  by\((rule \ sublinearD[OF \ slt, \ where \ a=1 \ and \ b=1 \ and \ c=0, \ simplified])\)
  also \{ 
  from \( sP \) \( sQ \) have \( \forall s. \theta \leq P s \) and \( \forall s. \theta \leq Q s \) by\((auto)\)
  hence \( \forall s. \theta \leq P s + Q s \) by\((auto)\)
  hence \( t (\lambda s. P s + Q s \ominus \theta) s = t (\lambda s. P s + Q s) s \)
  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:

lemma \( standard-negate \):
  assumes \( ht \): \( \text{healthy} \ t \)
  and \( sat \): \( \text{sub-add} \ t \)
  shows \( t \langle P \rangle s + t \langle \mathcal{N} P \rangle s \leq 1 \)
proof
  from \( sat \) have \( t \langle P \rangle s + t \langle \mathcal{N} P \rangle s \leq t (\lambda s. \langle P \rangle s + \langle \mathcal{N} P \rangle s) s \) 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 51

lemma sub-add-sum:
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::'
from ht have sound (t (\lambda s. 0)) by(auto)
thus 0 \leq t (\lambda s. 0) s by(auto)

fix F::'a set and x::'
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 x a. P x xa + (\sum y\in F. P y xa)) s
by(auto intro':sub-addD[OF sat] sum-sound)
finally
show t (P x) s + (\sum y\in F. t (P y) s) \leq t (\lambda x a. P x xa + (\sum y\in F. P y xa)) s
qed

lemma sub-add-guard-split:
fixes t::'s:finite trans and P::'s expect and s::'
assumes sat: sub-add t
and ht: healthy t
and sP: sound P
shows (\sum y\in \{s. G s\}, P y * t « \lambda z. z = y » s) +
(\sum y\in \{s. \neg G s\}, P y * t « \lambda z. z = y » s) \leq t P s
proof
have \{s. G s\} \cap \{s. \neg G s\} = {} by(blast)
hence (\sum y\in \{s. G s\}, P y * t « \lambda z. z = y » s) +
(\sum y\in \{s. \neg G s\}, P y * t « \lambda z. z = y » s) =
(\sum y\in (\{s. G s\} \cup \{s. \neg G s\}), P y * t « \lambda z. z = y » s)
by(auto intro: sum.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\}), P y * t « \lambda z. z = y » s) =
(\lambda x. \sum y\in UNIV. P y * (t (\lambda x. «\lambda z. z = y» x) s)
by(simp)
}
also {
from sP have \( \forall y. \ 0 \leq P \ y \) by(auto)
with healthy-scalingD[OF ht]
have \( (\lambda x. \sum y \in \text{UNIV}. \ P \ y * \ « \lambda z. \ z = y \» \ x) \ s = \)
(\( \lambda x. \sum y \in \text{UNIV}. \ t \ (\lambda x. \ P \ y * \ « \lambda z. \ z = y \» \ x) \ s \))
by(simp add:scalingD)
}
also { sP from have (\( \in sP\))
also from sP have \( \vdash P \)  \( \vdash P \)
also from sP have \( \vdash P \)  \( \vdash P \)
finally have \( (\lambda x. \sum y \in \text{UNIV}. \ P \ y * \ « \lambda z. \ z = y \» \ x) \)
moreover have sound \( (\lambda x. \sum y \in \text{UNIV}. \ P \ y * \ « \lambda z. \ z = y \» \ x) \)
proof(intro introI bounded-byI nnegI sum-nonneg ballI)
fix x
from leP have \( \sum y \in \text{UNIV}. \ P \ y * \ « \lambda z. \ z = y \» \ x) \)
also from sP have \( \sum y \in \text{UNIV}. \ P \ y * \ « \lambda z. \ z = y \» \ x) \)
finally show \( \sum y \in \text{UNIV}. \ P \ y * \ « \lambda z. \ z = y \» \ x) \)
fix y
from sP show \( 0 \leq P \ y * \ « \lambda z. \ z = y \» \ x) \)
by(auto intro:mult-nonneg-nonneg)
qed
ultimately have t \( (\lambda x. \sum y \in \text{UNIV}. \ P \ y * \ « \lambda z. \ z = y \» \ x) \ s \leq t \ P \ s \)
using sP by(auto intro:le-funD[OF mono-transD, OF healthy-monoD, OF
hl])
}
finally show \( \vdash P \)
qed

Sub-distributivity

definition sub-distrib ::
\( (('s \Rightarrow \text{real}) \Rightarrow ('s \Rightarrow \text{real})) \Rightarrow \text{bool} \)
where
\( \text{sub-distrib} \ t \leftarrow \ (\forall P \ s. \ \text{sound} \ P \longrightarrow t \ P \ s \odot 1 \leq t \ (\lambda s'. \ P \ s' \odot 1) \ s) \)

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 \ \text{sub-distrib} \ t
by(simp add:sub-distrib-def)
3.2. EXPECTATION TRANSFORMERS

**Lemma sub-distrib12:**

\[ \forall P. \text{sound } P \Rightarrow \lambda s. t \ P \ s \ominus 1 \vdash t (\lambda s. P \ s \ominus 1) \] \Rightarrow \text{sub-distrib } t

*by (auto)*

**Lemma sub-distribD[dest]:**

\[ \text{sub-distrib } t; \text{sound } P \] \Rightarrow \text{t } P \ s \ominus 1 \leq t (\lambda s'. P \ s' \ominus 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 \( \text{eq} \): \( \text{equiv-trans } t \ u \)

and \( \text{sd} \): \( \text{sub-distrib } t \)

shows \( \text{sub-distrib } u \)

*proof*

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

assume \( sP :: \text{sound } P \)

moreover have \( \text{sound } (\lambda -. 0) \) *by (auto)*

ultimately show \( t \ P \ s \ominus 1 \leq u (\lambda s. P \ s \ominus 1) \ s \)

*by (rule sublinearD[OF \( \text{slt} \), 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 \( \text{slt} :: \text{sublinear } t \)

shows \( \text{sub-distrib } t \)

*proof*

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

assume \( sP :: \text{sound } P \)

moreover have \( \text{sound } (\lambda -. 0) \) *by (auto)*

ultimately show \( t \ P \ s \ominus 1 \leq t (\lambda s. P \ s \ominus 1) \ s \)

*by (rule sublinearD[OF \( \text{slt} \), 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 \( \text{sd} :: \text{sub-distrib } t \) and \( \text{sat} :: \text{sub-add } t \) and \( \text{ht} :: \text{healthy } t \)

shows \( \text{sublinear } t \)

*proof*

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

and \( a :: \text{real} \) and \( b :: \text{real} \) and \( c :: \text{real} \)

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

and \( \text{nma} :: 0 \leq a \) and \( \text{nmb} :: 0 \leq b \) and \( \text{nnc} :: 0 \leq c \)
from ht sP sQ mna mnb
have saP: sound (λs. a * P s) and staP: sound (λs. a * t P s)
and sbQ: sound (λs. b * Q s) and stbQ: sound (λs. b * t Q s)
by(auto intro:sc-sound)
hence sabPQ: sound (λs. a * P s + b * Q s)
and stabPQ: sound (λs. a * t P s + b * t Q s)
by(auto intro:sound-sum)

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

show a * t P s + b * t Q s ⊓ c ≤ t (λs. a * P s' + b * Q s' ⊓ c) s
proof(cases c = 0)
case True note z = this
from stabPQ have \( \land s. 0 \leq a * t P s + b * t Q s \) by(auto)
moreover from sabPQ
have \( \land s. 0 \leq a * P s + b * Q s \) by(auto)
ultimately show \( \Rightarrow \)thesis by(simp add:z notm)

next
case False note nz = this
from nz and nnc have nni: 0 ≤ inverse c by(auto)

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

from nz
have a * t P s + b * t Q s ⊓ c =
\( (c * inverse c) * a * t P s + 
(c * inverse c) * b * t Q s ⊓ c \)
by(simp)
also
have ... = c * (inverse c * a * t P s) +
c * (inverse c * b * t Q s) ⊓ c
by(simp add:field-simps)
also from nnc
have ... = c * (inverse c * a * t P s + inverse c * b * t Q s ⊓ 1)
by(simp add:distrib-left tminus-left-distrib)
3.2. EXPECTATION TRANSFORMERS

\begin{align*}
\text{also } & \{ \\
\text{have } & X: \land s. \ (\text{inverse } c \ast a) \ast t \ P \ s + (\text{inverse } c \ast b) \ast t \ Q \ s = \\
& \text{inverse } c \ast (a \ast t \ P \ s + b \ast t \ Q \ s) \ \text{by(simp add: divide-simps)} \\
\text{also from } & \text{nni and notm} \\
\text{have } & \text{inverse } c \ast (a \ast t \ P \ s + b \ast t \ Q \ s) \leq \\
& \text{inverse } c \ast (t \ (\lambda s. \ a \ast P \ s + b \ast Q \ s) \ s) \\
& \ \text{by(blast intro:mult-left-mono)} \\
\text{also from } & \text{nni ht subPQ} \\
\text{have } & (\text{inverse } c \ast a) \ast t \ P \ s + (\text{inverse } c \ast b) \ast t \ Q \ s \ominus 1 \leq \\
& t \ (\lambda s. \ (\text{inverse } c \ast a) \ast P \ s + (\text{inverse } c \ast b) \ast Q \ s) \ s \ominus 1 \\
& \ \text{by(rule tminus-left-mono)} \\
\text{also } & \{ \\
\text{from } & \text{sdt si} \\
\text{have } & t \ (\lambda s. \ (\text{inverse } c \ast a) \ast P \ s + (\text{inverse } c \ast b) \ast Q \ s) \ s \ominus 1 \leq \\
& t \ (\lambda s. \ (\text{inverse } c \ast a) \ast P \ s + (\text{inverse } c \ast b) \ast Q \ s \ominus 1) \ s \\
& \ \text{by(blast)} \\
\} \\
\text{finally } & \text{have } c \ast (\text{inverse } c \ast a \ast t \ P \ s + \text{inverse } c \ast b \ast t \ Q \ s \ominus 1) \leq \\
& c \ast t \ (\lambda s. \ \text{inverse } c \ast a \ast P \ s + \text{inverse } c \ast b \ast Q \ s \ominus 1) \ s \\
& \ \text{using nnc by(blast intro:mult-left-mono)} \\
\} \\
\text{also from } & \text{nnc ht sim} \\
\text{have } & c \ast t \ (\lambda s. \ \text{inverse } c \ast a \ast P \ s + \text{inverse } c \ast b \ast Q \ s \ominus 1) \ s \\
& = t \ (\lambda s. \ (c \ast \text{inverse } c \ast a \ast P \ s + \text{inverse } c \ast b \ast Q \ s \ominus 1)) \ s \\
& \ \text{by(simp add:scalingD healthy-scalingD)} \\
\text{also from } & \text{nnc} \\
\text{have } & ... = t \ (\lambda s. \ (c \ast \text{inverse } c) \ast a \ast P \ s + \text{inverse } c \ast b \ast Q \ s \ominus c) \ s \\
& \ \text{by(simp add:distrib-left tminus-left-distrib)} \\
\text{also have } & ... = t \ (\lambda s. \ (c \ast \text{inverse } c) \ast a \ast P \ s + \\
& (c \ast \text{inverse } c) \ast b \ast Q \ s \ominus c) \ s \\
& \ \text{by(simp add:field-simps)} \\
\text{finally } & \text{show } 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 \\
& \ \text{using nz by(simp)} \\
\end{align*}

\text{qed
}

\text{qed
}

\textbf{Sub-conjunctivity}

\textbf{definition}

\textit{sub-conj} :: \((\forall s \Rightarrow \text{real}) \Rightarrow \forall s' \Rightarrow \text{real}) \Rightarrow \text{bool}

\textbf{where}

\textit{sub-conj} \ t \equiv \forall P \ Q. \ (\text{sound } P \land \text{sound } Q) \longrightarrow \\
\text{t} \ P \ \&\& \ t Q \vdash \ t (P \ \&\& \ Q)

\textbf{Sub-conjunctivity}
lemma sub-conj[intro]:
\[ \lambda P. Q. \ [ \ [ \text{sound } P; \text{sound } Q ] ] \implies\ t \ P \ & \ & t \ Q \implies t \ (P \ & \ & Q) \implies \text{sub-conj } t\]
unfolding sub-conj-def by(simp)

lemma sub-conjD[dest]:
\[ \ [ \text{sub-conj } t; \text{sound } P; \text{sound } Q ] ] \implies t \ P \ & \ & t \ Q \implies t \ (P \ & \ & Q)\]
unfolding sub-conj-def by(simp)

lemma sub-conj-wp-twice:
fixes f :: `'s ⇒ (′s ⇒ real) ⇒ ′s ⇒ real
assumes all: ∀s. sub-conj (f s)
shows sub-conj (λP s. f s P s)
proof (rule sub-conjI, rule le-funI, unfolding exp-conj-def pconj-def)
fix P::′s ⇒ real and Q::′s ⇒ real and s
assume sP: sound P and sQ: sound Q
have ((λs. f s P s) & & (λs. f s Q s)) s = (f s P & & f s Q) s
by(simp add:exp-conj-def)
also { from all have sub-conj (f s) by(blast)
  with sP and sQ have (f s P & & f s Q) s ≤ f s (P & & Q) s
  by(blast)
}
finally show ((λs. f s P s) & & (λs. f s Q s)) s ≤ f s (P & & Q) s .
qed

Sublinearity implies sub-conjunctivity:
lemma sublinear-sub-conj:
fixes t::′s ⇒ real ⇒ ′s ⇒ real
assumes slt: sublinear t
shows sub-conj t
proof (rule sub-conjI, rule le-funI, unfolding exp-conj-def pconj-def)
fix P::′s ⇒ real and Q::′s ⇒ real and s::′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:
[ equiv-trans t w; sublinear t; healthy t ] \implies \text{sublinear } u
by(intro:sd-sa-sublinear healthy-equivI
  dest:equiv-sub-distrib equiv-sub-add
  sublinear-sub-distrib sublinear-subadd
  healthy-feasibleD)
3.2. EXPECTATION TRANSFORMERS

3.2.4 Determinism

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

Additivity

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

definition

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

where

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

lemma \( \text{additive-D} \):

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

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

lemma \( \text{additiveI} \) [intro]:

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

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

Additivity is strictly stronger than sub-additivity.

lemma \( \text{additive-sub-add} \):

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

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

The additivity property extends to finite summation.

lemma \( \text{additive-sum} \):

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

\text{assumes additive: additive } t \text{ and healthy: healthy } t \text{ and finite: finite } S \text{ and } sPz: \ \( \land z. \text{sound } (P z) \)

\text{shows } t (\lambda x. \sum y \in S. P y x) = (\lambda x. \sum y \in S. t (P y) x) \)

proof \( \text{rule finite-induct, simp-all add:assms} \)

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

\text{assume finT: finite } T \text{ and IH: } t (\lambda x. \sum y \in T. P y x) = (\lambda x. \sum y \in T. t (P y) x) \)

from additive \( sPz \)

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

also from IH
we can group the states in the linear form, to split on the value of a predicate

\[
\text{have } ... = (\lambda x. \ t \ (P \ z) \ x + (\sum \ y \in T. \ t \ (P \ y) \ x))
\]

by (simp)

finally show \[
(\lambda x. \ t \ (P \ z) \ x + (\sum \ y \in T. \ t \ (P \ y) \ x)) = \\
(\lambda x. \ t \ (P \ z) \ x + (\sum \ y \in T. \ t \ (P \ y) \ x))
\]

qed

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

**Lemma additive-delta-split:**

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

assumes additive: additive \(t\)

and \(h\): healthy \(t\)

and \(s\): sound \(P\)

shows \(t \ P \ x = (\sum \ y \in UNIV. \ P \ y \ast \ t \ (\lambda z. \ z = y \Rightarrow x))\)

**proof**

- have \(\forall x. (\sum \ y \in UNIV. \ P \ y \ast (\lambda z. \ z = y \Rightarrow x)) = \)
  \((\sum \ y \in UNIV. \ \text{if } y = x \ \text{then } P \ y \ \text{else } 0)\)

  by (rule sum.cong) auto

also have \(\forall x. \ x = P \ x\)

by (simp add:sum.delta)

finally

have \(t \ P \ x = t \ (\lambda x. \ \sum \ y \in UNIV. \ P \ y \ast (\lambda z. \ z = y \Rightarrow x)) \ x\)

by (simp)

also \(
\begin{cases}
  \text{from } s\:P \ \text{have } \forall z. \ (\lambda a. \ P \ z \ast (\lambda za. \ za = z \Rightarrow a)) \\
  \text{by(auto intro!:mult-sound)}
  \\
  \text{hence } t \ (\lambda x. \ \sum \ y \in UNIV. \ P \ y \ast (\lambda z. \ z = y \Rightarrow x)) \ x = \\
  (\sum \ y \in UNIV. \ t \ (\lambda x. \ P \ y \ast (\lambda z. \ z = y \Rightarrow x)) \ x)
  \\
  \text{by(subst additive-sum, simp-all add:assms)}
\end{cases}
\)

also from \(s\):P

have \((\sum \ y \in UNIV. \ t \ (\lambda x. \ P \ y \ast (\lambda z. \ z = y \Rightarrow x)) \ x) = \\
(\sum \ y \in UNIV. \ P \ y \ast t \ (\lambda z. \ z = y \Rightarrow x))\)

by(subst scalingD[OF healthy-scalingD, OF ht], auto)

finally show \(t \ P \ x = (\sum \ y \in UNIV. \ P \ y \ast t \ (\lambda z. \ z = y \Rightarrow x))\).

qed

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

**Lemma additive-guard-split:**

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

assumes additive: additive \(t\)

and \(h\): healthy \(t\)

and \(s\): sound \(P\)

shows \(t \ P \ x = (\sum \ y \in \{s. \ G \ s\}. \ P \ y \ast t \ (\lambda z. \ z = y \Rightarrow x)) + \\
(\sum \ y \in \{s. \ \neg \ G \ s\}. \ P \ y \ast t \ (\lambda z. \ z = y \Rightarrow x))\)

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

Maximality

definition maximal :: \( \langle \langle 'a \Rightarrow \text{real} \rangle \Rightarrow 'a \Rightarrow \text{real} \rangle \Rightarrow \text{bool} \)
where
maximal \( t \equiv \forall \, c. \, \theta \leq c \longrightarrow t \, \langle \lambda -. \, c \rangle = (\lambda -. \, c) \)

lemma maximalI[intro]:
[ \( \forall \, c. \, \theta \leq c \Rightarrow t \, \langle \lambda -. \, c \rangle = (\lambda -. \, c) \) ] \( \Rightarrow \) maximal \( t \)
by (simp add: maximal-def)

lemma maximalD[dest]:
[ maximal \( t \); \( \theta \leq c \) ] \( \Rightarrow \) t \( \langle \lambda -. \, c \rangle = (\lambda -. \, c) \)
by (simp add: maximal-def)

A transformer that is both additive and maximal is deterministic:
definition determ :: \( \langle \langle 'a \Rightarrow \text{real} \rangle \Rightarrow 'a \Rightarrow \text{real} \rangle \Rightarrow \text{bool} \)
where
determ \( t \equiv \text{additive} \, t \land \text{maximal} \, t \)

lemma determI[intro]:
[ additive \( t \); 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** det-negate:
**assumes** det: detm 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 detm detm-additiveD)
**also** {
  **have** ∃s. «P» s + «N P» 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 detm detm-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:

\[
[ P \vdash t Q; \text{healthy } t; \text{sound } R; \ Q \vdash R; \text{sound } Q ] \Rightarrow P \vdash t R
\]
**by** (blast intro: entails-trans)
3.2. EXPECTATION TRANSFORMERS

**Lemma** entails-weaken-pre: 

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

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

and \( sQ : \text{sound} Q \) and \( \text{pos: } 0 \leq c \)

shows \((\lambda s . c \cdot P s) \vdash t (\lambda s . c \cdot Q s)\)

**Proof** (rule le-funI)

fix \( s \)

from \( \text{pos and wp have } c \cdot P s \leq c \cdot t Q s \)

by (auto intro: mult-left-mono)

with \( sQ \) \( h \) show \( c \cdot P s \leq t (\lambda s . c \cdot Q s) s \)

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 \( h : \text{healthy} t \) and \( wP : \forall s . P s \implies 1 \leq t \langle Q \rangle s \)

shows \( \langle P \rangle \vdash t \langle Q \rangle \)

**Proof** (rule entailsI)

fix \( s \)

assume \( P s \)

hence \( 1 = \langle P \rangle s \) by (simp)

also from \( wp \) have \( \ldots \leq t \langle Q \rangle s \) by (auto)

finally show \( 1 \leq t \langle Q \rangle s \).

qed
Predicate conjunction behaves as expected:

**lemma** *conj-post*:

\[
[P \vdash t \langle s, Q s \land R s \rangle; \text{healthy } t] \implies P \vdash t \langle Q s \rangle
\]

by (blast intro: entails-strengthen-post implies-entails)

Similar to \[\text{healthy } ?t; \bigwedge s. ?P s \implies 1 \leq ?t \langle ?Q s \rangle \implies ?P \vdash ?t \langle ?Q \rangle\], 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 *expconj-def*

proof (rule entailsI)

fix \(s\) show \(\langle P \rangle s \land Q s \leq t R s\)

proof (cases \(P s\))

case True

moreover from \(uQ\) have \(\theta \leq Q s\) by (auto)

ultimately show \(?thesis\) by (simp add: pconj-lone \(wP\))

next

case False

moreover from \(uQ\) have \(Q s \leq 1\) by (auto)

ultimately show \(?thesis\) using *assms* by (auto)

qed

qed

end

3.3 Induction

**theory** *Induction*

**imports** *Expectations Transformers*

**begin**

3.3.1 The Lattice of Expectations

Defining recursive (or iterative) programs requires us to reason about fixed points on the semantic objects, in this case expectations. The complication here, compared to the standard Knaster-Tarski theorem (for example, as shown in *HOL.Inductive*), is that we do not have a complete lattice. Finding a lower bound is easy (it’s \(\lambda- \, \theta::'b\)), 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.
3.3. INDUCTION

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 \( t \). In practice, \( t \) is generally doubly healthy, that is \( \forall P. \text{sound } P \rightarrow \text{healthy } (t P) \) and \( \forall Q. \text{sound } Q \rightarrow \text{healthy } (\lambda P. t P Q) \). Thus by feasibility, \( t P Q \) must be bounded by \( \text{bound-of } P \). 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 \( \lambda-. \text{ bound-of } P \).

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

**Lemma** Inf-exp-lower:
[ \( P \in S; \forall P \in S. \text{nneg } P \] \( \rightarrow \) Inf-exp S ≤ 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 ≤ 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 \( \lambda s. \text{Sup } \{ f s | f. f \in S \} \))

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

**Lemma** Sup-exp-least:
[ \( \forall P \in S. P \leq Q; \text{nneg } Q \] \( \rightarrow \) Sup-exp S ≤ Q
unfolding Sup-exp-def
by(cases S=\{\}, auto intro le-funI cSup-least)

**Lemma** Sup-exp-sound:
assumes sS: \( \forall P. P \in S \Rightarrow \text{sound } P \)
and bS: \( \forall P. P \in S \Rightarrow \text{bounded-by } b P \)
shows sound (Sup-exp S)
proof(cases S=\{\}, simp add:Sup-exp-def, blast,
intro soundI2 bounded-byI2 nnegI2)
assume neS: S \( \neq \{\}\)
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 ⊢ \( \lambda s. b \)
by(auto intro:Sup-exp-least)

from nP have \( \lambda s. 0 \neq P \) by(auto)
also from Pin bS have P \( \neq \) Sup-exp S
by (auto intro: Sup-exp-upper)
finally show \( \lambda s. \emptyset \vdash \text{Sup-exp} S \).
qed

definition lfp-exp :: 's trans \Rightarrow 's expect
where lfp-exp \( t \) = Inf-exp \( \{ P. \text{sound } P \land t \vdash P \} \)

lemma lfp-exp-lowerbound:
[ \[ t \vdash P; \text{sound } P \] \Rightarrow lfp-exp \( t \) \vdash P ]
unfolding lfp-exp-def by (auto intro: Inf-exp-lower)

lemma lfp-exp-greatest:
[ \[ \forall P. \[ t \vdash P; \text{sound } P \] \Rightarrow Q \vdash P; \text{sound } Q; t \vdash R; \text{sound } R \] \Rightarrow Q \vdash lfp-exp \( t \) ]
unfolding lfp-exp-def by (auto intro: Inf-exp-greatest)

lemma feasible-lfp-exp-sound:
assumes fR: \( t \vdash R \) and sR: sound \( R \)
shows sound (lfp-exp \( t \))
proof (intro soundI2 bounded-byI2 nnegI2, auto intro!: lfp-exp-lowerbound lfp-exp-greatest)
qed

lemma lfp-exp-bound:
(\( \forall P. \text{unitary } P \Rightarrow \text{unitary } (t \vdash P) \) \Rightarrow \text{bounded-by } 1 (lfp-exp \( t \))
by (auto intro: lfp-exp-lowerbound)

lemma lfp-exp-unitary:
(\( \forall P. \text{unitary } P \Rightarrow \text{unitary } (t \vdash P) \) \Rightarrow \text{unitary } (lfp-exp \( t \))
proof (intro unitaryI [OF lfp-exp-sound lfp-exp-bound], simp-all)
assume IH: \( \forall P. \text{unitary } P \Rightarrow \text{unitary } (t \vdash P) \)
have unitary (\( \lambda s. 1 \)) by (auto)
with IH have unitary (\( t (\lambda s. 1) \)) by (auto)
thus \( t (\lambda s. 1) \vdash \lambda s. 1 \) by (auto)
show sound (\( \lambda s. 1 \)) by (auto)
qed

lemma lfp-exp-lemma2:
fixes t::'s trans
assumes st: \( \forall P. \text{sound } P \Rightarrow \text{sound } (t \vdash P) \)
and mt: mono-trans \( t \)
3.3. INDUCTION

and \( fR: t R \vdash R \) and \( sR: \text{sound } R \)
shows \( t (\text{lfp-exp } t) \leq \text{lfp-exp } t \)

proof (rule lfp-exp-greatest[of \( t \), OF \( \cdot fR sR \)])
from \( fR \) \( sR \) show sound \( t (\text{lfp-exp } t) \) by (auto intro: lfp-exp-sound \( st \))

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

qed

lemma lfp-exp-lemma3:
assumes \( st: \bigwedge P. \text{sound } P \Longrightarrow \text{sound } (t P) \)
and \( mt: \text{mono-trans } t \)
and \( fR: t R \vdash R \) and \( sR: \text{sound } R \)
shows lfp-exp \( t \leq t (\text{lfp-exp } t) \)
by (iprover intro: lfp-exp-lowerbound lfp-exp-sound lfp-exp-lemma2 assms mono-transD \[ OF mt \])

lemma lfp-exp-unfold:
assumes \( nt: \bigwedge P. \text{sound } P \Longrightarrow \text{sound } (t P) \)
and \( mt: \text{mono-trans } t \)
and \( fR: t R \vdash R \) and \( sR: \text{sound } R \)
shows lfp-exp \( t = t (\text{lfp-exp } t) \)
by (iprover intro: antisym lfp-exp-lemma2 lfp-exp-lemma3 assms)

definition gfp-exp :: 's trans ⇒ 's expect
where gfp-exp \( t = \text{Sup-exp } \{ P. \text{unitary } P \land P \leq t P \} \)

lemma gfp-exp-upperbound:
\[ \{ P \leq t P; \text{unitary } P \} \Longrightarrow P \leq \text{gfp-exp } t \]
by (auto simp: gfp-exp-def intro: Sup-exp-upper)

lemma gfp-exp-least:
\[ \bigwedge P. \{ P \leq t P; \text{unitary } P \} \Longrightarrow P \leq Q; \text{unitary } Q \} \Longrightarrow \text{gfp-exp } t \leq Q \]
unfolding gfp-exp-def by (auto intro: Sup-exp-least)

lemma gfp-exp-bound:
\( (\bigwedge P. \text{unitary } P \Longrightarrow \text{unitary } (t P)) \Longrightarrow \text{bounded-by } 1 (\text{gfp-exp } t) \)
unfolding gfp-exp-def
by (rule bounded-byI2 [OF Sup-exp-least], auto)

lemma gfp-exp-nneg[iff]:
nneg (gfp-exp \( t \))
proof (intro nnegI2, simp add: gfp-exp-def, cases)
assume empty: \( \{ P. \text{unitary } P \land P \vdash t P \} = \{ \} \)
show \( \lambda s. 0 \vdash \text{Sup-exp } \{ P. \text{unitary } P \land P \vdash t P \} \)
CHAPTER 3. SEMANTIC STRUCTURES

by(simp only:empty Sup-exp-def, auto) next
assume \{ P. unitary P \land P \vdash t P \} \neq \{ \}
then obtain Q where \( Q \in \{ P. unitary P \land P \vdash t P \} \) by(auto)
hence \( \lambda s. 0 \vdash Q \) by(auto)
also from \( Q \) have \( Q \vdash \text{Sup-exp} \{ P. unitary P \land P \vdash t P \} \)
by(auto intro:Sup-exp-upper)
finally show \( \lambda s. 0 \vdash \text{Sup-exp} \{ P. unitary P \land P \vdash t P \} \).
qed

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

lemma gfp-exp-lemma2:
assumes \( ft: \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) \)
and \( mt: \forall P Q. \left[ \text{unitary } P; \text{unitary } Q; P \vdash Q \right] \Rightarrow t P \vdash t Q \)
shows \( gfp-exp t \leq t (gfp-exp t) \)
proof(rule gfp-exp-least)
show \( \text{unitary } (t (gfp-exp t)) \) by(auto intro:gfp-exp-unitary ft)
fix \( P \)
assume \( fp: P \leq t P \) and \( uP: \text{unitary } P \)
with \( ft \) have \( P \leq gfp-exp t \) by(auto intro:gfp-exp-upperbound)
with \( uP \) gfp-exp-unitary \( ft \)
have \( t P \leq t (gfp-exp t) \) by(blast intro: mt)
with \( fp \) show \( P \leq t (gfp-exp t) \) by(auto)
qed

lemma gfp-exp-lemma3:
assumes \( ft: \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) \)
and \( mt: \forall P Q. \left[ \text{unitary } P; \text{unitary } Q; P \vdash Q \right] \Rightarrow t P \vdash t Q \)
shows \( t (gfp-exp t) \leq gfp-exp t \)
by(iprover intro:gfp-exp-upperbound unitary-sound
\quad gfp-exp-unitary gfp-exp-lemma2 assms)

lemma gfp-exp-unfold:
\( (\forall P. \text{unitary } P \Rightarrow \text{unitary } (t P)) \Rightarrow (\forall P Q. \left[ \text{unitary } P; \text{unitary } Q; P \vdash Q \right] \Rightarrow t P \vdash t Q) \Rightarrow \)
\( gfp-exp t = t (gfp-exp t) \)
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, healthy transformers do form a complete lattice, where the bottom element is \( \lambda \cdot \cdot. \; 0::'c, \) 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:
\[ t \in S; \forall u \in S. \forall P. \text{sound } P \rightarrow \text{sound } (u P) \] \( \Rightarrow \) le-trans (Inf-trans \( S \)) \( t \)

unfolding Inf-trans-def
by (rule le-transI [OF Inf-exp-lower], blast+)

**lemma** Inf-trans-greatest:
\[ S \neq \{\}; \forall t \in S. \forall P. \text{le-trans } u \ t \] \( \Rightarrow \) le-trans \( u \) (Inf-trans \( S \))

unfolding Inf-trans-def
by (auto intro: le-transI [OF Inf-exp-greatest])

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

**lemma** Sup-trans-upper:
\[ t \in S; \forall u \in S. \forall P. \text{unitary } P \rightarrow \text{unitary } (u P) \] \( \Rightarrow \) le-utrans \( t \) \( (\text{Sup-trans } S ) \)

unfolding Sup-trans-def
by (intro le-utransI [OF Sup-exp-upper], auto intro: unitary-bound)

**lemma** Sup-trans-upper2:
\[ t \in S; \forall u \in S. \forall P. (\text{nneg } P \land \text{bounded-by } b P) \rightarrow (\text{nneg } (u P) \land \text{bounded-by } b (u P)); \]
\[ \text{nneg } P; \text{bounded-by } b P \] \( \Rightarrow \) \( t P \vdash \text{Sup-trans } S P \)

unfolding Sup-trans-def
by (blast intro: Sup-exp-upper)

**lemma** Sup-trans-least:
\[ \forall t \in S. \text{le-utrans } t u; \forall P. \text{unitary } P \rightarrow \text{unitary } (u P) \] \( \Rightarrow \) le-utrans \( (\text{Sup-trans } S ) \) \( u \)

unfolding Sup-trans-def
by (auto intro!: sound-nneg [OF unitary-sound] le-utransI [OF Sup-exp-least])

**lemma** Sup-trans-least2:
\[ \forall t \in S. \forall P. \text{nneg } P \rightarrow \text{bounded-by } b P \rightarrow t P \vdash u P; \]
\[ \forall u \in S. \forall P. (\text{nneg } P \land \text{bounded-by } b P) \rightarrow (\text{nneg } (u P) \land \text{bounded-by } b (u P)); \]
\[ \text{nneg } P; \text{bounded-by } b P; \forall P. [ \text{nneg } P; \text{bounded-by } b P \] \( \Rightarrow \) nneg \( (u P) \] \( \] \( \Rightarrow \) Sup-trans \( S P \vdash u P \)

unfolding Sup-trans-def
by (blast intro!: Sup-exp-least)

**lemma** feasible-Sup-trans:
fixes \( S::'s \text{ trans set} \)
assumes \( fS::\forall t \in S. \text{feasible } t \)
shows feasible \( (\text{Sup-trans } S) \)
proof (cases \( S=\{\} \), simp add: Sup-trans-def Sup-exp-def, blast, intro feasibleI)
fix \( b::\text{real} \) and \( P::'s \text{ expect} \)
assume \( bP::\text{bounded-by } b P \) and \( nP::\text{nneg } P \)
and \( \text{neS:: } S \neq \{\} \)
CHAPTER 3. SEMANTIC STRUCTURES

from \( ncS \) obtain \( t \) where \( \text{tin}: t \in S \) by(auto)
with \( fS \) have \( ft: \text{feasible} \ t \) by(auto)
with \( bP \ nP \) have \( \lambda s. \ 0 \vdash t \) \( P \) by(auto)
also {
  from \( bP \ nP \) have \( \text{sound} \ P \) by(auto)
  with \( \text{tin} fS \) 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 \( fS \ bP \ nP \) have \( \text{bounded-by} b \ (\text{Sup-trans} \ S \ P) \)
  by(auto intro!:bounded-byI2[OF Sup-trans-least2])
qed

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

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

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

lemma \( \text{lfp-trans-sound} \):
fixes \( P Q::\forall s \text{ expect} \)
assumes \( sP::\text{sound} \ P \)
and \( fv::\text{le-trans} \ (T \ v) \ v \)
and \( sv::\bigwedge P. \text{sound} \ P \Rightarrow \text{sound} \ (v \ P) \)
sows \( \text{sound} \ (\text{lfp-trans} \ T \ P) \)
proof(intro soundI2 bounded-byI2 nneqI2)
from \( fv \ sv \) have \( \text{le-trans} \ (\text{lfp-trans} \ T) \ v \)
  by(iiprover intro:lfp-trans-lowerbound)
with \( sP \) have \( \text{lfp-trans} \ T \ P \vdash v \ P \) by(auto)
also {
  from \( sv \ sP \) have \( \text{sound} \ (v \ P) \) by(iiprover)
    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 \( \text{le-trans} \ (\lambda P s. \ 0) \ (\text{lfp-trans} \ T) \)
proof(intro lfp-trans-greatest)
3.3. INDUCTION

\[
\begin{align*}
\text{fix } t :: \prime \text{'s trans} & \\
\text{assume } \lambda P. \text{sound } P \implies \text{sound } (t P) & \\
\text{hence } \lambda P. \text{sound } P \implies \text{\lam s. 0 } \vdash \text{ t P by(auto)} & \\
\text{thus le-trans } (\lambda P. \text{ s. 0}) t \text{ by(auto)} & \\
\text{next} & \\
\text{fix } P :: \prime \text{'s 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 show } \lambda s. 0 \vdash \text{lfp-trans } T P \text{ by(auto)} & \\
\text{qed} & \\
\text{lemma lfp-trans-unitary:} & \\
\text{fixes } P Q :: \prime \text{'s expect} & \\
\text{assumes } uP. \text{ unitary } P & \\
\text{and } \text{fv: le-trans } (T v) v & \\
\text{and } \text{sv: } \lambda P. \text{sound } P \implies \text{sound } (v P) & \\
\text{and } \text{[T: le-trans } (T (\lambda P. \text{ s. bound-of } P)) (\lambda P. \text{ bound-of } P) & \\
\text{shows } \text{unitary } ((\text{lfp-trans } T P) & \\
\text{proof(rule unitaryI)} & \\
\text{from unitary-sound}[OF uP] \text{ fv sv show } \text{sound } (\text{lfp-trans } T P) & \\
\text{by(rule lfp-trans-sound)} & \\
\text{show bounded-by 1 } ((\text{lfp-trans } T P) & \\
\text{proof(rule bounded-byI2)} & \\
\text{from } \text{[T: le-trans } (\text{lfp-trans } T) (\lambda P. \text{ bound-of } P) & \\
\text{by(auto intro: lfp-trans-lowerbound)} & \\
\text{with } uP \text{ have lfp-trans } T P \vdash \text{\lam s. bound-of } P \text{ by(auto)} & \\
\text{also from } uP \text{ have } \dots \vdash \text{\lam s. 1 by(auto)} & \\
\text{finally show lfp-trans } T P \vdash \text{\lam s. 1} . & \\
\text{qed} & \\
\text{qed} & \\
\text{lemma lfp-trans-lemma2:} & \\
\text{fixes } v :: \prime \text{'s trans} & \\
\text{assumes } \text{mono: } \lambda t u. [\text{le-trans } t u; \lambda P. \text{sound } P \implies \text{sound } (t P); & \\
\lambda P. \text{sound } P \implies \text{sound } (u P)] \implies \text{le-trans } (T t) (T u) & \\
\text{and } \text{nT: } \lambda t P. [\lambda Q. \text{sound } Q \implies \text{sound } (t Q); \text{sound } P ] \implies \text{sound } (T t & \\
P) & \\
\text{and } \text{fv: } \text{le-trans } (T v) v & \\
\text{and } \text{sv: } \lambda P. \text{sound } P \implies \text{sound } (v P) & \\
\text{shows } \text{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 :: \prime \text{'s trans and } P :: \prime \text{'s expect} & \\
\text{assume ft: } \text{le-trans } (T t) t \text{ and st: } \lambda P. \text{sound } P \implies \text{sound } (t P) & \\
\text{hence } \text{le-trans } ((\text{lfp-trans } T) t) t \text{ by(auto intro: lfp-trans-lowerbound)} & \\
\text{with } ft st \text{ have } \text{le-trans } (T ((\text{lfp-trans } T)) (T t) & \\
\text{by(iprove intro:mono lfp-trans-sound fe sv)} & 
\end{align*}
\]
also note \( l_t \)

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

qed

\textbf{lemma} \( lfp-trans-lemma3 \):

\textbf{fixes} \( v \:: \)'s \ trans

\textbf{assumes} \( \text{mono} : \forall t \ u. \ [\ \text{le-trans} \ t \ u; \ \forall P. \ \text{sound} \ P \implies \text{sound} \ (t \ P); \ \] \( \forall P. \ \text{sound} \ P \implies \text{sound} \ (u \ P) \] \implies \text{le-trans} \ (T \ t) \ (T \ u) \)

\( \text{and} \ sT : \forall t \ P. \ [\ [\ \forall Q. \ \text{sound} \ Q \implies \text{sound} \ (t \ Q); \ \text{sound} \ P ] \implies \text{sound} \ (T \ t \ P) \]

\( \text{and} \ fu : \ \text{le-trans} \ (T \ v) \ v \)

\( \text{and} \ sv : \ \forall P. \ \text{sound} \ P \implies \text{sound} \ (v \ P) \)

\textbf{shows} \( \text{le-trans} \ (lfp-trans \ T) \ (T \ (lfp-trans \ T)) \)

\textbf{proof} (\textbf{rule} \( lfp-trans-lowerbound \))

\textbf{fix} \( P :: \)'s \ expect

\textbf{assume} \( sP : \ \text{sound} \ P \)

\textbf{have} \( n1 : \forall P. \ \text{sound} \ P \implies \text{sound} \ (lfp-trans \ T \ P) \)

\( \text{by} (\text{iprover intro} : lfp-trans-sound \ fu \ sv) \)

\textbf{with} \( sP \) \textbf{have} \( n2 : \ \text{sound} \ (lfp-trans \ T \ P) \)

\( \text{by} (\text{iprover intro} : lfp-trans-sound \ fu \ sv \ sT) \)

\textbf{with} \( n1 \ sP \) \textbf{show} \( n3 : \ \text{sound} \ (T \ (lfp-trans \ T) \ P) \)

\( \text{by} (\text{iprover intro} : sT) \)

\textbf{next}

\textbf{show} \( \text{le-trans} \ (T \ (T \ (lfp-trans \ T))) \ (T \ (lfp-trans \ T)) \)

\( \text{by} (\text{rule} \ \text{mono} [\text{OF} \ lfp-trans-lemma2, \ \text{OF} \ \text{mono}],

\ (\text{iprover intro} \ \text{assms} \ lfp-trans-sound)+) \)

\textbf{qed}

\textbf{lemma} \( lfp-trans-unfold \):

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

\textbf{assumes} \( \text{mono} : \forall t \ u. \ [\ \text{le-trans} \ t \ u; \ \forall P. \ \text{sound} \ P \implies \text{sound} \ (t \ P); \ [\ \forall P. \ \text{sound} \ P \implies \text{sound} \ (u \ P) \] \implies \text{le-trans} \ (T \ t) \ (T \ u) \)

\( \text{and} \ sT : \forall t \ P. \ [\ [\ \forall Q. \ \text{sound} \ Q \implies \text{sound} \ (t \ Q); \ \text{sound} \ P ] \implies \text{sound} \ (T \ t \ P) \]

\( \text{and} \ fu : \ \text{le-trans} \ (T \ v) \ v \)

\( \text{and} \ sv : \ \forall P. \ \text{sound} \ P \implies \text{sound} \ (v \ P) \)

\textbf{shows} \( \text{equiv-trans} \ (lfp-trans \ T) \ (T \ (lfp-trans \ T)) \)

\( \text{by} (\text{rule} \ \text{le-trans-antisym},

\ (\text{rule} \ \text{lfp-trans-lemma2}[\text{OF} \ \text{mono}], \ \text{iprover intro} \ \text{assms} \ lfp-trans-sound)+,

\ (\text{rule} \ \text{lfp-trans-lemma2}[\text{OF} \ \text{mono}], \ (\text{iprover intro} \ \text{assms})+) \)

\textbf{definition} \( gfp-trans :: (\)'s \ trans \ \Rightarrow \ 's \ trans \ \Rightarrow \ 's \ trans \)

\textbf{where} \( gfp-trans \ T = \text{Sup-trans} \ \{ \ t. \ (\forall P. \ \text{unitary} \ P \implies \text{unitary} \ (t \ P)) \land \ \text{le-utrans} \ t \ (T \ t) \} \)

\textbf{lemma} \( gfp-trans-upperbound \):

\( \ [\ \text{le-utrans} \ t \ (T \ t); \ \forall P. \ \text{unitary} \ P \implies \text{unitary} \ (t \ P) ] \implies \text{le-utrans} \ t \ (gfp-trans \ T) \)

\textbf{unfolding} \( gfp-trans-def \) \textbf{by}(\text{auto intro} : \text{Sup-trans-upper})
lemma \texttt{gfp-trans-least}:

\[
\[ \forall t. [\text{le-utrans } t (T t); \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P)] \Rightarrow \text{le-utrans } t u; \\
\forall P. \text{unitary } P \Rightarrow \text{unitary } (u P) \] \Rightarrow \\
\text{le-utrans } (\text{gfp-trans } T) u
\]

unfolding \texttt{gfp-trans-def} \texttt{by(auto intro:Sup-trans-least)}

lemma \texttt{gfp-trans-unitary}:

\texttt{fixes} \( P :: \text{''s expect} \)
\texttt{assumes} \( uP :: \text{unitary } P \)
\texttt{shows} \( \text{unitary } (\text{gfp-trans } T P) \)

\texttt{proof(intro unitaryI2 nnegI2 bounded-byI2)}
\texttt{show} \( \lambda s. 1 \)

unfolding \texttt{gfp-trans-def Sup-trans-def}
\texttt{proof(rule Sup-exp-least, clarify)}
\texttt{fix} \( t :: \text{''s trans} \)
\texttt{assume} \( \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) \)
\texttt{with} \( uP \) \texttt{have} \( \text{unitary } (t P) \) \texttt{by(auto)}
\texttt{thus} \( t P \vdash \lambda s. 1 \) \texttt{by(auto)}

\texttt{next}
\texttt{show} \( \text{nneg } (\lambda s. 1 :: \text{real}) \) \texttt{by(auto)}
\texttt{qed}

\texttt{let} \( ?S = \{ t P \mid t \in \{ t. (\forall P. \text{unitary } P \Rightarrow \text{unitary } (t P)) \land \text{le-utrans } t (T t) \} \} \)
\texttt{show} \( \lambda s. 0 \vdash \text{gfp-trans } T P \)

unfolding \texttt{gfp-trans-def Sup-trans-def}
\texttt{proof(cases)}
\texttt{assume} \( \emptyset S = \{ \} \)
\texttt{show} \( \lambda s. 0 \vdash \text{Sup-exp } ?S \)
\texttt{by(simp only:empty Sup-exp-def, auto)}

\texttt{next}
\texttt{assume} \( ?S \neq \{ \} \)
\texttt{then obtain} \( Q \) \texttt{where} \( Q \in \emptyset S \) \texttt{by(auto)}
\texttt{with} \( uP \) \texttt{have} \( \text{unitary } Q \) \texttt{by(auto)}
\texttt{hence} \( \lambda s. 0 \vdash Q \) \texttt{by(auto)}
\texttt{also with} \( uP \) \texttt{Qin have} \( Q \vdash \text{Sup-exp } ?S \)
\texttt{proof(intro Sup-exp-upper, blast, clarify)}
\texttt{fix} \( t :: \text{''s trans} \)
\texttt{assume} \( \forall Q. \text{unitary } Q \Rightarrow \text{unitary } (t Q) \)
\texttt{with} \( uP \) \texttt{show} \( \text{bounded-by } 1 (t P) \) \texttt{by(auto)}
\texttt{qed}

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

lemma \texttt{gfp-trans-lemma2}:
\texttt{assumes} \( \text{mono:} \forall t u. [\text{le-utrans } t u; \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P); \\
\forall P. \text{unitary } P \Rightarrow \text{unitary } (u P)] \Rightarrow \text{le-utrans } (T t) (T u) \)
\texttt{and} \( hT: \forall t P. [\forall Q. \text{unitary } Q \Rightarrow \text{unitary } (t Q); \text{unitary } P] \Rightarrow \text{unitary } (T t P) \)
shows \textit{le-utrans} \((gfp-trans \ T) \ (T \ (gfp-trans \ T))\)

\textbf{proof}(\textit{rule} gfp-trans-least, \textit{simp-all} add\(hT\) gfp-trans-unitary)
fix \(t\)
assume \(fp\): \textit{le-utrans} \(t \ (T \ t)\) and \(ht\): \(\wedge P. \ \text{unitary} \ P \Rightarrow \ \text{unitary} \ (t \ P)\)

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

\textbf{lemma} \(\text{gfp-trans-lemma3}::\)
assumes \(\text{mono}: \forall t \ u. \ [\ \textit{le-utrans} \ t \ u; \ \wedge P. \ \text{unitary} \ P \Rightarrow \ \text{unitary} \ (t \ P);\]
\(\wedge P. \ \text{unitary} \ P \Rightarrow \ \text{unitary} \ (u \ P)] \Rightarrow \ \textit{le-utrans} \ ((T \ t) \ (T \ u))\)
and \(hT:: \forall t \ P. \ [\ \forall Q. \ \text{unitary} \ Q \Rightarrow \ \text{unitary} \ (t \ Q); \ \text{unitary} \ P)] \Rightarrow \ \text{unitary} \ (T \ t \ P)\)
shows \textit{le-utrans} \((T \ (gfp-trans \ T)) \ (gfp-trans \ T)\)
by(\textit{blast intro!:} \(\text{mono} \ \text{gfp-trans-unitary} \ \text{gfp-trans-upperbound} \ \text{gfp-trans-lemma2} \ \text{mono} \ hT)\)

\textbf{lemma} \(\text{gfp-trans-unfold}::\)
assumes \(\text{mono}: \forall t \ u. \ [\ \textit{le-utrans} \ t \ u; \ \wedge P. \ \text{unitary} \ P \Rightarrow \ \text{unitary} \ (t \ P);\]
\(\wedge P. \ \text{unitary} \ P \Rightarrow \ \text{unitary} \ (u \ P)] \Rightarrow \ \textit{le-utrans} \ ((T \ t) \ (T \ u))\)
and \(hT:: \forall t \ P. \ [\ \forall Q. \ \text{unitary} \ Q \Rightarrow \ \text{unitary} \ (t \ Q); \ \text{unitary} \ P)] \Rightarrow \ \text{unitary} \ (T \ t \ P)\)
shows \textit{equiv-utrans} \((gfp-trans \ T) \ (T \ (gfp-trans \ T))\)
using \textit{assms} by(\textit{auto intro!:} \(\text{le-utrans-antisym} \ \text{gfp-trans-lemma2} \ \text{gfp-trans-lemma3}\))

\subsection*{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.

\textbf{lemma} \(\text{gfp-pulldown}::\)
fixes \(P::s\) \textit{expect}
assumes \(\text{tailcall}: \wedge u. \ [\ \textit{le-utrans} \ t \ u; \ \wedge P. \ \text{unitary} \ P \Rightarrow \ \text{unitary} \ (t \ P);\]
\(\forall t. \ [\ \forall Q. \ \text{unitary} \ Q \Rightarrow \ \text{unitary} \ (t \ Q); \ \text{unitary} \ P)] \Rightarrow \ \text{unitary} \ (T \ t \ P)\)
and \(fT:: \forall P Q. \ \text{unitary} \ P \Rightarrow \ \text{unitary} \ Q \Rightarrow \ \text{unitary} \ (t \ P \ Q)\)
and \(mt:: \forall P Q R. \ [\ \text{unitary} \ P; \ \text{unitary} \ Q; \ \text{unitary} \ R; \ Q \vdash R)] \Rightarrow \ t \ P \ Q\)
and \(uP:: \ \text{unitary} \ P\)
and \(\text{mono}: \forall t. \ [\ \textit{le-utrans} \ t \ u; \ \wedge P. \ \text{unitary} \ P \Rightarrow \ \text{unitary} \ (t \ P);\]
\(\forall P. \ \text{unitary} \ P \Rightarrow \ \text{unitary} \ (u \ P)] \Rightarrow \ \textit{le-utrans} \ ((T \ t) \ (T \ u))\)
shows \(\text{gfp-trans} \ T \ P = \ \text{gfp-exp} \ (t \ P)\) (is \(\ ?X \ P = \ ?Y \ P\))
3.3. INDUCTION

proof (rule antisym)
  show \(?X P \leq ?Y P\)
  proof (rule gfp-exp-upperbound)
    from monoT \(T P uP \) have \((\text{gfp-trans } T P) \leq (T (\text{gfp-trans } T P)) P\)
    by (auto intro!: le-utransD[OF gfp-trans-lemma2])
  also from \(uP\) have \((T (\text{gfp-trans } T P)) P = t P (\text{gfp-trans } T P)\) by (rule tailcall)
  finally show \(\text{gfp-trans } T P \vdash t P (\text{gfp-trans } T P)\).
  from \(uP\) gfp-trans-unitary show \((\text{gfp-trans } T P)\) by (auto)
qed

show \(?Y P \leq ?X P\)
proof (rule le-utransD[OF gfp-trans-upperbound], simp_all add:assms)
  show le-utrans \((\lambda a. \text{gfp-exp } (t a))\) \((T (\lambda a. \text{gfp-exp } (t a)))\)
proof (rule le-utransI)
  fix \(Q\) \(\vdash s expect\) unitary \(Q\)
  with \(ft\) have \(\lambda R. \text{unitary } R \Rightarrow \text{unitary } (t Q R)\) by (auto)
    with \(mt\) (OF \(uQ\)) have \(\text{gfp-exp } (t Q) = t Q (\text{gfp-exp } (t Q))\) by (blast intro: gfp-exp-unfold)
  also from \(uQ\) have \(\ldots = T (\lambda a. \text{gfp-exp } (t a))\) \(Q\) by (rule tailcall[symmetric])
  finally show \(\text{gfp-exp } (t Q) \leq T (\lambda a. \text{gfp-exp } (t a))\) \(Q\) by (simp)
qed

fix \(Q\) \(\vdash s expect\) unitary \(Q\)
with \(ft\) have \(\lambda R. \text{unitary } R \Rightarrow \text{unitary } (t Q R)\) by (auto)
thus unitary \((\text{gfp-exp } (t Q))\) by (rule gfp-exp-unitary)
qed

qed

lemma lfp-pulldown:

fixes \(P\) \(\vdash s expect\) and \(t\) \(\vdash s expect\) \(\Rightarrow \) \(s\) trans
  and \(T\) \(\vdash t\) trans \(\Rightarrow \) \(t\) trans

assumes tailcall: \(\lambda u P. \text{sound } P \Rightarrow T u P = t P (u P)\)
  and st: \(\lambda P Q. \text{sound } P \Rightarrow \text{sound } Q \Rightarrow \text{sound } (t P Q)\)
  and mt: \(\lambda P. \text{sound } P \Rightarrow \text{mono-trans } (t P)\)
  and monoT: \(\lambda u. [ \text{le-trans } t u; \lambda P. \text{sound } P \Rightarrow \text{sound } (t P);\)
    \(\lambda P. \text{sound } P \Rightarrow \text{sound } (u P) ] \Rightarrow \text{le-trans } (T t) (t u)\)
  and uT: \(\lambda t P. [ \lambda Q. \text{sound } Q \Rightarrow \text{sound } (t Q); \text{sound } P ] \Rightarrow \text{sound } (T t P)\)

and ft: \(\text{le-trans } (T t v) v\)
  and sv: \(\lambda P. \text{sound } P \Rightarrow \text{sound } (v P)\)
  and sp: \(\text{sound } P\)

shows lfp-trans \(T P = \text{lfp-exp } (t P)\) \((\text{is } ?X P = ?Y P)\)
proof (rule antisym)
  show \(?Y P \leq ?X P\)
proof (rule lfp-exp-lowerbound)
    from \(sP\) have \(t P (\text{lfp-trans } T P) = (T (\text{lfp-trans } T) P\) by (rule tailcall[symmetric])
  also have \((T (\text{lfp-trans } T)) P \leq (\text{lfp-trans } T) P\)
    by (rule le-utransD[OF lfp-trans-lemma2[OF monoT]], (iprover intro:assms)+)
  finally show \(t P (\text{lfp-trans } T P) \leq \text{lfp-trans } T P\).
  from \(sP\) show sound \((\text{lfp-trans } T P)\)
by (iprover intro: lfp-trans-sound assms)
qed

have \( \bigwedge P. \ sound P \implies t P (v P) = T v P \) by (simp add: tailcall)
also have \( \bigwedge P. \ sound P \implies \vdash P \vdash v P \) by (auto intro: le-transD [OF \( f_\ell \)])
finally have \( f_\ell P: \bigwedge P. \ sound P \implies \vdash P (v P) \vdash v P \).
have \( s\ell P: \bigwedge P. \ sound P \implies \vdash (v P) \vdash v P \) by (rule \( s\ell \))

show \(?X P \leq ?Y P\)
proof (rule le-transD \([ OF \ lfp-trans-lowerbound, \ OF \ - \ s\ell P])\)
  show le-trans \((T (\lambda a. \ lfp-exp (t a))) (\lambda a. \ lfp-exp (t a))\)
proof (rule le-transI)
  fix \( P::'s expect \)
  assume \( s\ell P \)
  from \( s\ell P \) have \( T (\lambda a. \ lfp-exp (t a)) P = t P (lfp-exp (t P)) \) by (rule tailcall)
also have \( t P (lfp-exp (t P)) = lfp-exp (t P) \)
  by (iprover intro: lfp-exp-unfold [symmetric] \( s\ell P \) \( st \) \( mt \) \( f_\ell P \) \( s\ell P \))
finally show \( T (\lambda a. \ lfp-exp (t a)) P \vdash lfp-exp (t P) \) by (simp)
qed
fix \( P::'s expect \)
assume \( \sound P \)
with \( f_\ell P \) \( s\ell P \) show \( \sound (lfp-exp (t P)) \)
  by (blast intro: lfp-exp-sound)
qed

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

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

lemma Inf-utrans-greatest:
\[ \bigwedge P. \unitary P \implies \unitary (t P); \forall u \in S. \le-utrans t u \bigwedge \le-utrans t \]
(Inf-utrans S)
unfolding Inf-utrans-def Inf-trans-def
by (cases \( S = \{ \), simp-all, (blast intro!: le-utransI Inf-exp-greatest)+)

end
Chapter 4

The pGCL Language

4.1 A Shallow Embedding of pGCL in HOL

theory Embedding imports Misc Induction begin

4.1.1 Core Primitives and Syntax

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

type-synonym 's prog = bool ⇒ ('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
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
where PC a P b ≡ λab Q s. P s * a ab Q s + (1 − P s) * b ab Q s

75
CHAPTER 4. THE PGCL LANGUAGE

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. 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. 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
(\[ - / - 100 \]

translations \[ \prod_{x\in S.} p \Rightarrow \text{CONST SetDC } (\%x. \ p) \ S \]
The above syntax allows us to write \[ \prod_{x\in S.} \]

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. \[ \sum_{a\in\text{supp } (p s).} p s a * f a ab P s \]

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
\[ \text{Bind } g f \ ab \equiv \lambda P \ s. \ let a = g s in f a \ ab \ P \ s \]

This gives us something like let syntax

\begin{verbatim}
  syntax -Bind :: pttrn => ('s => 'a) => 's prog => 's prog
            (- is - in - [55,55,55],55)
  translations x is f in a => CONST Bind f (%x. a)
\end{verbatim}

\textbf{definition} \( \text{flip} :: ('a => 'b => 'c) \Rightarrow 'b => 'a => 'c \)

\textbf{where} [\texttt{simp}]: \( \text{flip } f = (\lambda b \ a. f a b) \)

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

\begin{verbatim}
  syntax -PBind :: pttrn => ('s => real) = 's prog
            (- is - in - [55,55,55],55)
  translations bind x at p in a = CONST SetPC (%x. a) (CONST flip (%x. p))
\end{verbatim}

\begin{verbatim}
  syntax -DBind :: pttrn => ('s => 'a set) \Rightarrow 's prog
            (bind - from - in - [55,55,55],55)
  translations bind x from S in a \Rightarrow CONST SetDC (%x. a) S
\end{verbatim}

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

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

  parse-translation :(@{syntax-const -assign}, assign-tr)]
\end{verbatim}

\begin{verbatim}
  syntax -SetPC :: ident => ('s => 'a => real) \Rightarrow 's prog
            (choose - at - [66,66],66)
  ML \begin{verbatim}
  fun set-pc-tr - [Const (f,-), P] =
    Const (SetPC, dummyT) $ Abs (v, dummyT, (Const (Embedding.Apply, dummyT) $ Abs (s, dummyT, Syntax.const (suffix Record.updateN f) $ Abs (Name.uu-, dummyT, Bound 2) $ Bound 0))) $ P |
    set-pc-tr - ts = raise TERM (set-pc-tr, ts)
  \end{verbatim}
\end{verbatim}
These definitions instantiate the embedding as either weakest precondition (True) or weakest liberal precondition (False).

**syntax**

\[-set-dc :: \text{ident} \Rightarrow (’s \Rightarrow ’\text{a set}) \Rightarrow ’\text{s prog} (- :\in -[66,66]66)\]

**ML**

```ml
fun set-dc-tr - [Const (f,-), S] =
  Const (SetDC, dummyT) $
  Abs (v, dummyT,
       (Const (Embedding.Apply, dummyT) $
   Abs (s, dummyT,
       Syntax.const (suffix Record.updateN f) $
       Abs (Name.uu-, dummyT, Bound 2) $ Bound 0))) $ S
  | set-dc-tr - ts = raise TERM (set-dc-tr, ts)
```

**translations**

\[-set-dc-UNIV x \Rightarrow -set-dc x (\%.-. \text{CONST UNIV})\]

**definition**

\[wp :: ’\text{s prog} \Rightarrow ’\text{s trans}\]

**where**

\[wp pr \equiv pr \text{ True}\]

**definition**

\[wlp :: ’\text{s prog} \Rightarrow ’\text{s trans}\]

**where**

\[wlp pr \equiv pr \text{ False}\]

If-Then-Else as a degenerate probabilistic choice.

**abbreviation**(input)

\[if-then-else :: [’s \Rightarrow \text{bool}, ’\text{s prog}, ’\text{s prog}] \Rightarrow ’\text{s prog}\]

**where**

\[\text{If } P \text{ Then } a \text{ Else } b \equiv a \cdot P @ \oplus b\]

Syntax for loops

**abbreviation**

\[do-while :: [’s \Rightarrow \text{bool}, ’\text{s prog}] \Rightarrow ’\text{s prog}\]

**where**

\[do-while P a \equiv \mu \text{ x. If } P \text{ Then } a \cdot x \text{ Else Skip}\]
4.1. A SHALLOW EMBEDDING OF PGCL IN HOL

4.1.2 Unfolding rules for non-recursive primitives

lemma eval-wp-Abort:
  \text{wp} \text{Abort} P = (\lambda s. 0)
  \text{unfolding wp-def Abort-def by(simp)}

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

lemma eval-wp-Skip:
  \text{wp} \text{Skip} P = P
  \text{unfolding wp-def Skip-def by(simp)}

lemma eval-wlp-Skip:
  \text{wlp} \text{Skip} P = P
  \text{unfolding wlp-def Skip-def by(simp)}

lemma eval-wp-Apply:
  \text{wp} (\text{Apply} f) P = P o f
  \text{unfolding wp-def Apply-def by(simp add:o-def)}

lemma eval-wlp-Apply:
  \text{wlp} (\text{Apply} f) P = P o f
  \text{unfolding wlp-def Apply-def by(simp add:o-def)}

lemma eval-wp-Seq:
  \text{wp} (a ;; b) P = (\text{wp} a o \text{wp} b) P
  \text{unfolding wp-def Seq-def by(simp)}

lemma eval-wlp-Seq:
  \text{wlp} (a ;; b) P = (\text{wlp} a o \text{wlp} b) P
  \text{unfolding 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)
  \text{unfolding 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)
  \text{unfolding wlp-def PC-def by(simp)}

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

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

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

lemma eval-wp-Embed:
\[ wp (Embed t) = t \]
unfolding wp-def Embed-def by(simp)

lemma eval-wlp-Embed:
\[ wlp (Embed t) = t \]
unfolding wlp-def Embed-def by(simp)

lemma eval-wp-SetDC:
\[ wp (SetDC p S) R s = \Inf ((\lambda a. wp (p a) R s) \mathbin{'} S s) \]
unfolding wp-def SetDC-def by(simp)

lemma eval-wlp-SetDC:
\[ wlp (SetDC p S) R s = \Inf ((\lambda a. wlp (p a) R s) \mathbin{'} S s) \]
unfolding wlp-def SetDC-def by(simp)

lemma eval-wp-SetPC:
\[ wp (SetPC f p) P = (\lambda s. \sum a \in supp (p s). p s a \ast wp (f a) P s) \]
unfolding wp-def SetPC-def by(simp)

lemma eval-wlp-SetPC:
\[ wlp (SetPC f p) P = (\lambda s. \sum a \in supp (p s). p s a \ast wlp (f a) P s) \]
unfolding wlp-def SetPC-def by(simp)

lemma eval-wp-Mu:
\[ wp (\mu t. T t) = \lfp-trans (\lambda t. wp (T (Embed t))) \]
unfolding wp-def Mu-def by(simp)

lemma eval-wlp-Mu:
\[ wlp (\mu t. T t) = \gfp-trans (\lambda t. wlp (T (Embed t))) \]
unfolding wlp-def Mu-def by(simp)

lemma eval-wp-Bind:
\[ wp (Bind g f) = (\lambda P s. wp (f (g s)) P s) \]
unfolding Bind-def wp-def Let-def by(simp)

lemma eval-wlp-Bind:
\[ wlp (Bind g f) = (\lambda P s. wlp (f (g s)) P s) \]
unfolding Bind-def wlp-def Let-def by(simp)

Use simp add:wp_eval to fully unfold a program fragment
4.2. HEALTHINESS

lemmas \( wp\-eval = eval\-wp\-Abort \) \( eval\-wp\-Skip \) \( eval\-wp\-Apply \) \( eval\-wp\-Seq \) \( eval\-wp\-PC \) \( eval\-wp\-DC \) \( eval\-wp\-AC \) \( eval\-wp\-Mu \) \( eval\-wp\-Bind \) 

lemma Skip-Seq:
\[
\text{Skip} \; ; \; A = A
\]
unfolding Skip-def Seq-def o-def by (rule refl)

lemma Seq-Skip:
\[
A \; ; \; \text{Skip} = A
\]
unfolding Skip-def Seq-def o-def by (rule refl)

Use these as simp rules to clear out Skips

lemmas skip-simps = Skip-Seq Seq-Skip

end

4.2 Healthiness

theory Healthiness imports Embedding begin

4.2.1 The Healthiness of the Embedding

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

lemma healthy-wp-Abort:
\[
\text{healthy (wp Abort)}
\]
proof (rule healthy-parts)
fix \( b \) and \( P::a \Rightarrow \text{real} \)
assume \( nP: \text{neg P and bP: bounded-by b P} \)
thus \( \text{bounded-by b (wp Abort P)} \)
unfolding wp-eval by (blast)
show \( \text{nneq (wp Abort P)} \)
unfolding wp-eval by (blast)
next
fix \( P Q::a \) expect
show \( \text{wp Abort P \vdash wp Abort Q} \)
unfolding wp-eval by (blast)
next
fix \( P \) and \( c \) and \( s::a \)
show \( \text{c * wp Abort P s = wp Abort (\lambda s. c * P s) s} \)
unfolding wp-eval by (auto)
qed
lemma \textit{nearly-healthy-wlp-Abort}:
\textit{nearly-healthy \ (wlp \ Abort)}
\begin{proof}(\textit{rule nearly-healthyI})
fix \( P \)\':\( 's \Rightarrow \) \text{real}
show \text{unitary \ (wlp \ Abort \ P)}
by(\textit{simp add:wp-eval})
next
fix \( P, Q \)\':\( 's \) \text{expect}
assume \( P \vdash Q \) \text{ and unitary \( P \) and unitary \( Q \)}
thus \( \text{wlp \ Abort \ P} \vdash \text{wlp \ Abort \ Q} \)
unfolding \text{wp-eval} by(\textit{blast})
qed
\end{proof}

lemma \textit{healthy-wp-Skip}:
\textit{healthy \ (wp \ Skip)}
by(\textit{force intro!:\text{healthy-parts simp:wp-eval}})

lemma \textit{nearly-healthy-wlp-Skip}:
\textit{nearly-healthy \ (wlp \ Skip)}
by(\textit{auto simp:wp-eval})

lemma \textit{healthy-wp-Seq}:
fixes \( t \)\':\( 's \) prog and \( u \)
assumes \( \text{ht} \)\: \text{healthy \ (wp \ t)} \text{ and hu} \: \text{healthy \ (wp \ u)}
shows \text{healthy \ (wp \ (t \;;\; u))}
\begin{proof}(\textit{rule healthy-parts}, \text{simp-all add:wp-eval})
fix \( b \) and \( P \)\':\( 's \Rightarrow \) \text{real}
assume \text{bounded-by \( b \) \( P \) and nneg \( P \)}
with \( \text{hu have} \) \text{bounded-by \( b \) \( (wp \ u \ P) \) and nneg \( (wp \ u \ P) \) by(auto)}
with \( \text{ht show} \) \text{bounded-by \( b \) \( (wp \ t \ (wp \ u \ P)) \)}
and \text{nneg \( (wp \ t \ (wp \ u \ P)) \) by(auto)}
next
fix \( P \)\':\( 's \Rightarrow \) \text{real and} \( Q \)
assume \text{sound \( P \) and sound \( Q \) and \( P \vdash Q \)}
with \( \text{hu have} \) \text{sound \( (wp \ u \ P) \) and sound \( (wp \ u \ Q) \)}
and \text{wp \( u \ P \vdash wp \ u \ Q \) by(auto)}
with \( \text{ht show} \) \text{wp \( t \ (wp \ u \ P) \vdash wp \ t \ (wp \ u \ Q) \) by(auto)}
next
fix \( P \)\':\( 's \Rightarrow \) \text{real and} \( c::\text{real and} \ s \)
assume \text{pos} \: \text{0 \leq} \text{ c and} \text{ sP: sound P}
with \( \text{ht and hu have} \) \text{c \* \( wp \ t \ (wp \ u \ P) \) \( s \ = \ wp \ t \ (\lambda s. \ c \* \( wp \ u \ P \) s) \) s}
by(\textit{auto intro\!:\text{scalingD}})
also with \( \text{hu and pos and sP have} \) \text{... \( = \ wp \ t \ (wp \ u \ (\lambda s. \ c \* \( P \) s)) \) s}
by(\textit{simp add:scalingD(OF healthy-scalingD)})
finally show \text{c \* \( wp \ t \ (wp \ u \ P) \) \( s \ = \ wp \ t \ (wp \ u \ (\lambda s. \ c \* \( P \) s)) \) s}.
qed

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

proof (rule nearly-healthyI, simp-all add: wp-eval)
  fix b and P::'s ⇒ 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 ⇒ real
  assume unitary P and unitary Q and P ⊢ Q
  with hu have unitary (wlp u P) and unitary (wlp u Q)
  and wlp u P ⊢ wlp u Q by (auto)
  with ht show wlp t (wlp u P) ⊢ wlp t (wlp u Q) by (auto)
qed

lemma healthy-wp-PC:
  fixes f::'s prog
  assumes hf: healthy (wp f) and hg: healthy (wp g)
  and uP: unitary P
  shows healthy (wp (f ⊕ g))
proof (intro healthy-parts bounded-byI nnegI le-funI, simp-all add: wp-eval)
  fix b and Q::'s ⇒ real and s::'s
  assume nQ: nneg Q and bQ: bounded-by b Q
Non-negative:
  from nQ and bQ and hf have 0 ≤ wp f Q s by (auto)
  with uP have 0 ≤ P s * ... by (auto intro: mult-nonneg-nonneg)
  moreover {
    from uP have 0 ≤ 1 − P s
      by (auto intro: mult-nonneg-nonneg)
    with nQ and bQ and hg have 0 ≤ ... * wp g Q s
      by (metis healthy-nnegD2 mult-nonneg-nonneg nneg-def)
  }
  ultimately show 0 ≤ P s * wp f Q s + (1 − P s) * wp g Q s
    by (auto intro: mult-nonneg-nonneg)
Bound:
  from nQ bQ hf have wp f Q s ≤ b by (auto)
  with uP nQ bQ hf have P s * wp f Q s ≤ P s * b
    by (blast intro!: mult-mono)
  moreover {
    from nQ bQ hg uP have wp g Q s ≤ b and 0 ≤ 1 − P s
      by (auto)
    with nQ bQ hg have (1 − P s) * wp g Q s ≤ (1 − P s) * b
      by (blast intro!: mult-mono)
  }
  ultimately have P s * wp f Q s + (1 − P s) * wp g Q s ≤ P s * b + (1 − P s) * b
by\,(\texttt{blast intro:add-mono})
also have \ldots = b \ by\, (\texttt{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:
fix \( Q \ R::'s \Rightarrow \text{real} \) and \( s\) assume \( sQ: \text{sound} \ Q \) and \( sR: \text{sound} \ R \) and \( \text{le}: Q \vdash \vdash R \) with \( \texttt{hf} \) have \( \text{wp} \ f \ Q \ s \leq \text{wp} \ f \ R \ s \) by\,(\texttt{blast dest:mono-transD})
with \( \texttt{uP} \) have \( P \ s \ast \ wp \ f \ Q \ s \leq P \ s \ast \ wp \ f \ R \ s \) by\,(\texttt{auto intro:mult-left-mono})
moreover \{ from \( sQ \ sR \ \text{le} \) \( \texttt{hg} \) have \( \text{wp} \ g \ Q \ s \leq \text{wp} \ g \ R \ s \) by\,(\texttt{blast dest:mono-transD})
moreover from \( \texttt{uP} \) have \( 0 \leq (1 - P \ s) \ast \ wp \ g \ Q \ s \) by\,(\texttt{auto}) ultimately have \((1 - P \ s) \ast \ wp \ g \ Q \ s \leq (1 - P \ s) \ast \ wp \ g \ R \ s \) by\,(\texttt{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\,(\texttt{auto})
next

Scaling:
fix \( Q::'s \Rightarrow \text{real} \) and \( c::\text{real} \) and \( s::'s \) assume \( \texttt{uQ}: \text{unitary} \ Q \) shows \( \text{nearby-healthy} \ (\text{wlp} \ (f \oplus g)) \)
proof\,(\texttt{intro nearly-healthyI unitaryI2 nnegI bounded-byI le-funI,}
\texttt{simp-all add:wp-eval})
fix \( Q::'s \) expect and \( s::'s \)
assume \( \texttt{uQ}: \text{unitary} \ Q \)
from \( \texttt{uQ \ hf \ hg} \) have \( \texttt{utQ}: \text{unitary} \ (\text{wlp} \ f \ Q) \) \text{unitary} \ (\text{wlp} \ g \ Q) \ by\,(\texttt{auto})
from \( uP \) have \( \mathbb{nnP} : 0 \leq P \ s \ 0 \leq 1 \ - \ P \ s \) 
by \( \text{auto} \) 
moreover from \( utQ \) have \( 0 \leq \wp f \ Q \ s \ 0 \leq \wp g \ Q \ s \) by\( \text{auto} \) 
ultimately show \( 0 \leq P \ * \ \wp f \ Q \ s \ + \ \(1\ - \ P \ s\) \ - \ \wp g \ Q \ s \) 
by\( \text{auto intro;} \text{add-nonneg-nonneg mult-nonneg-nonneg} \) 
from \( utQ \) have \( \wp f \ Q \ s \leq 1 \ \wp g \ Q \ s \leq 1 \) by\( \text{auto} \) 
with \( \mathbb{nnP} \) have \( P \ * \ \wp f \ Q \ s \leq \wp g \ Q \ s \leq P \ * \ 1 \ - \ (1\ - \ P \ s) \) 
by\( \text{(auto intro:} \text{add-mono mult-left-mono}) \) 
ultimately show \( P \ * \ \wp f \ Q \ s \ + \ \(1\ - \ P \ s\) \ - \ \wp g \ Q \ s \leq 1 \) by\( \text{simp} \) 

fix \( R \)’s expect 
assume \( uR : \text{unitary R and le: } Q \vdash R \) 
with \( uQ \) have \( \wp f \ Q \ s \leq \wp f \ R \ s \) 
by\( \text{(auto intro:le-funD[OF nearly-healthy-monoD, OF hf])} \) 
with \( \mathbb{nnP} \) have \( P \ * \ \wp f \ Q \ s \leq \wp g \ Q \ s \leq P \ * \ \wp f \ R \ s \) 
by\( \text{(auto intro:mult-left-mono)} \) 
moreover \{ 
from \( uQ \ uR \ le \) have \( \wp g \ Q \ s \leq \wp g \ R \ s \) 
by\( \text{(auto intro:le-funD[OF nearly-healthy-monoD, OF hg])} \) 
with \( \mathbb{nnP} \) have \( (1\ - \ P \ s) \ - \ wp g \ Q \ s \leq (1\ - \ P \ s) \ - \ wp g \ R \ s \) 
by\( \text{(auto intro:mult-left-mono)} \) 
\} 
ultimately show \( P \ * \ \wp f \ Q \ s \ + \ \(1\ - \ P \ s\) \ - \ \wp g \ Q \ s \leq P \ * \ \wp f \ R \ s \ + \ \(1\ - \ P \ s\) \ - \ \wp g \ R \ s \) 
by\( \text{(auto)} \) 

qed 

lemma \( \text{healthy-wp-DC:} \) 
fixes \( f \)’s \( \text{real} \) 
assumes \( hP : \text{sound P and le: } P \vdash Q \) 
shows \( \text{healthy (wp f)} \) 
proof\( \text{intro healthy-parts bounded-byI nnegI le-funI, simp-all only:wp-eval) \) 
fix \( b \) and \( P \)’s \( \Rightarrow \) \( \text{real} \) and \( s \)’s 
assume \( nP : \text{nneg P and bP: bounded-by } b \ P \) 
with \( hP \) have bounded-by-b \( (wp f P) \) by\( \text{(auto)} \) 
hence \( wp f P \ s \leq b \) by\( \text{(blast)} \) 
thus \( \text{min} \ (wp f P \ s) \ (wp g P \ s) \leq b \) by\( \text{(auto)} \) 

from \( nP \ bP \) \( \text{assms show} \ 0 \leq \text{min} \ (wp f P \ s) \ (wp g P \ s) \) by\( \text{(auto)} \) 

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

next

fix \( P: s \Rightarrow real \) and \( c: real \) and \( s: s \)

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

from \( \text{assms} \) have \( sf: \text{scaling } (wp f) \) and \( sg: \text{scaling } (wp g) \) by(auto)

from \( \text{pos} \) 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 \( \text{pos} \) have ...

= \( \min (wp f (\lambda s. c * P s) s) \)

by(simp add:scalingD[OF sf] scalingD[OF sg])

finally show \( c * \min (wp f P s) (wp g P s) = \min (wp f (\lambda s. c * P s) s) (wp g (\lambda s. c * P s) s) \).

dqed

lemma nearly-healthy-wlp-DC:

fixes \( f: s \Rightarrow \) prog

assumes \( hf: \text{nearly-healthy } (wp f) \)

and \( hg: \text{nearly-healthy } (wp g) \)

shows \( \text{nearly-healthy } (wp (f \sqcap g)) \)

proof(intro nearly-healthyI bounded-byI nnegI le-funI unitaryI2, simp-all add:wp-eval, safe)

fix \( P: s \Rightarrow real \) and \( s: s \)

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

with \( hf \) \( hg \) have \( atP: \text{unitary } (wp f P) \) \( \text{unitary } (wp g P) \) by(auto)

thus \( 0 \leq \min (wp f P \ s \ 0 \leq \min (wp g P \ s) \)

by(auto)

have \( \min (wp f P s) (wp g P s) \leq wp f P s \) by(auto)

also from \( atP \) have ...

by(auto)

finally show \( \min (wp f P s) (wp g P s) \leq 1 \).

fix \( Q: s \Rightarrow real \)

assume \( uQ: \text{unitary } Q \) and \( le: P \vdash Q \)

have \( \min (wp f P s) (wp g P s) \leq wp f P s \) by(auto)

also from \( uP \) \( uQ \) \( le \) have ...

by(auto intro:le-funD[OF nearly-healthy-monoD, OF hf])

finally show \( \min (wp f P s) (wp g P s) \leq wp f Q s \).

have \( \min (wp f P s) (wp g P s) \leq wp g P s \) by(auto)

also from \( uP \) \( uQ \) \( le \) have ...

by(auto intro:le-funD[OF nearly-healthy-monoD, OF hg])

finally show \( \min (wp f P s) (wp g P s) \leq wp g Q s \).

dqed

lemma healthy-wp-AC:

fixes \( f: s \Rightarrow \) prog

assumes \( hf: \text{healthy } (wp f) \) and \( hg: \text{healthy } (wp g) \)

shows \( \text{healthy } (wp (f \sqcap g)) \)

proof(intro healthy-parts bounded-byI nnegI le-funI, simp-all only:wp-eval)
4.2. HEALTHINESS

\begin{verbatim}
fix \( b \) and \( P::'s \Rightarrow \text{real and } s::'s \)
assume \( nP: \text{nneg \( P \)} \text{ and } bP: \text{bounded-by } b \ P \)

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

\from \( nP \) \( bP \) \textbf{assms have} \( 0 \leq \text{wp } f \ P \ s \) by(auto)
\textbf{thus} \( 0 \leq \text{max} \ (\text{wp } f \ P \ s) \ (\text{wp } g \ P \ s) \) by(auto)
\textbf{next}
\fix \( P::'s \Rightarrow \text{real and } Q \text{ and } s::'s \)
\from \textbf{assms have} \( \text{mf}: \text{mono-trans } (\text{wp } f) \) \textbf{and} \( \text{mg}: \text{mono-trans } (\text{wp } g) \) by(auto)
\assume \( sP: \text{sound } P \) \textbf{and} \( sQ: \text{sound } Q \) \textbf{and} \( \text{le}: P \vdash Q \)
\textbf{hence} \( \text{wp } f \ P \ s \leq \text{wp } f \ Q \ s \textbf{ and} \text{wp } g \ P \ s \leq \text{wp } g \ Q \ s \)
by(auto intro:le-funD[OF mono-transD, OF mf] le-funD[OF mono-transD, OF mg])
\textbf{thus} \( \text{max} \ (\text{wp } f \ P \ s) \ (\text{wp } g \ P \ s) \leq \text{max} \ (\text{wp } f \ Q \ s) \ (\text{wp } g \ Q \ s) \) by(auto)
\textbf{next}
\fix \( P::'s \Rightarrow \text{real and } c::\text{real and } s::'s \)
\assume \( sP: \text{sound } P \) \textbf{and} \( \text{pos}: 0 \leq c \)
\from \textbf{assms have} \( \text{sf}: \text{scaling } (\text{wp } f) \) \textbf{and} \( \text{sg}: \text{scaling } (\text{wp } g) \) by(auto)
\from \textbf{pos have} \( c \ast \text{max} \ (\text{wp } f \ P \ s) \ (\text{wp } g \ P \ s) = \)
\textbf{max} \( (c \ast \text{wp } f \ P \ s) \ (c \ast \text{wp } g \ P \ s) \)
by(simp add:max-distrib)
\also \from \textbf{sP and pos have} \( \ldots = \text{max} \ (\text{wp } f (\lambda s. c \ast P s) s) \ (\text{wp } g (\lambda s. c \ast P s) s) \)
by(simp add:scalingD[OF sf] scalingD[OF sg])
\textbf{finally show} \( c \ast \text{max} \ (\text{wp } f \ P \ s) \ (\text{wp } g \ P \ s) = \)
\text{max} \( (\text{wp } f (\lambda s. c \ast P s) s) \ (\text{wp } g (\lambda s. c \ast P s) s) \).
\end{verbatim}

\textbf{qed}

\textbf{lemma} nearly-healthy-wlp-AC:
\textbf{fixes} \( f::'s \text{ prog} \)
\textbf{assumes} \( hf: \text{nearly-healthy } (\text{wlp } f) \)
\textbf{and} \( hg: \text{nearly-healthy } (\text{wlp } g) \)
\textbf{shows} \( \text{nearly-healthy } (\text{wlp } (f \bigcup g)) \)
\textbf{proof}(intro nearly-healthyI bounded-byI \text{nnegI unitaryI2 le-funI, simp-all only:wp-eval)
\fix \( b \) and \( P::'s \Rightarrow \text{real and } s::'s \)
\assume \( uP: \text{unitary } P \)

\with \( hf \) have \( \text{wlp } f \ P \ s \leq 1 \) by(auto)
\textbf{moreover} \from \( uP \) \( hg \) \textbf{have} \( \text{unitary } (\text{wlp } g \ P) \) by(auto)
\textbf{hence} \( \text{wlp } g \ P \ s \leq 1 \) by(auto)
\textbf{ultimately show} \( \text{max} \ (\text{wlp } f \ P \ s) \ (\text{wlp } g \ P \ s) \leq 1 \) by(auto)
from uP hf have unitary (wlp f P) by(auto)
  hence 0 ≤ wlp f P s by(auto)
  thus 0 ≤ max (wlp f P s) (wlp g P s) by(auto)

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

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

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

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

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

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

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

ultimately have $\Inf ((\lambda a. \ wp \ (\text{prog} \ a) \ P \ s) \ > S \ s) \leq x$

by(intro $c\inf$-lower $\bdd$-belowI, auto)

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

finally show $\Inf ((\lambda a. \ wp \ (\text{prog} \ a) \ P \ s) \ > S \ s) \leq b$ .

from $xin$ and $sP$ and $healthy$
show $0 \leq \Inf ((\lambda a. \ wp \ (\text{prog} \ a) \ P \ s) \ > S \ s)$ by(blast intro:$c\inf$-greatest)

next
fix $P::a \Rightarrow \text{real and } Q$ and $s: a$
assume $sP$: sound $P$ and $sQ$: sound $Q$ and $le: P \vdash Q$

from nonempty obtain $x$ where $xin: x \in (\lambda a. \ wp \ (\text{prog} \ a) \ P \ s) \ > S \ s$ by(blast) moreover from $sP$ and $healthy$

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

moreover

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

proof(rule ballI, clarify, rule $\text{bexI})$

fix $x$ and $a$ assume $ain$: $a \in S \ s$

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-$\text{trans}D[\text{OF} healthy$-$\text{mono}D])$

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

qed ultimately

show $\Inf ((\lambda a. \ wp \ (\text{prog} \ a) \ P \ s) \ > S \ s) \leq \Inf ((\lambda a. \ wp \ (\text{prog} \ a) \ Q \ s) \ > S \ s)$

by(intro $c\inf$-mono, $\text{blast}+$)

next
fix $P::a \Rightarrow \text{real and } c::\text{real and } s::a$
assume $sP$: sound $P$ and $pos: \theta \leq c$

from nonempty obtain $x$ where $xin: x \in (\lambda a. \ wp \ (\text{prog} \ a) \ P \ s) \ > S \ s$ by(blast) have $c * \Inf ((\lambda a. \ wp \ (\text{prog} \ a) \ P \ s) \ > S \ s) =$

$\Inf ((\text{*} \ c \ > ((\lambda a. \ wp \ (\text{prog} \ a) \ P \ s) \ > S \ s)) \ (\text{is } ?U = ?V)$

proof(rule antisym)

show $?U \leq ?V$

proof(rule $c\inf$-greatest)

from nonempty show $\text{(*)} \ c \ > ((\lambda a. \ wp \ (\text{prog} \ a) \ P \ s) \ > S \ s \neq \{\})$ by(auto)

fix $x$ assume $x \in (\text{*)} \ c \ > ((\lambda a. \ wp \ (\text{prog} \ a) \ P \ s) \ > S \ s$

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

* $y$ by(auto)

have $\Inf ((\lambda a. \ wp \ (\text{prog} \ a) \ P \ s) \ > S \ s) \leq y$

proof(intro $c\inf$-lower[\text{OF} $yin$] $\text{bdd}$-belowI)

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

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

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

qed with $pos \ rwux$ show $c * \Inf ((\lambda a. \ wp \ (\text{prog} \ a) \ P \ s) \ > S \ s) \leq x$ by(auto
\textbf{CHAPTER 4. THE PGCL LANGUAGE}

\begin{verbatim}
intro:mult-left-mono
  qed
  show \(?V \leq ?U\)
  proof (cases)
    assume \( \v c: c = 0 \)
    moreover \{
      from nonempty obtain \( c \) where \( c \in S \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \) \( \)...


4.2. HEALTHINESS

assumes healthy: \( \forall x. x \in S \Rightarrow \text{nearby-healthy} (\text{wp} (\text{prog} x)) \)
and nonempty: \( \exists x. x \in S \)
shows nearby-healthy (\text{wp} (\text{SetDC} \text{ prog} S)) (is nearby-healthy ?T)

proof (\text{intro nearby-healthy I unitary I2 bounded-by I neg I le-fun I, simp-all only: wp-eval})

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

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

moreover \{ 
  from \( uP \) healthy
  have \( \forall x \in (\lambda a. \text{wp} (\text{prog} a) P) \cdot S s. \text{unitary} x \) by (auto)
  hence \( \forall x \in (\lambda a. \text{wp} (\text{prog} a) P) \cdot S s. 0 \leq x s \) by (auto)
  hence \( \forall y \in (\lambda a. \text{wp} (\text{prog} a) P s) \cdot S s. 0 \leq y \) by (auto)
\}

ultimately have \( \text{Inf} ((\lambda a. \text{wp} (\text{prog} a) P s) \cdot S s) \leq x \) by (intro cInf-lower)

bdd_below I, auto

also from \( x \in \text{healthy} uP \) have \( x \leq 1 \) by (blast)

finally show \( \text{Inf} ((\lambda a. \text{wp} (\text{prog} a) P s) \cdot S s) \leq 1 \).

from \( x \in uP \) healthy

show \( 0 \leq \text{Inf} ((\lambda a. \text{wp} (\text{prog} a) P s) \cdot S s) \)
  by (blast dest!: unitary-sound [OF nearby-healthy-unitary I [OF - uP]]
    intro cInf-greatest)

next

fix \( P :: 'a \Rightarrow \text{real} \) and \( Q \) and \( s :: 'a \)
assume \( uP :: \text{unitary} P \) and \( uQ :: \text{unitary} Q \) and \( \text{le} :: P \Rightarrow Q \)

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

moreover \{ 
  from \( uP \) healthy
  have \( \forall x \in (\lambda a. \text{wp} (\text{prog} a) P) \cdot S s. \text{unitary} x \) by (auto)
  hence \( \forall x \in (\lambda a. \text{wp} (\text{prog} a) P) \cdot S s. 0 \leq x s \) by (auto)
  hence \( \forall y \in (\lambda a. \text{wp} (\text{prog} a) P s) \cdot S s. 0 \leq y \) by (auto)
\}

moreover

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

proof (rule ballI, clarify, rule bexI)

fix \( x \) and \( a \) assume \( \alpha :: a \in S s \)
from \( uP \) uQ le show \( \text{wp} (\text{prog} a) P s \leq \text{wp} (\text{prog} a) Q s \)
  by (auto intro: le-fun D [OF nearby-healthy-mono D [OF healthy, OF \alpha]])
from \( \alpha \) show \( \text{wp} (\text{prog} a) P s \in (\lambda a. \text{wp} (\text{prog} a) P s) \cdot S s \) by (simp)

qed

ultimately

show \( \text{Inf} ((\lambda a. \text{wp} (\text{prog} a) P s) \cdot S s) \leq \text{Inf} ((\lambda a. \text{wp} (\text{prog} a) Q s) \cdot S s) \)
  by (intro cInf-mono, blast +)

qed

lemma healthy-wp-SetPC:

fixes \( p :: 'a \Rightarrow 'a \Rightarrow \text{real} \)

and $f::a \Rightarrow s\ prog$
assumes healthy: $\forall a\ s.\ a \in\ supp\ (p\ s) \Rightarrow healthy\ (wp\ (f\ a))$
and sound: $\forall s.\ sound\ (p\ s)$
and sub-dist: $\forall s.\ (\sum a\in supp\ (p\ s).\ p\ s\ a) \leq 1$
shows healthy (wp (SetPC $f\ p$)) (is healthy $\ ?X$)
proof (intro healthy-parts bounded-byI nnegI le-funI, simp-all add:wp-eval)
fix $b$ and $P::s\Rightarrow real\ and\ s::\ s'$
assume $bP$: bounded-by $b\ P\ and\ nP$: nneg $P$
hence $sP$: sound $P$ by(auto)

from $sP$ and $bP$ and healthy have $\forall a\ s.\ a \in\ supp\ (p\ s) \Rightarrow wp\ (f\ a)\ P\ s \leq b$
by(blast dest:healthy-bounded-byD)
with sound have $(\sum a\in supp\ (p\ s).\ p\ s\ a \ast wp\ (f\ a)\ P\ s) \leq (\sum a\in supp\ (p\ s).\ p\ s\ a \ast b)$
by(blast intro:sum-mono mult-left-mono)
also have ... = $(\sum a\in supp\ (p\ s).\ p\ s\ a \ast b)$
by(simp add:sum-distrib-right)
also {}
from $bP$ and $nP$ have $0 \leq b$ by(blast)
with sub-dist have $(\sum a\in supp\ (p\ s).\ p\ s\ a \ast b) \leq 1 \ast b$
by(rule mult-right-mono)
}
also have $1 \ast b = b$ by(simp)
finally show $(\sum a\in supp\ (p\ s).\ p\ s\ a \ast wp\ (f\ a)\ P\ s) \leq b$.

show $0 \leq (\sum a\in supp\ (p\ s).\ p\ s\ a \ast wp\ (f\ a)\ P\ s)$
proof (rule sum-nonneg [OF mult-nonneg-nonneg])
fix $x$
from sound show $0 \leq p\ s\ x$ by(blast)
assume $x \in supp\ (p\ s)$ with $sP$ and healthy
show $0 \leq wp\ (f\ x)\ P\ s$ by(blast)
qed

next
fix $P::s\Rightarrow real\ and\ Q::s\Rightarrow real\ and\ s$
assume $s\ P$: sound $P$ and $s\ Q$: sound $Q$ and $ent: P \vdash Q$
with healthy have $\forall a.\ a \in\ supp\ (p\ s) \Rightarrow wp\ (f\ a)\ P\ s \leq wp\ (f\ a)\ Q\ s$
by(blast)
with sound have $(\sum a\in supp\ (p\ s).\ p\ s\ a \ast wp\ (f\ a)\ P\ s) \leq (\sum a\in supp\ (p\ s).\ p\ s\ a \ast wp\ (f\ a)\ Q\ s)$
by(blast intro:sum-mono mult-left-mono)

next
fix $P::s\Rightarrow real\ and\ c::real\ and\ s::\ s'$
assume sound: sound $P$ and $pos: 0 \leq c$
have $c \ast (\sum a\in supp\ (p\ s).\ p\ s\ a \ast wp\ (f\ a)\ P\ s) = (\sum a\in supp\ (p\ s).\ p\ s\ a \ast (c \ast wp\ (f\ a)\ P\ s))$
(is $?A = ?B$)
by(simp add:sum-distrib-left ac-simps)
also from sound and $pos$ and healthy
have ... = $(\sum a\in supp\ (p\ s).\ p\ s\ a \ast wp\ (f\ a)\ (\lambda s.\ c \ast P\ s))$
4.2. HEALTHINESS

by(auto simp:scalingD[OF healthy-scalingD])
finally show ?A = ... .
qed

lemma nearly-healthy-wlp-SetPC:
  fixes p::'s ⇒ 'a ⇒ real
  and f::'a ⇒ 's prog
  assumes healthy: \( \forall a. s. a \in \text{supp} (p s) \implies \text{nearly-healthy} (\text{wlp} (f a)) \)
  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 nearly-healthy (\text{wlp} (SetPC f p)) (is nearly-healthy ?X)
proof(intro nearly-healthyI unitaryI2 bounded-byI nnegI le-funI, simp-all only:wp-eval)
  fix b and P::'s ⇒ real
  assume uP: unitary P
from uP healthy have \( \forall a. a \in \text{supp} (p s) \implies \text{unitary} (\text{wlp} (f a) \text{ P}) \)
  by(auto)
  hence \( \forall a. a \in \text{supp} (p s) \implies \text{wp} (f a) \text{ P s} \leq 1 \) by(auto)
with sound have \( (\sum a\in\text{supp} (p s). p s a \ast \text{wlp} (f a) \text{ P s}) \leq (\sum a\in\text{supp} (p s). p s a \ast 1) \)
  by(blast intro:sum-mono mult-left-mono)
also have ... = (\sum a\in\text{supp} (p s). p s a)
  by(simp add:sum-distrib-right)
also note sub-dist
finally show \( (\sum a\in\text{supp} (p s). p s a \ast \text{wlp} (f a) \text{ P s}) \leq 1 \)
  by(blast intro:sum-mono mult-left-mono)
proof(rule sum-nonneg [OF mult-nonneg-nonneg])
  fix x
  from sound show \( 0 \leq p s x \) by(blast)
  assume x\in\text{supp} (p s) with uP healthy
  show \( 0 \leq \text{wlp} (f x) \text{ P s} \) 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 \( \forall a. a \in \text{supp} (p s) \implies \text{wp} (f a) \text{ P s} \leq \text{wp} (f a) \text{ Q s} \)
  by(black intro:le-funD[OF nearly-healthy-monoD, OF healthy])
with sound show \( (\sum a\in\text{supp} (p s). p s a \ast \text{wlp} (f a) \text{ P s}) \leq (\sum a\in\text{supp} (p s). p s a \ast \text{wlp} (f a) \text{ Q s}) \)
  by(black intro:sum-mono mult-left-mono)
qed

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

lemma nearly-healthy-wlp-Apply:
  nearly-healthy (wp (Apply f))
by(intro nearly-healthyI unitaryI2 nnegI bounded-byI, auto simp:o-def wp-eval)
lemma healthy-wp-Bind:
fixes f::'s \Rightarrow 'a
assumes hsib: \forall s. healthy (wp (p (f s)))
shows healthy (wp (Bind f p))
proof(intro healthy-parts nnegI bounded-byI le-funI, simp-all only:wp-eval)
  fix b and P::'s expect and s::'
  assume bP: bounded-by b P and nP: nneg P
  with hsib have bounded-by b (wp (p (f s))) P by(auto)
  thus wp (p (f s)) P s \leq b by(auto)
  from bP nP hsib have nneg (wp (p (f s))) P by(auto)
  thus 0 \leq wp (p (f s)) P s by(auto)
next
fix P Q::'s expect and s::'
  assume sound P sound Q P \vdash Q
  thus 0 \leq wp (p (f s)) P s \leq 1 by(auto)
  from uP show wp (p (f s)) P s \leq wp (p (f s)) Q s
    by(blast intro:le-funD[OF nearly-healthy-monoD, OF hsib])
qed

lemma nearly-healthy-wlp-Bind:
fixes f::'s \Rightarrow 'a
assumes hsib: \forall s. nearly-healthy (wlp (p (f s)))
shows nearly-healthy (wlp (Bind f p))
proof(intro nearly-healthyI unitaryI2 nnegI bounded-byI le-funI, simp-all only:wp-eval)
  fix P::'s expect and s::'
  assume uP: unitary P
  with hsib have unitary (wlp (p (f s))) P by(auto)
  thus 0 \leq wlp (p (f s)) P s \leq 1 by(auto)
fix Q::'s expect
  assume unitary Q P \vdash Q
  with uP show wlp (p (f s)) P s \leq wlp (p (f s)) Q s
    by(blast intro:le-funD[OF nearly-healthy-monoD, OF hsib])
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: \forall P. sound P \Rightarrow sound (t P)
and hu: \forall P. sound P \Rightarrow sound (u P)
shows le-trans (wp (body :: Embed t a G s \oplus Skip))
    (wp (body :: Embed u a G s \oplus Skip))
proof(intro le-transI le-funI, simp add:wp-eval)
4.2. HEALTHINESS

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:**

**Assumptions**
- \( \text{hb}: \text{healthy (wp body)} \)
- \( \text{sP}: \text{sound P} \)

**Shows**
- \( \lambda s. \text{«G» s * wp body (} \lambda s. \text{bound-of P) s + } \lambda s. \text{N G} \text{ s * P s ⊨ } \lambda s. \text{bound-of P} \)

**Proof** (rule le-funI)

- \( \text{fix s} \)
- \( \text{from sP have sound (} \lambda s. \text{bound-of P) by(auto)} \)
- **Moreover hence bounded-by (bound-of P) (} \lambda s. \text{bound-of P) by(auto)} \)
- **Ultimately have bounded-by (} bound-of P) (wp body (} \lambda s. \text{bound-of P))
- **using hb by(auto)} \)
- **Hence wp body (} \lambda s. \text{bound-of P) s ≤ bound-of P by(auto)} \)
- **Moreover from sP have P s ≤ bound-of P by(auto)} \)
- **Ultimately have «G» s * wp body (} \lambda a. \text{bound-of P) s + (} 1 - «G» s \text{) * P s ≤ «G» s * bound-of P + (} 1 - «G» s \text{) * bound-of P)
- by(blast intro:add-mono mult-left-mono)} \)
- **Thus «G» s * wp body (} \lambda a. \text{bound-of P) s + } \lambda s. \text{N G} \text{ s * P s ≤ bound-of P)}
- by(simp add:algebra-simps negate-embed)}}
qed

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

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

lemma wlp-loop-step-unitary:
fixes t u ::'s trans
assumes hb: nearly-healthy (wlp body)
and ht: P. sound t P ⊢ sound (t P)
and uP: sound P
shows sound (wlp (body ;; Embed t «Gs» ⊕ Skip) P)
proof(intro soundI2 nnegI bounded-byI, simp-all add:wp-eval)
fix s::'
s from ht uP have atP: unitary (t P) by(auto)
with hb have unitary (wlp body (t P)) by(auto)
hence 0 ≤ wlp body (t P) s by(auto)
with uP show 0 ≤ «Gs» s ∗ wp body (t P) s + (1 − «Gs» s) ∗ P s
by(auto intro!:add-nonneg-nonneg mult-nonneg-nonneg)
from ht uP have bounded-by 1 (t P) by(auto)
with utP hb have bounded-by 1 (wlp body (t P)) by(auto)
hence wlp body (t P) s ≤ 1 by(auto)
with uP have «Gs» s ∗ wp body (t P) s + (1 − «Gs» s) ∗ P s ≤ «Gs» s + 1 + (1 − «Gs» s) ∗ 1
by(blast intro:add-mono mult-left-mono)
also have ... = 1 by(simp)
finally show «Gs» s ∗ wp body (t P) s + (1 − «Gs» s) ∗ P s ≤ 1 .
qed

lemma wp-loop-step-sound:
fixes t u::'s trans
assumes hb: healthy (wp body)
and ht: P. sound t P ⊢ sound (t P)
and sP: sound P
shows sound (wp (body ;; Embed t «Gs» ⊕ Skip) P)
proof(intro soundI2 nnegI bounded-byI, simp-all add:wp-eval)
4.2. **HEALTHINESS**

fix s::'s
from ht sP have stP: sound (t P) by(auto)
with hb have 0 ≤ wp body (t P) s by(auto)
with sP show 0 ≤ « G » s * wp body (t P) s + (1 − « G » s) * P s
  by(auto intro!:add-nonneg-nonneg mult-nonneg-nonneg)

from ht sP have sound (t P) by(auto)
moreover hence bounded-by (bound-of (t P)) (t P) by(auto)
ultimately have wp body (t P) s ≤ bound-of (t P) using hb 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-nonneg mult-left-nonneg)
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
asummes unitary: unitary P
  and healthy: nearly-healthy (wlp body)
sows 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(simp only: wp-eval, intro gfp-pulldown assms le-funI)
  fix u P
  show wlp (?Z u) P = ?Y P (u P) by(simp add:wp-eval negate-embed)
next
fix t::'s trans and P::'s expect
assume 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 (wp body $Q$) by(auto)
hence $0 \leq \text{wp body } Q \ s$ by(auto)
with $uP$ show $0 \leq \langle G \rangle \ s \ast \ \text{wp body } Q \ s + \langle N G \rangle \ s \ast P \ s$
by(auto intro!:add-nonneg-nonneg mult-nonneg-nonneg)

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

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

lemma wp-loop-sound:
assumes sP: sound $P$
and hb: healthy (wp body)
shows sound (wp do $G$ \rightarrow body od $P$)
proof(simp only: wp-eval, intro lfp-trans-sound sP)
let $?v = \lambda P \ s. \ \text{bound-of } P$
show le-trans (wp (body :: Embed $?v \ G \oplus \text{Skip}) \ ?v)
by(intro le-transI, simp add:wp-eval lfp-loop-fp[unfolded negate-embed] hb)
show $\forall P. \ \text{sound } P \rightarrow \text{sound } (?v \ P)$ by(auto)
qed

Likewise, we can rewrite strict loops.

lemma wp-Loop1:
fixes body :: 's prog
assumes sP: sound $P$
4.2. HEALTHINESS

and healthy; healthy (wp body)
shows wp (do G → body od) P =
\( \text{lfp-exp} \ (\lambda Q. s. \ {G} ~ s * wp \ body ~ Q ~ s + \ {N} ~ G ~ s * P ~ s) \)
is \( \ ?X = \text{lfp-exp} \ (\ ?Y ~ P) \)

proof –
let \( ?Z ~ u = (\ \text{body} :: \ \text{Embed} ~ u \ {G} \oplus \text{Skip}) \)
show \( ?\text{thesis} \)
proof (simp only: wp-eval, intro lfp-pulldown assms le-funI sP mono-transI)
fix \( u \) P
show \( wp \ (\ ?Z ~ u) ~ P = \ ?Y ~ P \ (u ~ P) \) by (simp add: wp-eval negate-embed)
next
fix \( P \) s::’s expect
assume \( sP: \ \text{sound} \ P \) and \( sQ: \ \text{sound} \ Q \)
show \( \text{sound} \ (\ \lambda a. \ {G} ~ a * wp \ body \ Q ~ a + \ {N} ~ G ~ a * P ~ a) \)
proof (intro soundI2 nnegI bounded-byI)
fix \( s:’s \)
from \( sQ \) have \( \text{nneg} \ Q \ \text{bounded-by} \ (\text{bound-of} ~ Q) \ Q \) by (auto)
with healthy have \( \text{bounded-by} \ (\text{bound-of} ~ Q) \ (wp \ body \ Q) \) by (auto)
\( \text{hence wp body Q s} \leq \text{bound-of Q by} \) (auto)
\( \text{hence wp body Q s} \leq \text{max} \ (\text{bound-of} ~ P) \ (\text{bound-of} ~ Q) \) by (auto)
moreover {
from \( sP \) have \( P ~ s \leq \text{bound-of} ~ P \) by (auto)
\( \text{hence P s} \leq \text{max} \ (\text{bound-of} ~ P) \ (\text{bound-of} ~ Q) \) by (auto)
}
ultimately have \( \ {G} ~ s * wp \ body \ Q ~ s + \ {N} ~ G ~ s * P ~ s \leq \)
\( \ {G} ~ s * \text{max} \ (\text{bound-of} ~ P) \ (\text{bound-of} ~ Q) \)
\( \ {N} ~ G ~ s * \text{max} \ (\text{bound-of} ~ P) \ (\text{bound-of} ~ Q) \)
by (auto intro: add: mono mult-left mono)
also have \( ... = \text{max} \ (\text{bound-of} ~ P) \ (\text{bound-of} ~ Q) \) by (simp add: algebra-simps negate-embed)
finally show \( \ {G} ~ s * wp \ body \ Q ~ s + \ {N} ~ G ~ s * P ~ s \leq \text{max} \ (\text{bound-of} ~ P) \)
(\text{bound-of} ~ Q) .
from \( sP \) have \( 0 \leq P ~ s \) by (auto)
moreover from \( sQ \) healthy have \( 0 \leq wp \ body ~ Q ~ s \) by (auto)
ultimately show \( 0 \leq \ {G} ~ s * wp \ body ~ Q ~ s + \ {N} ~ G ~ s * P ~ s \)
by (auto intro: add: nonneg-mult nonneg-mult nonneg-mult)
qed
next
fix \( P \) Q R::’s expect and \( s::’s \)
assume \( sQ: \ \text{sound} \ Q \) and \( sR: \ \text{sound} \ R \)
and \( le: \ Q \vdash \ R \)
\( \text{hence wp body Q s} \leq \text{wp body R s} \)
by (blast intro: le-funD[OF mono-transD, OF healthy-monoD, OF healthy])
thus \( \ {G} ~ s * wp \ body ~ Q ~ s + \ {N} ~ G ~ s * P ~ s \leq \)}
«G» s * wp body R s + «N G» s * P s
by(auto intro:mult-left-mono)

next
fix t w::'s trans
assume le: le-trans t u
and st: \( \forall P. \text{sound } P \Rightarrow \text{sound } (t P) \)
and sw: \( \forall P. \text{sound } P \Rightarrow \text{sound } (u P) \)
with healthy show le-trans (wp (Z t)) (wp (Z u))
by(rule wp-loop-step-mono)

next
from healthy show le-trans (wp (Z t)) (wp (Z u))
by(intro le-transI, simp add:wp-eval lfp-loop-fp[unfolded negate-embed])

next
fix P::'s expect and s::'
assume sound P
thus sound \((\lambda s. \text{bound-of } P)\)
by(auto)
qed

qed

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

show \( \lambda s. \text{0} \vdash \text{gfp-exp (Z P)} \)
proof(rule gfp-exp-upperbound)
  show unitary \((\lambda s. \text{0}::\text{real})\)
  by(auto)
  with hb have unitary \((\text{wp body } (\lambda s. \text{0}))\)
  by(auto)
with uP show \( \lambda s. \text{0} \vdash \text{(Z P (\lambda s. \text{0}))} \)
  by(bl ast intro! :le-funI add-nonneg-nonneg mult-nonneg-nonneg)
qed

show gfp-exp (Z P) \( \vdash \lambda s. \text{1} \)
proof(rule gfp-exp-least)
  show unitary \((\lambda s. \text{1}::\text{real})\)
  by(auto)
fix Q::'s expect
assume unitary Q
thus \( Q \vdash \lambda s. \text{1} \)
by(auto)
qed

fix Q::'s expect
assume uQ: unitary Q and le: P \( \vdash Q \)
show gfp-exp (Z P) \( \vdash \text{gfp-exp (Z Q)} \)
proof(rule gfp-exp-least)
4.2. HEALTHINESS

fix R::'s expect assume uR: unitary R
assume fP: R ⊢ ?X P R
also from le have ... ⊢ ?X Q R
  by(blast intro:add-mono mult-left-mono le-funI)
finally show R ⊢ gfp-exp (?X Q)
  using uR by(auto intro:gfp-exp-upperbound)

next
  show unitary (gfp-exp (?X Q))
proof(rule gfp-exp-unitary, intro unitaryI2 nnegI bounded-byI)
fix R::'s expect and s::'s assume uR: unitary R
with hh have ubP: unitary (wp body R) by(auto)
with uQ show 0 ≤ « G » s * wp body R s + « N/G » s * Q s
  by(blast intro:add-nonneg-nonneg mult-nonneg-nonneg)
from ubP uQ have wp body R s ≤ 1 Q s ≤ 1 by(auto)
hence « G » s * wp body R s + « N/G » s * Q s ≤ « G » s * 1 + « N/G » s
  * 1
  by(blast intro:add-mono mult-left-mono)
thus « G » s * wp body R s + « N/G » s * Q s ≤ 1
  by(simp add:negate-embed)
qed
qed

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

lemma healthy-wp-loop:
  fixes body::'s prog
  assumes hh: healthy (wp body)
  shows healthy (wp (do G → body od))
proof
  show ?thesis
proof(intro healthy-parts bounded-byI2 nnegI2, simp-all add:wp-Loop1 hh soundI2 sound-intros)
fix P::'s expect and c::real and s::'s
assume sP: sound P and anc: 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: \( A P \ c. c \neq 0 \implies 0 \leq c \implies \text{sound } P \implies \)

\[ \lambda a. \ c \ast \lfpexp(\lambda a \ b. \ G_b \ b \ast \wpbody a \ b + \langle N \ G_b \ b \ast P \ b \rangle) \ a \vdash \]

\[ \lfpexp(\lambda a \ b. \ G_b \ b \ast \wpbody a \ b + \langle N \ G_b \ b \ast (c \ast P \ b) \rangle) \]

proof

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

assume \( cNZ: c \neq 0 \) and \( \text{nnc}: 0 \leq c \) and \( \text{sP: sound } P \)

with \( \text{nnc} \) have \( \text{cpos}: 0 < c \) by(auto)

hence \( \text{nnc}: 0 \leq \text{inverse } c \) by(auto)

show \( \lambda a. \ c \ast \lfpexp(\lambda a \ b. \ G_b \ b \ast \wpbody a \ b + \langle N \ G_b \ b \ast P \ b \rangle) \ a \vdash \]

\[ \lfpexp(\lambda a \ b. \ G_b \ b \ast \wpbody a \ b + \langle N \ G_b \ b \ast (c \ast P \ b) \rangle) \]

proof (rule \( \text{lfp-exp-greatest} \))

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

assume \( sQ \) \( \text{sound } Q \)

and \( \text{fp}: \lambda b. \ G_b \ b \ast \wpbody Q \ b + \langle N \ G_b \ b \ast (c \ast P \ b) \rangle \vdash Q \)

hence \( \bigwedge s. \ (G_s \ s \ast \wpbody Q \ s + \langle N \ G_s \ s \ast (c \ast P \ s) \rangle \leq Q \ s \) by(auto)

with \( \text{nnc} \)

have \( \bigwedge s. \ \text{inverse } c \ast (G_s \ s \ast \wpbody Q \ s + \langle N \ G_s \ s \ast (c \ast P \ s) \rangle \leq \)

\[ \text{inverse } c \ast Q \ s \]

by(auto intro:\text{mult-left-mono})

hence \( \bigwedge s. \ (G_s \ s \ast (\text{inverse } c \ast \wpbody Q \ s) \rangle + (\text{inverse } c \ast c) \ast (N \ G_s \ s \ast P \ s) \leq \)

\[ \text{inverse } c \ast Q \ s \]

by(simp add:algebra-simps)

hence \( \bigwedge s. \ (G_s \ s \ast \wpbody (\lambda s. \ \text{inverse } c \ast Q \ s) \ s + \langle N \ G_s \ s \ast P \ s \rangle \leq \)

\[ \text{inverse } c \ast Q \ s \]

by(simp add:cnz scalingD[OF healthy-scalingD, OF \( \text{hb sQ nnic} \)])

hence \( \lambda s. \ (G_s \ s \ast \wpbody (\lambda s. \ \text{inverse } c \ast Q \ s) \ s + \langle N \ G_s \ s \ast P \ s \rangle \vdash \)

\[ \lambda s. \ \text{inverse } c \ast Q \ s \]

by(rule le-funI)

moreover from \( \text{nnic sQ have sound } (\lambda s. \ \text{inverse } c \ast Q \ s) \)

by(proper intro:sound-intros)

ultimately have \( \text{lfp-exp}(\lambda a \ b. \ G_b \ b \ast \wpbody a \ b + \langle N \ G_b \ b \ast P \ b \rangle \vdash \)

\[ \lambda s. \ \text{inverse } c \ast Q \ s \]

by(rule \( \text{lfp-exp-lowerbound} \))

hence \( \bigwedge s. \ \text{lfp-exp}(\lambda a \ b. \ G_b \ b \ast \wpbody a \ b + \langle N \ G_b \ b \ast P \ b \rangle \ s \leq \)

\[ \text{inverse } c \ast Q \ s \]

by(rule le-funD)

with \( \text{nnc} \)

have \( \bigwedge s. \ c \ast \text{lfp-exp}(\lambda a \ b. \ G_b \ b \ast \wpbody a \ b + \langle N \ G_b \ b \ast P \ b \rangle) \ s \leq \)

\[ c \ast (\text{inverse } c \ast Q \ s) \]

by(auto intro:\text{mult-left-mono})

also from \( \text{cnz have } \bigwedge s. \ldots \ s = Q \ s \) by(simp)

finally show \( \lambda a. \ c \ast \text{lfp-exp}(\lambda a \ b. \ G_b \ b \ast \wpbody a \ b + \langle N \ G_b \ b \ast P \ b \rangle \ a \vdash \)

\[ Q \]

by(rule le-funI)

next

from \( \text{sP have sound } (\lambda s. \ \text{bound-of } P) \) by(auto)

with \( \text{hb sP have sound } (\text{lfp-exp } (\exists X \ P)) \)

by(blast intro:lfp-exp-sound lfp-loop-fp)
4.2. HEALTHINESS

with nnc show sound \((\lambda s. \ c \ast \ lfp\exp\ (\exists X P) s)\)
by(auto intro!:sound-intros)

from hb sP nnc

show \((\lambda s. \ \langle G \rangle \ast s \ast wp\ body \ (\lambda s. \ \text{bound-of} \ (\lambda s. \ c \ast P s) s) \ast \langle N \rangle G \rangle \ast (c \ast P s) \not\vdash \lambda s. \ \text{bound-of} \ (\lambda s. \ c \ast P s)\)
by(iprove intro:ifp-loop-fp sound-intros)

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

qed

assume nzc: \(c \not= 0\)
show \(?\text{thesis}\) \((\exists X P c s = \exists Y P c s)\)
proof \((\text{rule fun-cong}[\text{where}\ x=s], \text{rule antisym})\)
from nzc nnc sP show \(?X P c \vdash \exists Y P c\) by(rule onesided)

from nzc have nzc: inverse \(c \not= 0\) by(auto)
moreover with nnc have nnic: \(0 \leq \text{inverse} \ c\) by(auto)
moreover from nnc sP have scP: sound \((\lambda s. \ c \ast P s)\) by(auto intro!:sound-intros)
ultimately have \(?X (\lambda s. \ c \ast P s) (\text{inverse} \ c) \vdash ?Y (\lambda s. \ c \ast P s) (\text{inverse} \ c)\)
by(rule onesided)
with nnc have \((\lambda s. \ c \ast \langle ?X (\lambda s. \ c \ast P s) \ast \text{inverse} \ c) \ast s \vdash \lambda s. \ c \ast \langle ?Y (\lambda s. \ c \ast P s) \ast \text{inverse} \ c\) \ast s)\)
by(blast intro:mult-left-mono)
with nze show \(?Y P c \vdash ?X P c\) by(simp add:mult.assoc[symmetric])

qed

next

fix \(P::s\) expect and \(b::\text{real}\)
assume \(bP::\text{bounded-by} \ b \ P \ and\ nP::\text{nmeg} \ P\)
show \(\text{lfp-exp} \ (\lambda Q s. \ \langle G \rangle \ast s \ast wp\ body \ Q s + \langle N \rangle G \rangle \ast \langle N \rangle G \rangle s * P s) \not\vdash \lambda s. \ b\)
proof(intro lfp-exp-lowerbound le-fun1)
fix \(s::s\)
from \(bP nP hb\) have \(\text{bounded-by} \ b \ (wp\ body \ (\lambda s. \ b))\) by(auto)
hence \(wp\ body \ (\lambda s. \ b) s \leq b\) by(auto)
moreover from \(bP\) have \(P s \leq b\) by(auto)
ultimately have \(\langle G \rangle s * wp\ body \ (\lambda s. \ b) s + \langle N \rangle G \rangle s * P s \leq \langle G \rangle s * b + \langle N \rangle G \rangle s * b\)
by(auto intro!:add-mono mult-left-mono)
also have \(\ldots = b\) by(simp add:negate-embed field-simps)
finally show \(\langle G \rangle s * wp\ body \ (\lambda s. \ b) s + \langle N \rangle G \rangle s * P s \leq b\)
from \(bP nP\) have \(0 \leq b\) by(auto)
thus \(\text{sound} \ (\lambda s. \ b)\) by(auto)

qed

from \(hb bP nP\) show \(\lambda s. \ 0 \vdash \text{lfp-exp} \ (\lambda Q s. \ \langle G \rangle s \ast wp\ body \ Q s + \langle N \rangle G \rangle\)
CHAPTER 4.  THE PGCL LANGUAGE

\[
s \ast P s
\]
by (auto dest!: sound-nneg intro!: lfp-loop-greatest)

next
fix \( P Q \) ::
\[
\begin{align*}
\text{assume } & sP : \text{sound } P \text{ and } sQ : \text{sound } Q \text{ and } \text{le}: P \vdash Q \\
\text{show } & (\text{lfp-exp } (?X P) \vdash \text{lfp-exp } (?X Q))
\end{align*}
\]
proof (rule lfp-exp-greatest)
fix \( R \) ::
\[
\begin{align*}
\text{assume } & sR : \text{sound } R \\
\text{and } & \text{fp}: \lambda s. \langle G \rangle s \ast \text{wp body } R \ s + \langle N \ G \rangle s \ast Q \ s \vdash R \\
\text{from } & \text{le have } \lambda s. \langle G \rangle s \ast \text{wp body } R \ s + \langle N \ G \rangle s \ast P \ s \vdash \\
& \lambda s. \langle G \rangle s \ast \text{wp body } R \ s + \langle N \ G \rangle s \ast Q \ s
\end{align*}
\]
by (auto intro: le-funI add-left-mono mult-left-mono)
also note \( \text{fp} \)
finally show \( \text{lfp-exp } (\lambda R \ s. \langle G \rangle s \ast \text{wp body } R \ s + \langle N \ G \rangle s \ast P \ s) \vdash R \)
using \( sR \) by (auto intro: lfp-exp-lowerbound)
next
from \( \text{hb } sP \) show \( \text{sound } (\text{lfp-exp } (\lambda R \ s. \langle G \rangle s \ast \text{wp body } R \ s + \langle N \ G \rangle s \ast P \ s)) \)
by (rule lfp-loop-sound)
from \( \text{hb } sQ \) show \( \lambda s. \langle G \rangle s \ast \text{wp body } (\lambda s. \text{bound-of } Q) \ s + \langle N \ G \rangle s \ast Q \ s \vdash \lambda s. \text{bound-of } Q \)
by (rule lfp-loop-fp)
from \( sQ \) show \( \text{sound } (\lambda s. \text{bound-of } Q) \) by (auto)
qed
qed
qed

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

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

end

4.3 Continuity

theory Continuity imports Healthiness begin

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

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

**Definition** \(bd-cts \::\ 's\ \text{trans} \Rightarrow\ \text{bool}\)
where \(bd-cts\ t = (\forall M. (\forall i. (M \ i \mapsto M \ Suc \ i) \land\ \text{sound} \ (M \ i)) \Rightarrow \ (\exists b. \forall i. \text{bounded-by} \ b \ (M \ i)) \Rightarrow\ t\ (\text{Sup-exp} \ (\text{range} \ M)) = \text{Sup-exp} \ (\text{range} \ (t \ o\ M)))\)

**Lemma** \(bd-ctsD:\)
\[
[\ bd-cts\ t; \ \forall i. (M \ i \mapsto M \ Suc \ i); \ \forall i. \text{sound} \ (M \ i); \ \forall i. \text{bounded-by} \ b \ (M \ i) ] \Rightarrow\ t\ (\text{Sup-exp} \ (\text{range} \ M)) = \text{Sup-exp} \ (\text{range} \ (t \ o\ M))
\]
unfolding \(bd-cts-def\) by (auto)

**Lemma** \(bd-ctsI:\)
\[
(\forall M. (\forall i. (M \ Suc \ i) \mapsto M \ Suc \ i); \ \forall i. \text{sound} \ (M \ i); \ \forall i. \text{bounded-by} \ b \ (M \ i)) \Rightarrow\ t\ (\text{Sup-exp} \ (\text{range} \ M)) = \text{Sup-exp} \ (\text{range} \ (t \ o\ M)) \Rightarrow\ bd-cts\ t
\]
unfolding \(bd-cts-def\) by (auto)

A generalised property for transformers of transformers.

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

**Lemma** \(bd-cts-trD:\)
\[
[\ bd-cts-tr\ T; \ \forall i. \text{le-trans} \ (M \ Suc \ i) \mapsto M \ Suc \ i); \ \forall i. \text{feasible} \ (M \ Suc \ i) ] \Rightarrow\ \text{equiv-trans} \ (T\ (\text{Sup-trans} \ (M \ \text{UNIV}))) \ (\text{Sup-trans} \ ((T \ o\ M) \ \text{UNIV}))
\]
by (simp add: bd-cts-tr-def)

**Lemma** \(bd-cts-trI:\)
\[
(\forall M. (\forall i. \text{le-trans} \ (M \ Suc \ i) \mapsto M \ Suc \ i); \ \forall i. \text{feasible} \ (M \ Suc \ i)) \Rightarrow\ \text{equiv-trans} \ (T\ (\text{Sup-trans} \ (M \ \text{UNIV}))) \ (\text{Sup-trans} \ ((T \ o\ M) \ \text{UNIV}))
\]
\[\Rightarrow\ bd-cts-tr\ T\]
by (simp add: bd-cts-tr-def)

4.3.1 Continuity of Primitives

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

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

**Lemma** \(cts-wp-Apply:\

\[ \text{bd-cts } (\text{wp } \text{Apply } f) \]

**proof**
- have \( X \cap \{ P \mid P \in \text{range } M \} = \{ P \mid P \in \text{range } (\lambda i. M i (f s)) \} \)
- show \( \text{thesis} \) by (intro bd-ctsI, simp add: wp-eval o-def Sup-exp-def X)

q.e.d.

**lemma** cts-wp-Bind:
- fixes \( a : 'a \Rightarrow 's \text{ prog} \)
- assumes \( ca : \bigwedge s. \text{bd-cts } (\text{wp } a (f s)) \)
- shows \( \text{bd-cts } (\text{wp } (\text{Bind } f a)) \)

**proof** (rule bd-ctsI)
- fix \( M : \text{nat } \Rightarrow 's \text{ expect} \)
- assume \( \text{chain } : \bigwedge i. M i \vdash M (\text{Suc } i) \) and \( sM : \bigwedge i. \text{sound } (M i) \)
- and \( bM : \bigwedge i. \text{bounded-by } c (M i) \)
- with \( \text{bd-ctsD[OF } ca] \)
- have \( \bigwedge s. \text{wp } (a (f s)) (\text{Sup-exp } (\text{range } M)) = \text{Sup-exp } (\text{range } (\text{wp } (a (f s)) o M)) \)
  by (auto)
- moreover have \( \bigwedge s. \{ fa s \mid \text{fa } \in \text{ range } (\lambda x. \text{wp } a (f s)) (M x) \} = \{ fa s \mid \text{fa } \in \text{ range } (\lambda x s. \text{wp } a (f s)) (M x) s \} \)
  by (auto)
- ultimately show \( \text{wp } (\text{Bind } f a) (\text{Sup-exp } (\text{range } M)) = \text{Sup-exp } (\text{range } (\text{wp } (\text{Bind } f a) o M)) \)
  by (simp add: wp-eval o-def Sup-exp-def)

q.e.d.

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 \( \text{bd-cts } (\text{wp } (a \sqcup b)) \)

**proof** (rule bd-ctsI, rule antisym)
- fix \( M : \text{nat } \Rightarrow 's \text{ expect} \) and \( c : \text{real} \)
- assume \( \text{chain } : \bigwedge i. M i \vdash M (\text{Suc } i) \) and \( sM : \bigwedge i. \text{sound } (M i) \)
- and \( bM : \bigwedge i. \text{bounded-by } c (M i) \)
- from \( ha \) have \( h b : \text{healthy } (\text{wp } (a \sqcup b)) \) by (rule healthy-intros)
- from \( bM \) have \( \text{leSup } : \bigwedge i. M i \vdash \text{Sup-exp } (\text{range } M) \) by (auto intro: Sup-exp-upper)
- from \( sM bM \) have \( sSup : \text{sound } (\text{Sup-exp } (\text{range } M)) \) by (auto intro: Sup-exp-sound)
- show \( \text{Sup-exp } (\text{range } (\text{wp } (a \sqcup b) o M)) \vdash \text{wp } (a \sqcup b) (\text{Sup-exp } (\text{range } M)) \)
  proof (rule Sup-exp-least, clarsimp, rule le-funI)
  fix \( i s \)
4.3. CONTINUITY

from mono-transD[OF healthy-monoD[OF hab]] leSup sM sSup
have wp (a ∩ b) (M i) ⊆ wp (a ∩ b) (Sup-exp (range M)) by(auto)
thus wp (a ∩ b) (M i) s ≤ wp (a ∩ b) (Sup-exp (range M)) s by(auto)

from hab sSup have sound (wp (a ∩ b) (Sup-exp (range M))) by(auto)
thus nneg (wp (a ∩ b) (Sup-exp (range M))) by(auto)
qed

from sM bM ha have ∀i. bounded-by c (wp a (M i)) by(auto)
 hence baM: ∀i s. wp a (M i) s ≤ c by(auto)
 from sM bM hb have ∀i. bounded-by c (wp b (M i)) by(auto)
 hence bbM: ∀i s. wp b (M i) s ≤ c by(auto)

show wp (a ∩ b) (Sup-exp (range M)) ⊆ Sup-exp (range (wp (a ∩ b) o M))
proof(simp add:wp-eval o-def, rule le-funI)
 fix s:'s
 from bd-ctsD[OF ca, of M, OF chain sM bM] bd-ctsD[OF cb, of M, OF chain sM bM]
 have min (wp a (Sup-exp (range M))) (wp b (Sup-exp (range M))) s =
 min (Sup-exp (range (wp a o M))) (Sup-exp (range (wp b o M))) s
 by(simp)
 also { 
 have {f s | f ∈ range λx. wp a (M x)} = range (λi. wp a (M i) s)
 f s | f ∈ range λx. wp b (M x)} = range (λi. wp b (M i) s)
 by(auto)
 hence min (Sup-exp (range (wp a o M))) (Sup-exp (range (wp b o M))) s
 =
 min (Sup (range (λi. wp a (M i) s))) (Sup (range (λi. wp b (M i) s)))
 by(simp add:Sup-exp-def o-def)
 }
 also { 
 have (λi. wp a (M i) s) ---> Sup (range (λi. wp a (M i) s))
 proof(rule increasing-LIMSEQ)
 fix n
 from mono-transD[OF healthy-monoD, OF ha] sM chain
 show wp a (M n) s ≤ wp a (M (Suc n)) s by(auto intro:le-funID)
 from baM show wp a (M n) s ≤ Sup (range (λi. wp a (M i) s))
 by(intro cSup-upper bdd-aboveI, auto)

 fix c::real assume pe: 0 < e
 from baM have cSup: Sup (range (λi. wp a (M i) s)) ∈ closure (range (λi. wp a (M i) s))
 by(blast intro:closure-contains-Sup)
 with pe obtain y where yin: y ∈ (range (λi. wp a (M i) s))
 and dy: dist y (Sup (range (λi. wp a (M i) s))) < e
 by(blast dest:iffD[OF closure-approachable])
 from yin obtain i where y = wp a (M i) s by(auto)
 with dy have dist (wp a (M i) s) (Sup (range (λi. wp a (M i) s))) < e
 by(simp)
moreover from \( baM \) have \( wp \ a \ (M \ i) \ s \leq \operatorname{Sup} \ (\lambda i. \ wp \ a \ (M \ i) \ s) \)
    by(intro cSup-upper bdd-aboveI, auto)

ultimately have \( \operatorname{Sup} \ (\lambda i. \ wp \ a \ (M \ i) \ s) \leq wp \ a \ (M \ i) \ s + e \)
    by(simp add:dist-real-def)

thus \( \exists i. \operatorname{Sup} \ (\lambda i. \ wp \ a \ (M \ i) \ s) \leq wp \ a \ (M \ i) \ s + e \) by(auto)
qed

moreover
have \( (\lambda i. \ wp \ b \ (M \ i) \ s) \longrightarrow \operatorname{Sup} \ (\lambda i. \ wp \ b \ (M \ i) \ s) \)
proof(rule increasing-LIMSEQ)

fix \( n \)
from mono-transD[OF healthy-monoD, OF \( hb \)] \( sM \) chain

show \( wp \ b \ (M \ n) \ s \leq wp \ b \ (M \ (\operatorname{Suc} \ n)) \) by(auto intro:le-fanD)
from \( bbM \) show \( wp \ b \ (M \ m) \ s \leq \operatorname{Sup} \ (\lambda i. \ wp \ b \ (M \ i) \ s) \)
    by(intro cSup-upper bdd-aboveI, auto)

fix \( c::\operatorname{real} \) assume \( pe: \ 0 < e \)
from \( bbM \) have \( c\Sigma p: \operatorname{Sup} \ (\lambda i. \ wp \ b \ (M \ i) \ s) \) \( \in \) closure \( (\lambda i. \ wp \ b \ (M \ i) \ s) \)
    by(blast intro:closure-contains-Sup)

with \( pe \) obtain \( y \) where \( \gamma in: \ y \in \ (\lambda i. \ wp \ b \ (M \ i) \ s) \)
    and \( dy: \operatorname{dist} \ y \ (\operatorname{Sup} \ (\lambda i. \ wp \ b \ (M \ i) \ s)) < e \)
    by(blast dest:iffD[OF closure-approachable])
from \( \gamma in \) obtain \( i \) where \( y = wp \ b \ (M \ i) \ s \) by(auto)
with \( dy \) have \( \operatorname{dist} \ (wp \ b \ (M \ i) \ s) \ (\operatorname{Sup} \ (\lambda i. \ wp \ b \ (M \ i) \ s)) < e \)
    by(simp)

moreover from \( bbM \) have \( wp \ b \ (M \ i) \ s \leq \operatorname{Sup} \ (\lambda i. \ wp \ b \ (M \ i) \ s) \)
    by(intro cSup-upper bdd-aboveI, auto)
ultimately have \( \operatorname{Sup} \ (\lambda i. \ wp \ b \ (M \ i) \ s) \leq wp \ b \ (M \ i) \ s + e \)
    by(simp add:dist-real-def)
thus \( \exists i. \operatorname{Sup} \ (\lambda i. \ wp \ b \ (M \ i) \ s) \leq wp \ b \ (M \ i) \ s + e \) by(auto)
qed

ultimately have \( (\lambda i. \ \min \ (wp \ a \ (M \ i) \ s) \ (wp \ b \ (M \ i) \ s)) \longrightarrow \)
    \( \min \ (\operatorname{Sup} \ (\lambda i. \ wp \ a \ (M \ i) \ s)) \ (\operatorname{Sup} \ (\lambda i. \ wp \ b \ (M \ i) \ s)) \)
by(rule tendsto-min)

moreover have \( bdd-above \ (\lambda i. \ \min \ (wp \ a \ (M \ i) \ s) \ (wp \ b \ (M \ i) \ s)) \)
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 \( ha sM \) \( bbM \) have bounded-by \( c \ (wp \ a \ (M \ i) \ s) \) 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 \( \min \ (\operatorname{Sup} \ (\lambda i. \ wp \ a \ (M \ i) \ s)) \ (\operatorname{Sup} \ (\lambda i. \ wp \ b \ (M \ i) \ s)) \)
\( \leq \)
\( \operatorname{Sup} \ (\lambda i. \ \min \ (wp \ a \ (M \ i) \ s) \ (wp \ b \ (M \ i) \ s)) \)
by (blast intro: LIMSEQ_le_const2 cSup-upper min.mono[OF baM bbM])

also {  
  have range (λi. min (wp a (M i) s) (wp b (M i) s)) =  
    {f s | f ∈ range (λi. s. min (wp a (M i) s) (wp b (M i) s))}  
  by (auto)  
  hence Sup (range (λi. min (wp a (M i) s) (wp b (M i) s))) =  
    Sup-exp (range (λi. s. min (wp a (M i) s) (wp b (M i) s)))  
  by (simp add: Sup-exp-def cong del: SUP-cong-simp)  
}

finally show min (wp a (Sup-exp (range M)) s) (wp b (Sup-exp (range M)) s) ≤  
  Sup-exp (range (λi. s. min (wp a (M i) s) (wp b (M i) s))) s .

qed

lemma cts-wp-Seq:  
fixes a b :: 's prog  
assumes ca: bd-cts (wp a)  
  and cb: bd-cts (wp b)  
  and hb: healthy (wp b)  
shows bd-cts (wp (a ;; b))

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

qed

lemma cts-wp-PC:  
fixes a b :: 's prog  
assumes ca: bd-cts (wp a)  
  and cb: bd-cts (wp b)
\section*{CHAPTER 4. THE PGCL LANGUAGE}

\begin{quote}
\texttt{and ha: healthy (wp a)}
\texttt{and bb: healthy (wp b)}
\texttt{and wp: unitary p}
\texttt{shows bd-cts (wp (PC a p b))}
\end{quote}

\textbf{proof} (rule bd-ctsI, rule ext, simp add:o-def wp-eval)

\texttt{fix M::nat \Rightarrow 's expect and c::real and s:'s}

\texttt{assume chain: } \land_i. M i \vdash M (Suc i) \textbf{and } sM: \land_i. \text{sound} (M i)

\texttt{and bM: } \land_i. \text{bounded-by } c (M i)

\texttt{from sM have } \land_i. \text{nneg} (M i) \texttt{ by(auto)}

\texttt{with bM have } nc: 0 \leq c \texttt{ by(auto)}

\texttt{from chain sM bM have wp a (Sup-exp (range M)) = Sup-exp (range (wp a o M))}

\texttt{by(rule bd-ctsD[OF ca])}

\texttt{hence wp a (Sup-exp (range M)) s = Sup-exp (range (wp a o M)) s}

\texttt{by(simp)}

\texttt{also \{}

\texttt{have \{ f s |. f \in range \langle \lambda x. wp a (M x) \rangle \} = range \langle \lambda i. wp a (M i) s \rangle}

\texttt{by(auto)}

\texttt{hence Sup-exp (range (wp a o M)) s = Sup (range (\lambda i. wp a (M i) s))}

\texttt{by(simp add:Sup-exp-def o-def)}

\texttt{\}}

\texttt{finally have } p s * wp a (Sup-exp (range M)) s =

\texttt{p s * Sup (range (\lambda i. wp a (M i) s)) by(simp)}

\texttt{also have \ldots = Sup \{ p s * x |. x \in range (\lambda i. wp a (M i) s)\}}

\texttt{proof (rule cSup-mult, blast, clarsimp)}

\texttt{from wp show } \theta \leq p s \texttt{ by(auto)}

\texttt{fix i}

\texttt{from sM bM ha have bounded-by c (wp a (M i)) by(auto)}

\texttt{thus wp a (M i) s \leq c by(auto)}

\texttt{qed}

\texttt{also \{}

\texttt{have \{ p s * x |. x \in range (\lambda i. wp a (M i) s)\} = range (\lambda i. p s * wp a (M i) s)}

\texttt{by(auto)}

\texttt{hence Sup \{ p s * x |. x \in range (\lambda i. wp a (M i) s)\} =

\texttt{Sup (range (\lambda i. p s * wp a (M i) s)) by(simp)}}

\texttt{\}}

\texttt{finally have p s * wp a (Sup-exp (range M)) s = Sup (range (\lambda i. p s * wp a (M i) s))}.

\texttt{moreover \{}

\texttt{from chain sM bM have wp b (Sup-exp (range M)) = Sup-exp (range (wp b o M))}

\texttt{by(rule bd-ctsD[OF cb])}

\texttt{hence wp b (Sup-exp (range M)) s = Sup-exp (range (wp b o M)) s}

\texttt{by(simp)}

\texttt{also \{}

\texttt{have \{ f s |. f \in range (\lambda x. wp b (M x))\} = range (\lambda i. wp b (M i) s)}

\texttt{\}}
4.3. CONTINUITY

\[
\text{by(auto)}
\]
\[
\text{hence } \text{Sup-exp (range (wp b o M)) } s = \text{Sup (range (λi. wp b (M i) s))}
\]
\[
\text{by(simp add:Sup-exp-def o-def)}
\]

}\)
finally have \((1 - p s) * wp b (\text{Sup-exp (range M)}) s =

\((1 - p s) * \text{Sup (range (λi. wp b (M i) s))}\) by(simp)
also have \(\ldots = \text{Sup \{\((1 - p s) * x | x \in \text{range (λi. wp b (M i) s))\}\}}

\text{proof(rule cSup-mult, blast, clarsimp)}
from \text{up show } 0 \leq 1 - p s
\text{by auto}
fix \(i\)
from \(sM bM hh\) have \(\text{bounded-by c (wp b (M i))}\) by(auto)
thus \(\text{wp b (M i) s} \leq c\) by(auto)
qed
also \{
have \(\{\{(1 - p s) * x | x \in \text{range (λi. wp b (M i) s))\}\} =
\text{range (λi. (1 - p s) * wp b (M i) s)}
\text{by(auto)}
\}
\text{hence } \text{Sup \{\{(1 - p s) * x | x \in \text{range (λi. wp b (M i) s))\}\} =}
\text{Sup (range (λi. (1 - p s) * wp b (M i) s))}\) by(simp)
}\)
finally have \((1 - p s) * wp b (\text{Sup-exp (range M)}) s =
\text{Sup (range (λi. (1 - p s) * wp b (M i) s))}.
\}
ultimately
have \(p s * wp a (\text{Sup-exp (range M)}) s + (1 - p s) * wp b (\text{Sup-exp (range M)}) s =
\text{Sup (range (λi. p s * wp a (M i) s))} + \text{Sup (range (λi. (1 - p s) * wp b (M i) s))}
\text{by(simp)}
\) also \{
from \(bM sM ha\) have \(\text{∀i. bounded-by c (wp a (M i))}\) by(auto)
hence \(\text{∀i. wp a (M i) s} \leq c\) by(auto)
moreover from \text{up have } 0 \leq p s\) by(auto)
ultimately have \(\text{∀i. p s * wp a (M i) s} \leq p s * c\) by(auto intro:mult-left-mono)
also from \text{up nc have } p s * c \leq 1 * c\) by(blast intro:mult-right-mono)
also have \(\ldots = c\) by(simp)
finally have \text{baM: } \text{∀i. p s * wp a (M i) s} \leq c .
\)

have \text{lina: } \text{(λi. p s * wp a (M i) s)} \longrightarrow \text{Sup (range (λi. p s * wp a (M i) s))}
\text{proof(rule increasing-LIMSEQ)}
fix \(n\)
from \sM chain healthy-monoD[\text{OF ha}]\) have \(\text{wp a (M n) \vdash wp a (M (Suc n))}\)
by(auto)
with \text{up show } p s * wp a (M n) s \leq p s * wp a (M (Suc n)) s
by(blast intro:mult-left-mono)
from \text{baM show } p s * wp a (M n) s \leq \text{Sup (range (λi. p s * wp a (M i) s))}
by(intro cSup-upper bdd-above1, auto)
next
fix e::real
assume pe: 0 < e
from baM have \(\text{Sup (range } (\lambda i. p\ s \ast wp\ a\ (M\ i)\ s))\in\)
closure (range (\lambda i. p\ s \ast wp\ a\ (M\ i)\ s))
  by(blast intro:closure-contains-Sup)
thm closure-approachable
with pe obtain y where yin: y \in range (\lambda i. p\ s \ast wp\ a\ (M\ i)\ s)
    and dy: dist y (Sup (range (\lambda i. p\ s \ast wp\ a\ (M\ i)\ s))) < e
  by(blast dest:iffD[OF closure-approachable])
from yin obtain i where y = p\ s \ast wp\ a\ (M\ i)\ s
  by(auto)
with dy have dist (p\ s \ast wp\ a\ (M\ i)\ s) (Sup (range (\lambda i. p\ s \ast wp\ a\ (M\ i)\ s))) < e
  by(simp)
moreover from baM have p\ s \ast wp\ a\ (M\ i)\ s \leq \text{Sup (range } (\lambda i. p\ s \ast wp\ a\ (M\ i)\ s))
  by(intro cSup-upper bdd-above1, auto)
ultimately have Sup (range (\lambda i. p\ s \ast wp\ a\ (M\ i)\ s)) \leq p\ s \ast wp\ a\ (M\ i)\ s + e
  by(simp add:dist-real-def)
thus \(\exists i.\ \text{Sup (range } (\lambda i. p\ s \ast wp\ a\ (M\ i)\ s)) \leq p\ s \ast wp\ a\ (M\ i)\ s + e\)
by(auto)
qed

from bM sM hb have \(\text{\(\forall i.\ bounded-by\ c\ (wp\ b\ (M\ i))\ by(auto)\)}\)
hence \(\forall i.\ wp\ b\ (M\ i)\ s \leq c\ by(auto)\)
moreover from wp have \(\theta \leq (1 - p\ s)\)
  by auto
ultimately have \(\forall i.\ (1 - p\ s) \ast wp\ b\ (M\ i)\ s \leq (1 - p\ s) \ast c\ by(auto intro:mult-left-mono)\)
also \{ from wp have \(1 - p\ s \leq 1\ by(auto)\)
  with nc have \((1 - p\ s) \ast c \leq 1 \ast c\ by(blast intro:mult-right-mono)\} \}
also have \(1 \ast c = c\ by(simp)\)
finally have bbM: \(\forall i.\ (1 - p\ s) \ast wp\ b\ (M\ i)\ s \leq c\ .\)

have \(\text{limb: } (\lambda i.\ (1 - p\ s) \ast wp\ b\ (M\ i)\ s) \longrightarrow \text{Sup (range } (\lambda i.\ (1 - p\ s) \ast wp\ b\ (M\ i)\ s))\)
proof(rule increasing-LIMSEQ)
fix n
from sM chain healthy-veroD[OF hb] have wp b (M n) \leq wp b (M (Suc n))
  by(auto)
moreover from wp have \(\theta \leq 1 - p\ s\)
  by auto
ultimately show \((1 - p\ s) \ast wp\ b\ (M\ n)\ s \leq (1 - p\ s) \ast wp\ b\ (M\ (Suc\ n))\)
  by(blast intro:mult-left-mono)
from bbM show \((1 - p\ s) \ast wp\ b\ (M\ n)\ s \leq \text{Sup (range } (\lambda i.\ (1 - p\ s) \ast wp\ b\ (M\ i)\ s))\)
4.3. CONTINUITY

\( b \ (M \ i \ s) \)

\( \text{by (intro cSup-upper bdd-aboveI, auto)} \)

next

fix \( e :: \text{real} \)

assume \( p e: 0 < e \)

from \( bbM \) have \( \text{Sup} \ (\lambda i. (1 - p \ s) * wp b \ (M \ i \ s)) \in \)

\( \text{closure} \ (\lambda i. (1 - p \ s) * wp b \ (M \ i \ s)) \)

\( \text{by (blast intro: closure-contains-Sup)} \)

with \( p e \) obtain \( y \) where \( yin: y \in \text{range} \ (\lambda i. (1 - p \ s) * wp b \ (M \ i \ s)) \)

\( \text{and dist y} (\text{Sup} \ (\lambda i. (1 - p \ s) * wp b \ (M \ i \ s))) < e \)

\( \text{by (blast dest: iff D1 [OF closure-approachable])} \)

from \( yin \) obtain \( i \) where \( y = (1 - p \ s) * wp b \ (M \ i) \ s \) by (auto)

with \( \text{dist y} \)

\( (\text{Sup} \ (\lambda i. (1 - p \ s) * wp b \ (M \ i) \ s)) < e \)

\( \text{by (simp)} \)

moreover from \( bbM \)

have \( (1 - p \ s) * wp b \ (M \ i) \ s \leq \text{Sup} \ (\lambda i. (1 - p \ s) * wp b \ (M \ i) \ s)) \)

\( \text{by (intro cSup-upper bdd-aboveI, auto)} \)

ultimately have \( \text{Sup} \ (\lambda i. (1 - p \ s) * wp b \ (M \ i) \ s)) \leq (1 - p \ s) * wp b \ (M \ i) \ s + e \)

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

thus \( \exists i. \text{Sup} \ (\lambda i. (1 - p \ s) * wp b \ (M \ i) \ s) \leq (1 - p \ s) * wp b \ (M \ i) \ s + e \) by (auto)

qed

from \( lima \) have \( \text{Sup} \ (\lambda i. p \ s * wp a \ (M \ i) \ s + (1 - p \ s) * wp b \ (M \ i) \ s) \)

\( \rightarrow \)

\( \text{Sup} \ (\lambda i. p \ s * wp a \ (M \ i) \ s) + \text{Sup} \ (\lambda i. (1 - p \ s) * wp b \ (M \ i) \ s) \)

\( \text{by (rule tendsto-add)} \)

moreover from \( add-mono [OF baM bbM] \)

have \( \lambda i. p \ s * wp a \ (M \ i) \ s + (1 - p \ s) * wp b \ (M \ i) \ s \leq \)

\( \text{Sup} \ (\lambda i. p \ s * wp a \ (M \ i) \ s + (1 - p \ s) * wp b \ (M \ i) \ s) \)

\( \text{by (intro cSup-upper bdd-aboveI, auto)} \)

ultimately have \( \text{Sup} \ (\lambda i. p \ s * wp a \ (M \ i) \ s) + \)

\( \text{Sup} \ (\lambda i. (1 - p \ s) * wp b \ (M \ i) \ s) \leq \)

\( \text{Sup} \ (\lambda i. p \ s * wp a \ (M \ i) \ s + (1 - p \ s) * wp b \ (M \ i) \ s) \)

\( \text{by (blast intro: LIMSEQ-le-const2)} \)

} \}

also \{

have range \( \lambda i. p \ s * wp a \ (M \ i) \ s + (1 - p \ s) * wp b \ (M \ i) \ s = \)

\( \{ f s | f \in \text{range} \ (\lambda x s. p \ s + wp a \ (M \ x) \ s + (1 - p \ s) * wp b \ (M \ x) \ s) \} \)

\( \text{by (auto)} \)

hence \( \text{Sup} \ (\lambda i. p \ s * wp a \ (M \ i) \ s + (1 - p \ s) * wp b \ (M \ i) \ s) = \)

\( \text{Sup-exp} \ (\lambda x s. p \ s + wp a \ (M \ x) \ s + (1 - p \ s) * wp b \ (M \ x) \ s) \) s

\( \text{by (simp add: Sup-exp-def cong del: SUP-cong-simp)} \)

\}

finally

have \( p \ s * wp a \ (\text{Sup-exp} \ (\text{range} M)) \ s + (1 - p \ s) * wp b \ (\text{Sup-exp} \ (\text{range} M)) \)
\[ s \leq \text{Sup-exp} (\text{range} (\lambda i. p \ s \star \text{wp} a (M i) s + (1 - p \ s) \star \text{wp} b (M i) s)) s. \]

moreover

\text{have} \text{Sup-exp} (\text{range} (\lambda i. p \ s \star \text{wp} a (M i) s + (1 - p \ s) \star \text{wp} b (M i) s)) s \leq p \ s \star \text{wp} a (\text{Sup-exp} (\text{range} M)) s + (1 - p \ s) \star \text{wp} b (\text{Sup-exp} (\text{range} M)) s

\text{proof} (\text{rule le-funD}[\text{OF Sup-exp-least}], \text{clarsimp}, \text{rule le-funI})

fix \ i :: \text{nat} \ 	ext{and} \ s :: s.

from \ bM \ 	ext{have} \ leSup: M i \vdash \text{Sup-exp} (\text{range} M)

by (\text{blast intro: Sup-exp-upper})

moreover from \ sM \ bM \ 	ext{have} \ sSup: \text{sound} (\text{Sup-exp} (\text{range} M))

by (\text{auto intro: Sup-exp-sound})

moreover note \ healthy-monoD[\text{OF ha}] sM

ultimately have \ wp a (M i) \vdash wp a (\text{Sup-exp} (\text{range} M)) by (\text{auto})

hence \ wp a (M i) s \leq wp a (\text{Sup-exp} (\text{range} M)) s \ \text{by (auto)}

moreover {

from \ leSup sSup \text{healthy-monoD[OF hb]} sM

have \ wp b (M i) \vdash wp b (\text{Sup-exp} (\text{range} M)) \ \text{by (auto)}

hence \ wp b (M i) s \leq wp b (\text{Sup-exp} (\text{range} M)) s \ \text{by (auto)}

}

moreover from \ wp \ 	ext{have} \ 0 \leq p \ s \ 0 \leq 1 - p \ s

by auto

ultimately

\text{show} p \ s \star \text{wp} a (M i) s + (1 - p \ s) \star \text{wp} b (M i) s \leq p \ s \star \text{wp} a (\text{Sup-exp} (\text{range} M)) s + (1 - p \ s) \star \text{wp} b (\text{Sup-exp} (\text{range} M)) s

\text{by (blast intro: add-mono mult-left-mono)}

from \ sSup \ ha \ hb \ 	ext{have} \ \text{sound} (wp a (\text{Sup-exp} (\text{range} M)))

\text{sound} (wp b (\text{Sup-exp} (\text{range} M)))

by (\text{auto})

hence \ \land s. \ 0 \leq wp a (\text{Sup-exp} (\text{range} M)) s \ \land s. \ 0 \leq wp b (\text{Sup-exp} (\text{range} M)) s

\text{by (auto)}

moreover from \ wp \ 	ext{have} \ \land s. \ 0 \leq p \ s \ \land s. \ 0 \leq 1 - p \ s

\text{by (auto)}

ultimately show \ \text{nngen} (\lambda c. p \ c \star \text{wp} a (\text{Sup-exp} (\text{range} M)) c + (1 - p \ c) \star \text{wp} b (\text{Sup-exp} (\text{range} M)) c)

\text{by (blast intro: add-nonneg-nonneg mult-nonneg-nonneg)}

qed

ultimately show \ p \ s \star \text{wp} a (\text{Sup-exp} (\text{range} M)) s + (1 - p \ s) \star \text{wp} b (\text{Sup-exp} (\text{range} M)) s = \text{Sup-exp} (\text{range} (\lambda x. p \ s \star \text{wp} a (M x) s + (1 - p \ s) \star \text{wp} b (M x)) s)

\text{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 oper-
4.3. CONTINUITY

lemmas.

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

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

lemma bd-cts-scale:
  fixes \( a : s \text{ trans} \)
  assumes \( ca : \text{bd-cts } a \)
  and \( ha : \text{healthy } a \)
  and \( nnc: 0 \leq c \)
  shows \( \text{bd-cts } (\lambda P \ s. \ c \ast a \ P \ s) \)
proof(intro bd-ctsI ext, simp add:o-def)
  fix \( M :: \text{nat} \) ⇒ \( 's \text{ expect} \) and \( d :: \text{real} \) and \( s : s' \)
  assume \( \text{chain: } \bigwedge i \ M \ i \vdash M (\text{Suc } i) \) and \( sM: \bigwedge i. \text{sound } (M \ i) \)
  and \( bM: \bigwedge i. \text{bounded-by } d (M \ i) \)

from \( sM \) have \( \bigwedge i. \text{nnc } (M \ i) \) by(auto)
with \( bM \) have \( \text{nnd: } 0 \leq d \) by(auto)

from \( sM \) \( bM \) have \( \text{sSup: } \text{sound } (\text{Sup-exp } \text{range } M) \)
  by(auto intro:Sup-exp-sound)
with \( \text{healthy-scaling}[OF ha] \) \( nnc \)
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(auto intro:scalingD)
also { 
  have \( \bigwedge s. \{ f \, s \mid f. \ f \in \text{range } M \} = \text{range } (\lambda i. \ M \ i \ s) \) by(auto)
  hence \( a (\lambda s. c \ast \text{Sup-exp } \text{range } (\lambda i. \ M \ i \ s)) \ s = \)
    \( a (\lambda s. \text{Sup } \{ c \ast x \mid x \in \text{range } \lambda i. \ M \ i \ s \}) \ s \)
  by(simp add:Sup-exp-def)
}
also { 
  from \( bM \) have \( \bigwedge x s. x \in \text{range } (\lambda i. \ M \ i \ s) \implies x \leq d \) by(auto)
  with \( nnc \) have \( a (\lambda s. \text{Sup } \{ c \ast x \mid x \in \text{range } \lambda i. \ M \ i \ s \}) \ s = \)
    \( a (\lambda s. \text{Sup } (\lambda i. c \ast M \ i \ s)) \ s \)
  by(subst cSup-mult, blast+)
}
also { 
  have \( X: \bigwedge s. \{ c \ast x \mid x \in \text{range } (\lambda i. \ M \ i \ s) \} = \text{range } (\lambda i. c \ast M \ i \ s) \) by(auto)
  have \( a (\lambda s. \text{Sup } \{ c \ast x \mid x \in \text{range } \lambda i. \ M \ i \ s \}) \ s = \)
    \( a (\lambda s. \text{Sup } (\lambda i. c \ast M \ i \ s)) \ s \) by(simp add:X)
}
also { 
  have \( \bigwedge s. \text{range } (\lambda i. c \ast M \ i \ s) = \{ f \, s \mid f. \ f \in \text{range } (\lambda i s. c \ast M \ i \ s) \} \) 
    by(auto)
  hence \( a (\lambda s. \text{Sup } (\lambda i. c \ast M \ i \ s)) = \text{Sup-exp } (\lambda i s. c \ast M \ i \ s) \) 
    by (simp add: Sup-exp-def cong del; SUP-cong-simp)
  hence \( a (\lambda s. \text{Sup } (\lambda i. c \ast M \ i \ s)) \ s = \)
    \( a (\text{Sup-exp } (\lambda i s. c \ast M \ i \ s)) \ s \) by(simp)
}
also { from le-funD[OF chain] nnc have \( \bigwedge i. (\lambda s. c \cdot M i s) \vdash (\lambda s. c \cdot M (Suc i) s) \)
by(auto intro:le-funI[OF mult-left-mono])
moreover from sM nnc have \( \bigwedge i. \) sound \( (\lambda s. c \cdot M i s) \)
by(auto intro:sound-intros)
moreover from bM nnc have \( \bigwedge i. \) bounded-by \( (c \cdot d) \) \( (\lambda s. c \cdot M i s) \)
by(auto intro:mult-left-mono)
ultimately have \( a \) \( (Sup-exp (range (\lambda i s. c \cdot M i s))) = Sup-exp (range (\lambda x. a (\lambda s. c \cdot M x s))) \)
by(rule bd-ctsD[OF ca])
hence \( a \) \( (Sup-exp (range (\lambda x. a (\lambda s. c \cdot M x s))) = Sup-exp (range (\lambda x. c \cdot a (M x s))) \)
by(auto)
}
also have \( Sup-exp (range (\lambda i s. c \cdot M i s))) \) \( s = Sup-exp (range (\lambda x. a (\lambda s. c \cdot M x s))) \)
by(simp add:o-def)
also {
from nnc sM have \( \bigwedge x. a \) \( (\lambda s. c \cdot M x s) = (\lambda s. c \cdot a (M x s)) \)
by(auto intro:scalingD[OF healthy-scalingD, OF ha, symmetric])
hence \( Sup-exp (range (\lambda x. a (\lambda s. c \cdot M x s))) \) \( s = Sup-exp (range (\lambda x. c \cdot a (M x s))) \)
by(simp)
}
finally show \( c \cdot a \) \( (Sup-exp (range M)) \) \( s = Sup-exp (range (\lambda x. c \cdot a (M x) s)) \)
by simp
qed

lemma cts-wp-SetPC-const:
fixes a::'a ⇒ 's prog
assumes ca: \( \bigwedge x. x \in (supp p) \implies bd-cts (wp (a x)) \)
and ha: \( \bigwedge x. x \in (supp p) \implies healthy (wp (a x)) \)
and up: unitary p
and sump: sum p \( (supp p) \leq 1 \)
and fsupp: finite \( (supp p) \)
shows bd-cts \( (wp (SetPC a (\lambda-. p))) \)
proof(cases supp p = \{ \}, simp add:supp-empty SetPC-def wp-def cts-bot)
assume nonsupp: supp p ≠ \{ \}
from fsupp have unitary p \implies sum p \( (supp p) \leq 1 \implies \\
(\forall x \in supp p. bd-cts (wp (a x))) \implies \\
(\forall x \in supp p. healthy (wp (a x))) \implies \\
bd-cts (wp (SetPC a (\lambda-. 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 \( p : 'a \Rightarrow \text{real} \)
assume \( fF : \text{finite} \ F \)
assume \( \text{insert} \ x \ F = \text{supp} \ p \)
hence \( p \text{step} : \text{supp} \ p = \text{insert} \ x \ F \) by(simp)
hence \( \text{xin} : x \in \text{supp} \ p \) by(auto)
assume \( up : \text{unitary} \ p \) and \( ca : \forall x \in \text{supp} \ p. \ \text{bd-cts} \ (\text{wp} \ (a \ x)) \)
and \( ha : \forall x \in \text{supp} \ p. \ \text{healthy} \ (\text{wp} \ (a \ x)) \)
and \( \text{sump} : \text{sum} \ p \ (\text{supp} \ p) \leq 1 \)
and \( \text{xn} : x \notin F \)
assume \( \text{IH} : \forall p. F \leq \text{supp} p \leq \Rightarrow \text{unitary} \ p \rightarrow \text{sum} \ p \ (\text{supp} \ p) \leq 1 \rightarrow \)
(\( \forall x \in \text{supp} \ p. \ \text{bd-cts} \ (\text{wp} \ (a \ x)) \)) \rightarrow \( \forall x \in \text{supp} \ p. \ \text{healthy} \ (\text{wp} \ (a \ x)) \)) \rightarrow \( \text{bd-cts} \ (\text{wp} \ (\text{SetPC} \ a \ (\lambda-. \ p))) \)
from \( fF \) \( p \text{step} \) have \( \text{fsupp} : \text{finite} \ (\text{supp} \ p) \) by(auto)
from \( \text{xin} \) have \( \text{nzp} : p \ x \neq 0 \) by(simp add:supp-def)

have \( \text{xy-ic-sum} : \)
\( \forall y. y \in \text{supp} \ p \Rightarrow y \neq x \Rightarrow p \ x + p \ y \leq \text{sum} \ p \ (\text{supp} \ p) \)

proof -
fix \( y \) assume \( \text{yin} : y \in \text{supp} \ p \) and \( \text{yne} : y \neq x \)
from \( up \) have \( 0 \leq \text{sum} \ p \ (\text{supp} \ p - \{x, y\}) \)
by(auto intro:sum-nonneg)
hence \( p \ x + p \ y \leq p \ x + p \ y + \text{sum} \ p \ (\text{supp} \ p - \{x, y\}) \)
by(auto)
also {
from \( \text{yin} \ \text{yne} \ \text{fsupp} \)
have \( p \ y + \text{sum} \ p \ (\text{supp} \ p - \{x, y\}) = \text{sum} \ p \ (\text{supp} \ p - \{x\}) \)
by(subst \text{sum.insert}[symmetric], (blast intro:sum.cong)+)
moreover
from \( \text{xin} \) \( \text{fsupp} \)
have \( p \ x + \text{sum} \ p \ (\text{supp} \ p - \{x\}) = \text{sum} \ p \ (\text{supp} \ p) \)
by(subst \text{sum.insert}[symmetric], (blast intro:sum.cong)+)
ultimately
have \( p \ x + p \ y + \text{sum} \ p \ (\text{supp} \ p - \{x, y\}) = \text{sum} \ p \ (\text{supp} \ p) \) by(simp)
}
finally show \( p \ x + p \ y \leq \text{sum} \ p \ (\text{supp} \ p) \).

qed

have \( \text{n1p} : \forall y. y \in \text{supp} \ p \Rightarrow y \neq x \Rightarrow p \ x \neq 1 \)

proof(rule \text{contr}, \text{simp})
assume \( \text{px1} : p \ x = 1 \)
fix \( y \) assume \( \text{yin} : y \in \text{supp} \ p \) and \( \text{yne} : y \neq x \)
from \( up \) have \( 0 \leq p \ y \) by(auto)
with \( \text{yin} \) have \( 0 < p \ y \) by(auto simp:supp-def)
hence \( 0 + p \ x < p \ y + p \ x \) by(rule add-strict-right-mono)
with \( \text{px1} \) have \( 1 < p \ x + p \ y \) by(simp)
also from \( yin \) \( yne \) have \( p \ x + p \ y \leq \sum p \ (supp \ p) \)
by \((\text{rule } xy-le-sum)\)

finally show \( \text{False} \) using \( sump \) by \((\text{simp})\)
qed

show \( bd-cts \ (wp \ (\text{SetPC} \ a \ (\lambda- \ p))) \)
proof \((\text{cases } F = \{\})\)
case \( \text{True} \)
with \( pstep \)
have \( \text{supp} \ p = \{x\} \)
by \((\text{simp})\)

hence \( wp \ (\text{SetPC} \ a \ (\lambda- \ p)) = (\lambda P s. \ p \ x * wp \ (a x) \ P \ s) \)
by \((\text{simp add: SetPC-sgl wp-def})\)

moreover \{
from \( up \ ca \ ha \ xin \)
have \( bd-cts \ (wp \ (a x)) \)
by \((\text{auto})\)

hence \( bd-cts \ (\lambda P \ s. \ p \ x * wp \ (a x) \ P \ s) \)
by \((\text{rule bd-cts-scale})\)
\}

ultimately show \( ?thesis \)
by \((\text{simp})\)

next
assume \( \neg F: F \neq \{\} \)
then obtain \( y \) where \( yinF: y \in F \)
by \((\text{auto})\)

with \( xni \)
have \( yne: y \neq x \)
by \((\text{auto})\)

from \( yinF \)
\( pstep \)
have \( yin: y \in \text{supp} \ p \)
by \((\text{auto})\)

from \( \text{supp-dist-remove}[of \ p \ x, \ OF \ nzp \ n1p, \ OF \ yin \ yne] \)
have \( \text{supp-sub: supp} \ (\text{dist-remove} \ p \ x) \subseteq \text{supp} \ p \)
by \((\text{auto})\)

from \( xin \ ca \)
have \( cax: bd-cts \ (wp \ (a x)) \)
by \((\text{auto})\)

from \( xin \ ha \)
have \( hax: \text{healthy} \ (wp \ (a x)) \)
by \((\text{auto})\)

from \( \text{supp-sub} \ ha \)
have \( hra: \forall x \in \text{supp} \ (\text{dist-remove} \ p \ x). \ \text{healthy} \ (wp \ (a x)) \)
by \((\text{auto})\)

from \( \text{supp-sub} \ ca \)
have \( cra: \forall x \in \text{supp} \ (\text{dist-remove} \ p \ x). \ bd-cts \ (wp \ (a x)) \)
by \((\text{auto})\)

from \( \text{supp-dist-remove}[of \ p \ x, \ OF \ nzp \ n1p, \ OF \ yin \ yne] \)
\( pstep \ xni \)
have \( Fsupp: F = \text{supp} \ (\text{dist-remove} \ p \ x) \)
by \((\text{simp})\)

have \( udp: \text{unitary} \ (\text{dist-remove} \ p \ x) \)
proof \((\text{intro unitaryI2 nnegI bounded-byI})\)
fix \( y \)
show \( 0 \leq \text{dist-remove} \ p \ x \ y \)
proof \((\text{cases } y=x, \ simp-all add:dist-remove-def})
from \( up \)
have \( 0 \leq p \ y \ 0 \leq 1 - p \ x \)
by \((\text{auto})\)

thus \( 0 \leq p \ y / (1 - p \ x) \)
by \((\text{rule divide-nonneg-nonneg})\)
qed

show \( \text{dist-remove} \ p \ x \ y \leq 1 \)
proof\((cases\ \text{y=x, simp-all add:dist-remove-def, cases y\in supp\ p, simp-all add:nsupp-zero})\)

assume \(y\neq x\) and \(y\in supp\ p\)

hence \(p\ x + p\ y \leq \text{sum}\ (supp\ p)\)

by\((auto\ intro:xy-le-sum)\)

also note \(s\ p\)

finally have \(p\ y \leq 1 - p\ x\) by\((auto)\)

moreover from \(up\) have \(p\ x \leq 1\) by\((auto)\)

moreover from \(y\neq y\) have \(p\ x \neq 1\) by\((\text{rule n1p})\)

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 \(yin\) have \(pxn1: p\ x \neq 1\) by\((\text{rule n1p})\)

from \(pxn0\) \(pxn1\) have \(\text{sum}\ (\text{dist-remove}\ p\ x)\ (\text{supp}\ (\text{dist-remove}\ p\ x)) = \text{sum}\ (\text{dist-remove}\ p\ x)\ (\text{supp}\ p - \{x\})\)

by\((\text{simp add:supp-dist-remove})\)

also have \(\ldots = (\sum y\in supp\ p - \{x\}.\ p\ y / (1 - p\ x))\)

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

also have \(\ldots = (\sum y\in supp\ p - \{x\}.\ p\ y) / (1 - p\ x)\)

by\((\text{simp add:sum-divide-distrib})\)

also \{

from \(xin\) have \(insert\ x\ (supp\ p) = supp\ p\) by\((auto)\)

with \(fsupp\) have \(p\ x + (\sum y\in supp\ p - \{x\}.\ p\ y) = \text{sum}\ p\ (supp\ p)\)

by\((\text{simp add:sum.insert[symmetric]})\)

also note \(s\ p\)

finally have \(\text{sum}\ p\ (supp\ p - \{x\}) \leq 1 - p\ x\) 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 \(\text{sum}\ p\ (supp\ p - \{x\}) / (1 - p\ x) \leq 1\)

by\((auto)\)

\}

finally have \(s\ p: \text{sum}\ (\text{dist-remove}\ p\ x)\ (\text{supp}\ (\text{dist-remove}\ p\ x)) \leq 1\).

from \(Fsupp\) \(udp\) \(sdp\) \(hra\) \(cra\) \(IH\)

have \(cts-dr: bd-cts\ (wp\ (\text{SetPC}\ a\ (\lambda.\ \text{dist-remove}\ p\ x)))\)

by\((auto)\)

from \(up\) have \(upx: \text{unitary}\ (\lambda.\ p\ x)\) by\((auto)\)

from \(pxn0\) \(pxn1\) \(fsupp\) \(hra\) show \(?\)thesis

by\((\text{simp add:SetPC-remove, blast intro:cts-wp-PC caz cts-dr hax healthy-intros unitary-sound[OF udp] sdp upx})\)
4.3. CONTINUITY

qed
qed
with assms show ?thesis by (auto)
qed

lemma cts-wp-SetPC:
  fixes a::'a ⇒ 's prog
  assumes ca: "∀ x s. x ∈ (supp (p s)) ⟹ bd-cts (wp (a x))
  and ha: "∀ s. unitary (p s)
  and sump: "∀ s. sum (p s) (supp (p s)) ≤ 1
  and fsupp: "∀ s. finite (supp (p s))
  shows bd-cts (wp (SetPC a p))
proof
  from assms have bd-cts (wp (Bind p (λ p. SetPC a (λ x. p))))
  by (iprover intro!: cts-wp-Bind cts-wp-SetPC-const)
  thus ?thesis by (simp add: SetPC-Bind [symmetric])
qed

lemma wp-SetDC-Bind:
  SetDC a S = Bind S (λ S. SetDC a (λ x. S))
by (intro ext, simp add: SetDC-def Bind-def)

lemma SetDC-finite-insert:
  assumes fS: "finite S
  and neS: "S ≠ {}
  shows SetDC a (λ x. insert x S) = a x ∩ SetDC a (λ x. S)
proof (intro ext, simp add: SetDC-def DC-def cong del: INF-cong-simp)
  fix ab P s
  from fS have A: "finite (insert (a x ab P s) ((λ x. a x ab P s) ' S))
  and B: "finite (((λ x. a x ab P s) ' S))
  by (auto)
  from neS have C: "insert (a x ab P s) ((λ x. a x ab P s) ' S) ≠ {}
  and D: "λ x. a x ab P s) ' S ≠ {}
  by (auto)
  from A C have Inf (insert (a x ab P s) ((λ x. a x ab P s) ' S)) =
    Min (insert (a x ab P s) ((λ x. a x ab P s) ' S))
  by (auto intro: cInf-eq-Min)
  also from B D have ... = min (a x ab P s) (Min ((λ x. a x ab P s) ' S))
  by (auto intro: Min-insert)
  also from B D have ... = min (a x ab P s) (Inf ((λ x. a x ab P s) ' S))
  by (simp add: cInf-eq-Min)
  finally show (INF x∈insert x S. a x ab P s) =
    min (a x ab P s) (INF x∈S. a x ab P s)
  by (simp cong del: INF-cong-simp)
qed

lemma SetDC-singleton:
  SetDC a (λ x. {x}) = a x
by (simp add: SetDC-def cong del: INF-cong-simp)
lemma cts-wp-SetDC-const:
fixes a::'a ⇒ 's prog
assumes ca: \( \forall x. x \in S \Rightarrow bd-cts (wp (a x)) \)
and ha: \( \forall x. x \in S \Rightarrow healthy (wp (a x)) \)
and fS: finite S
and neS: S ≠ {}
shows \( bd-cts (wp (SetDC a (λ-. S))) \)
proof –
  have finite S \( \Rightarrow S \neq {} \) \( \Rightarrow \)
  \( (\forall x \in S. bd-cts (wp (a x))) \) \( \Rightarrow \)
  \( (\forall x \in S. healthy (wp (a x))) \) \( \Rightarrow \)
  \( bd-cts (wp (SetDC a (λ-. S))) \)
proof (induct S rule: finite-induct, simp, clarsimp)
fix x::'a and F::'a set
assume fF: finite F
and IH: F ≠ {} \( \Rightarrow \)
  \( (\forall x \in F. healthy (wp (a x))) \) \( \Rightarrow \)
show bd-cts (wp (SetDC a (λ-. insert x F)))
proof (cases F = {}, simp add: SetDC-finite-insert)
  assume F ≠ {}
  with fF cax haF IH show bd-cts (wp (SetDC a (λ-. insert x F)))
by (auto intro!: cts-wp-SetDC-const !)
thus ?thesis by (simp add: wp-SetDC-Bind [symmetric])
qed
qed

lemma cts-wp-repeat:
bd-cts (wp a) \( \Rightarrow \) healthy (wp a) \( \Rightarrow \) bd-cts (wp (repeat n a))
by (induct n, auto intro!: cts-wp-Skip cts-wp-Seq healthy-intros)

lemma cts-wp-Embed:
bd-cts t \( \Rightarrow \) bd-cts (wp (Embed t))
4.3. CONTINUITY

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** \( \text{body} ::'s \text{ prog} \)

**assumes** \( \text{hb} :: \text{ healthy} \ (\text{wp body}) \)

and \( \text{cb} :: \text{ bd-cts} \ (\text{wp body}) \)

**shows** \( \text{bd-cts-tr} \ (\lambda x. \text{ wp} \ (\text{body} ;; \text{Embed} \ x \ « G \times Skip))) \ (\text{is bd-cts-tr} \ ?F) \)

**proof**(rule bd-cts-trI, rule le-trans-antisym)

fix \( M :: \text{nat} \Rightarrow \text{'s \ trans} \) and \( b :: \text{real} \)

assume \( \text{chain} :: \exists i. \text{ le-trans} \ (M i) \ (M (\text{Suc} \ i)) \)

and \( \text{fM} :: \exists i. \text{ feasible} \ (M i) \)

show \( \text{fw} :: \text{ le-trans} \ (\text{Sup-trans} \ (\text{range} \ ?F \circ M)) \)

proof(rule le-transI[OF Sup-trans-least2], clarsimp)

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

assume \( \text{sP} :: \text{sound} \ P \)

assume \( \text{nQ} :: \text{nneg} \ Q \) and \( \text{bP} :: \text{bounded-by} \ (\text{bound-of} \ P) \ Q \)

hence \( \text{sQ} :: \text{sound} \ Q \) by (auto)

from \( \text{fM} \) have \( \text{fSup} :: \text{feasible} \ (\text{Sup-trans} \ (\text{range} \ M)) \)

by (auto intro:feasible-Sup-trans)

from \( \text{sQ} \ \text{fM} \) have \( \text{M t Q \vdash} \ (\text{Sup-trans} \ (\text{range} \ M)) \)

by (auto intro:Sup-trans-upper2)

moreover from \( \text{sQ} \ \text{fM} \ \text{fSup} \)

have \( \text{sMtP} :: \text{sound} \ (\text{M t Q}) \).

ultimately have \( \text{wp body} \ (M t Q) \vdash \ (\text{wp body} \ (\text{Sup-trans} \ (\text{range} \ M)) \ Q) \)

using healthy-monoD[OF \text{hb}] by (auto)

hence \( \exists s. \ (\text{wp body} \ (M t Q) \ s \leq \wp body \ (\text{Sup-trans} \ (\text{range} \ M)) \ s) \)

by (rule le-funD)

thus \( \exists F \ (\text{M t Q}) \vdash \exists F \ (\text{Sup-trans} \ (\text{range} \ M)) \)

by (intro le-funI, simp add:wp-eval mult-left mono)

show \( \text{nneg} \ (\text{wp} \ (\text{body} ;; \text{Embed} \ (\text{Sup-trans} \ (\text{range} \ M)) \ « G \times Skip)) \)

proof(rule nnegI, simp add:wp-eval)

fix \( s ::'s \)

from \( \text{fSup} \ \text{sQ} \) have \( \text{sound} \ (\text{Sup-trans} \ (\text{range} \ M) \ Q) \) by (auto)

with \( \text{hb} \) have \( \text{sound} \ (\text{wp body} \ (\text{Sup-trans} \ (\text{range} \ M) \ Q)) \) by (auto)

hence \( 0 \leq \wp body \ (\text{Sup-trans} \ (\text{range} \ M) \ Q) \ s \) by (auto)

moreover from \( \text{sQ} \) have \( 0 \leq Q \ s \) by (auto)

ultimately show \( 0 \leq \ « G \ s \ast \wp body \ (\text{Sup-trans} \ (\text{range} \ M) \ Q) \ s \ast (1 - « G \ s) \ast Q \ s \)

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

qed

next

fix \( P ::'s \text{ expect} \) assume \( \text{sP} :: \text{sound} \ P \)
thus \text{nneg} \ P \text{ bounded-by (bound-of} \ P) \ \text{by(auto)}

\text{show} \ \forall \ u \in \text{range} \ (\langle x. \ \text{wp (body :: Embed} \ x = G \odot \text{Skip} \rangle \odot M). 
\forall R. \ \text{nneg} \ R \land \text{bounded-by (bound-of} \ P) \ R \hookrightarrow 
\text{nneg} \ (u \ R) \land \text{bounded-by (bound-of} \ P) \ (u \ R)

\text{proof (clarsimp, intro conjI nnegI bounded-byI, simp-all add:wp-eval)}
\text{fix} \ u :: \text{nat} \ \text{and} \ R :: 's \ \text{expect} \ \text{and} \ s :: 's
\text{assume} \ \text{nneg} \ R \ \text{and} \ bR \ : \ \text{bounded-by (bound-of} \ P) \ R
\text{hence} \ sr : \ \text{sound} \ R \ \text{by(auto)}
\text{with} \ fM \ \text{have} \ \text{sndR} \ : \ \text{sound} \ (M \ u \ R) \ \text{by(auto)}
\text{with} \ hb \ \text{have} \ \text{sound} (\text{wp body} (M \ u \ R)) \ \text{by(auto)}
\text{hence} \ 0 \leq \ \text{wp body} (M \ u \ R) \ s \ \text{by(auto)}
\text{moreover from} \ nR \ \text{have} \ 0 \leq R \ s \ \text{by(auto)}
\text{ultimately show} \ 0 \leq «G» s \ast \ \text{wp body} (M \ u \ R) \ s + (1 - «G» s) \ast R \ s
\text{by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)}
from \ sR \ bR \ fM \ \text{have} \ \text{bounded-by (bound-of} \ P) \ (M \ u \ R) \ \text{by(auto)}
\text{with} \ sMuR \ hb \ \text{have} \ \text{bounded-by (bound-of} \ P) \ (\text{wp body} (M \ u \ R)) \ \text{by(auto)}
\text{hence} \ \text{wp body} (M \ u \ R) \ s \leq \ \text{bound-of} \ P \ \text{by(auto)}
\text{moreover from} \ bR \ \text{have} \ R \ s \leq \ \text{bound-of} \ P \ \text{by(auto)}
\text{ultimately have} \ «G» s \ast \ \text{wp body} (M \ u \ R) \ s + (1 - «G» s) \ast R \ s \leq
\text{ultimately show} \ «G» s \ast \ \text{wp body} (M \ u \ R) \ s + (1 - «G» s) \ast \ \text{bound-of} \ P
\text{by(auto intro:add-mono mult-left-mono)}
\text{also have} \ ... = \ \text{bound-of} \ P \ \text{by(simp add:algebra-simps)}
\text{finally show} \ «G» s \ast \ \text{wp body} (M \ u \ R) \ s + (1 - «G» s) \ast R \ s \leq \ \text{bound-of} \ P.
\text{qed}
\text{qed}

\text{show} \ \text{le-trans} (\langle F \ (\text{Sup-trans (range} \ M)) \rangle) \ (\text{Sup-trans (range} \ \langle F \circ M \rangle))
\text{proof (rule le-transI, rule le-funI, simp add:wp-eval cong del: image-cong-simp)}
\text{fix} \ P :: 's \ \text{expect} \ \text{and} \ s :: 's
\text{assume} \ \text{sp} : \ \text{sound} \ P
\text{have} \ \{ t \ P \ | \ t \in \text{range} \ M \} = \text{range} \ (\lambda i. \ M \ i \ P)
\text{by(blast)}
\text{hence} \ \text{wp body} (\text{Sup-trans (range} \ M) \ P) \ s = \ \text{wp body} (\text{Sup-exp (range} \ (\lambda i. \ M \ i \ P))) \ s
\text{by(simp add:Sup-trans-def)}
\text{also}
\text{from} \ \text{sp fM} \ \text{have} \ \langle i. \ \text{sound} \ (M \ i \ P) \ \text{by(auto)}
\text{moreover from} \ \text{sp chain} \ \text{have} \ \langle i. \ M \ i \ P \vdash M \ (\text{Suc} \ i) \ P \ \text{by(auto)}
\text{moreover}
\text{from} \ \text{sp fM} \ \text{have} \ \langle i. \ \text{bounded-by (bound-of} \ P) \ M \ i \ P) \ \text{by(auto)}
\text{with} \ \text{sp fM} \ \text{have} \ \langle i. \ \text{bounded-by (bound-of} \ P) \ M \ i \ P) \ \text{by(auto)}
\text{ultimately have} \ \text{wp body} (\text{Sup-exp (range} \ (\lambda i. \ M \ i \ P))) \ s =
\text{Sup-exp (range} \ (\lambda i. \ \text{wp body} (M \ i \ P))) \ s
\text{by(subst bd-ctsD[OF cb, auto simp:o-def)}
\text{also have} \ \text{Sup-exp (range} \ (\lambda i. \ \text{wp body} (M \ i \ P))) \ s =
4.3. CONTINUITY

\[
\text{by(simp add:Sup-exec-def)}
\]

finally have \(\langle \! \langle G \rangle \! \rangle \ast \\text{wp body} (\text{Sup-trans (range M) P}) \ast \lambda \, s \in \text{range} (\lambda i. \text{wp body} (M \ast P)) \rangle \ast (1 - \langle \! \langle \! \langle G \rangle \! \rangle \! \rangle \ast P \ast s) \ast P \ast s \ast \langle \! \langle G \rangle \! \rangle \ast \text{Sup} \{f \ast [f. f \in \text{range} (\lambda i. \text{wp body} (M \ast P)) \} \ast (1 - \langle \! \langle \! \langle G \rangle \! \rangle \! \rangle \ast P \ast s)
\]

by(simp)

also { 
from \(sP \ast M \) have \(\bigwedge i. \text{sound} (M \ast i \ast P) \) by(auto)
moreover from \(sP \ast FM \) have \(\bigwedge i. \text{bounded-by (bound-of \ast P) (M \ast i \ast P)} \) by(auto)
ultimately have \(\bigwedge i. \text{bounded-by (bound-of \ast P) (wp body (M \ast i \ast P))} \) using hb
by(auto)

hence bound: \(\bigwedge i. \text{wp body} (M \ast i \ast P) \ast s \leq \text{bound-of} \ast P \) by(auto)
moreover have \(\{ \langle \! \langle G \rangle \! \rangle \ast s \ast f \ast [f. f \in \text{range} (\lambda i. \text{wp body} (M \ast i \ast P)) \} \} = \{ \langle \! \langle G \rangle \! \rangle \ast s \ast f \ast [f. f \in \text{range} (\lambda i. \text{wp body} (M \ast i \ast P)) \} \}
\)

by(blast)

ultimately have \(\langle \! \langle G \rangle \! \rangle \ast \text{Sup} \{f \ast [f. f \in \text{range} (\lambda i. \text{wp body} (M \ast i \ast P)) \} = \text{Sup} \{\langle \! \langle G \rangle \! \rangle \ast s \ast f \ast [f. f \in \text{range} (\lambda i. \text{wp body} (M \ast i \ast P)) \} \}\)
by(subst cSup-mult, auto)
moreover \{ 
have \(\{x + (1 - \langle \! \langle G \rangle \! \rangle \ast P \ast s \ast x. x \in \{\langle \! \langle G \rangle \! \rangle \ast s \ast f \ast [f. f \in \text{range} (\lambda i. \text{wp body} (M \ast i \ast P)) \} \}\) = \(\{\langle \! \langle G \rangle \! \rangle \ast s \ast f \ast [f. f \in \text{range} (\lambda i. \text{wp body} (M \ast i \ast P)) \} \})
by(blast)
moreover from bound \(sP \ast M \) have \(\bigwedge i. \langle \! \langle G \rangle \! \rangle \ast s \ast \text{wp body} (M \ast i \ast P) \ast s \leq \text{bound-of} \ast P \)
by(cases G s, auto)
ultimately have \(\text{Sup} \{\langle \! \langle G \rangle \! \rangle \ast s \ast f \ast [f. f \in \text{range} (\lambda i. \text{wp body} (M \ast i \ast P)) \} + (1 - \langle \! \langle G \rangle \! \rangle \ast P \ast s \ast P \ast s \ast \langle \! \langle G \rangle \! \rangle \ast \text{Sup} \{f \ast [f. f \in \text{range} (\lambda i. \text{wp body} (M \ast i \ast P)) \} \}
\}
by(subst cSup-add, auto)
\}
ultimately have \(\langle \! \langle G \rangle \! \rangle \ast \text{Sup} \{f \ast [f. f \in \text{range} (\lambda i. \text{wp body} (M \ast i \ast P)) \} + (1 - \langle \! \langle G \rangle \! \rangle \ast P \ast s \ast P \ast s \ast \langle \! \langle G \rangle \! \rangle \ast \text{Sup} \{f \ast [f. f \in \text{range} (\lambda i. \text{wp body} (M \ast i \ast P)) \} \})
\)
by(simp)

also { 
have \(\bigwedge i. \langle \! \langle G \rangle \! \rangle \ast s \ast \text{wp body} (M \ast i \ast P) \ast s + (1 - \langle \! \langle G \rangle \! \rangle \ast P \ast s =\)
\((\lambda x. \text{wp (body ; Embed x \ast G \ast Skip)}) \circ M \) \ast i \ast P \ast s
by(simp add:wp-eval)
also have \(\bigwedge i. \ast t \leq \text{Sup} \{f \ast [f. f \in \{t \ast P \ast t. t \in \text{range} ((\lambda x. \text{wp (body ; Embed x \ast G \ast Skip)}) \circ M \) \ast i \ast P \ast s
\}
\)
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 \(\text{mono-trans} (\lambda Q s. \langle G \rangle s \star \text{wp body} Q s + \langle N \rangle G s \star P s)\)

proof (intro mono-transI le-funI, simp)

fix \(Q R::s\) expect and \(s::s\)

assume \(\text{sound} Q\) and \(\text{sound} R\) and \(\text{le:} Q \vdash R\)

hence wp body \(Q \vdash \text{wp body} R\)

by (rule mono-transD[OF \(\text{healthy-monoD}\), OF \(hb\)])

thus \(\langle G \rangle s \star \text{wp body} Q s \leq \langle G \rangle s \star \text{wp body} R s\)

by (auto dest: le-funD intro: mult-left-mono)

qed

We can therefore apply the standard fixed-point lemmas to unfold it:

lemma \(\text{lfp-wp-loop-unfold}\):

fixes \(body::s\) prog

assumes \(\text{hb: healthy} (\text{wp body})\)

and \(sP::s\) sound \(P\)

shows \(\text{lfp-exp} (\lambda Q s. \langle G \rangle s \star \text{wp body} Q s + \langle N \rangle G s \star P s) = (\lambda s. \langle G \rangle s \star \text{wp body} (\text{lfp-exp} (\lambda Q s. \langle G \rangle s \star \text{wp body} Q s + \langle N \rangle G s \star P s)) s + \langle N \rangle G s \star P s)\)

proof (rule lfp-exp-unfold)

from assms show mono-trans \((\lambda Q s. \langle G \rangle s \star \text{wp body} Q s + \langle N \rangle G s \star P s)\)

by (blast intro: wp-loop-step-mono-trans)

from assms show \(\lambda s. \langle G \rangle s \star \text{wp body} (\lambda s. \text{bound-of} P) s + \langle N \rangle G s \star P s \vdash \lambda s. \text{bound-of} P\)

by (blast intro: lfp-loop-fp)

from \(sP\) show \(\text{sound} (\lambda s. \text{bound-of} P)\)

by (auto)

fix \(Q::s\) expect

assume \(\text{sound} Q\)

with assms show \(\text{sound} (\lambda s. \langle G \rangle s \star \text{wp body} Q s + \langle N \rangle G s \star P s)\)

by (intro wp-loop-step-sound[unfolded wp-eval, simplified, folded negate-embed], auto)

qed

lemma \(\text{wp-loop-step-unitary}\):

fixes \(body::s\) prog

assumes \(\text{hb: healthy} (\text{wp body})\)

and \(uP::s\) unitary \(P\) and \(uQ::s\) unitary \(Q\)

shows \(\text{unitary} (\lambda s. \langle G \rangle s \star \text{wp body} Q s + \langle N \rangle G s \star P s)\)

proof (intro unitaryI2 nnegI bounded-byI)

fix \(s::s\)

from \(uQ \text{ hb have} uuQ::s\) unitary \((\text{wp body} Q)\) by (auto)

with \(uP\) have \(0 \leq \text{wp body} Q s \leq P s\) by (auto)

thus \(0 \leq \langle G \rangle s \star \text{wp body} Q s + \langle N \rangle G s \star P s\)

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

from \(uP\) \(uuQ\) have \(\text{wp body} Q s \leq 1 P s \leq 1\) by (auto)

hence \(\langle G \rangle s \star \text{wp body} Q s + \langle N \rangle G s \star P s \leq \langle G \rangle s \star 1 + \langle N \rangle G s \star 1\)
by (blast intro: add-mono mult-left-mono)
also have ... = 1 by (simp add: negate-embed)
finally show "G" s * wp body Q s + "N" G" s * P s ≤ 1 .
qed

lemma lfp-loop-unitary:
fixes body :: 's prog
assumes hb: healthy (wp body)
and uP: unitary P
shows unitary (lfp-exp (λQ s. "G" s * wp body Q s + "N" G" s * P s))
using assms by (blast intro: lfp-exp-unitary wp-loop-step-unitary)

From the lattice structure on transformers, we establish a transfinite induction principle for loops. We use this to show a number of properties, particularly subdistributivity, for loops. This proof follows the pattern of lemma lfp_ordinal_induct in HOL/Inductive.

lemma loop-induct:
fixes body :: 's prog
assumes hwp: healthy (wp body)
and hwlp: nearly-healthy (wlp body)
— The body must be healthy, both in strict and liberal semantics.
and Limit: \( \forall S. \big[ \forall x \in S. P (\text{fst } x) (\text{snd } x); \forall x \in S. \forall Q. \text{unitary } Q \rightarrow \text{unitary } (\text{snd } x \; Q) \big] \rightarrow \text{P} \ (\text{Sup-trans (fst ' S)}) (\text{Inf-utrans (snd ' S)}) \)
— The property holds at limit points.
and IH: \( \forall t \; u \; . \big[ \text{P } t \; u; \text{feasi}\text{ble } t; \forall Q. \text{unitary } Q \rightarrow \text{unitary } (u \; Q) \big] \rightarrow \text{P} \ (\text{wp } (\text{body } ; \text{Embed } t \; "G" \oplus \text{Skip})) (\text{wlp } (\text{body } ; \text{Embed } u \; "G" \oplus \text{Skip})) \)
— The inductive step. The property is preserved by a single loop iteration.
and P-equiv: \( \forall t \; t' \; u \; u' . \big[ \text{P } t \; u; \text{equiv-trans } t \; t'; \text{equiv-utrans } u \; u' \big] \rightarrow \text{P } t' \; u' \)
— The property must be preserved by equivalence
shows P (wp (do G → body od)) (wlp (do G → body od))
— The property can refer to both interpretations simultaneously. The unifier will happily apply the rule to just one or the other, however.

proof (simp add: wp-eval)
let ?X t = wp (body ;; Embed t "G" ⊕ Skip)
let ?Y t = wlp (body ;; Embed t "G" ⊕ Skip)

let ?M = \{ x. P (\text{fst } x) (\text{snd } x) \} ∧ 
\text{feasible } (\text{fst } x) \land 
(\forall Q. \text{unitary } Q \rightarrow \text{unitary } (\text{snd } x \; Q)) \land 
\text{le-trans } (\text{fst } x) (\text{lfp-trans } ?X) \land 
\text{le-utrans } (\text{gfp-trans } ?Y) (\text{snd } x) \}

have fSup: feasible (Sup-trans (fst ' ?M))
proof (intro feasibleI bounded-byI2 nnegI2)
fix Q::'s expect and b::real
assume nQ: nneg Q and bQ: bounded-by b Q
4.4. CONTINUITY AND INDUCTION FOR LOOPS

show $\text{Sup-trans (fst ' {?M}) Q} \vdash \lambda s. b$

unfolding $\text{Sup-trans-def}$

using $nQ bQ$ by (auto intro; $\text{Sup-expQ-least}$)

show $\lambda s. 0 \vdash \text{Sup-trans (fst ' {?M}) Q}$

proof (cases)

assume empty: $?M = \{\}$

show $?thesis$ by (simp add: $\text{Sup-trans-def Sup-expQ-def empty}$)

next

assume $ne: \ ?M \neq \{\}$

then obtain $x$ where $xin: x \in ?M$ by auto

hence $ffx$: feasible (fst $x$) by (simp)

with $nQ bQ$ have $\lambda s. 0 \vdash \text{fst x Q}$ by (auto)

also from $xin$ have $\text{fst x Q} \vdash \text{Sup-trans (fst ' {?M}) Q}$

apply (intro $\text{Sup-trans-upper2 [OF imageI - nQ bQ], assumption}$)

apply (clarsimp, blast intro: $\text{sound-nneg [OF feasible-sound feasible-boundedD]}$

done

finally show $\lambda s. 0 \vdash \text{Sup-trans (fst ' {?M}) Q}$.

qed

qed

have $\text{uInf: } \bigwedge P. \text{unitary P} \Rightarrow \text{unitary (Inf-utrans (snd ' {?M}) P)}$

proof (cases $?M = \{\}$)

fix $P$

assume empty: $?M = \{\}$

show $?thesis$ $P$ by (simp only: empty, simp add: $\text{Inf-utrans-def}$)

next

fix $P$: 's expect

assume $uP$: unitary $P$

and $ne: \ ?M \neq \{\}$

show $?thesis$ $P$

proof (intro unitaryI2 nnegI2 bounded-byI2)

from $ne$ obtain $x$ where $xin: x \in ?M$ by auto

hence $sxin$: snd $x \in \text{snd ' {?M}$ by (simp)

hence $\text{le-utrans (Inf-utrans (snd ' {?M)}) (snd x)}$

by (intro $\text{Inf-utrans-lower, auto}$)

with $uP$

have $\text{Inf-utrans (snd ' {?M}) P} \vdash \text{snd x P}$ by (auto)

also {

from $xin$ $uP$ have unitary (snd $x$ $P$) by (simp)

hence snd $x$ $P \vdash \lambda s. 1$ by (auto)
}

finally show $\text{Inf-utrans (snd ' {?M}) P} \vdash \lambda s. 1$.

have $\lambda s. 0 \vdash \text{Inf-trans (snd ' {?M}) P}$

unfolding $\text{Inf-trans-def}$

proof (rule $\text{Inf-exp-greatest}$)

from $sxin$ show $\{ t P | t \in \text{snd ' {?M}\} \neq \{\}$ by (auto)

show $\forall P \in \{ t P | t \in \text{snd ' {?M}\}. \lambda s. 0 \vdash P$

proof (clarsimp)
CHAPTER 4. THE PGCL LANGUAGE

fix t::'s trans
assume \forall Q. \text{unitary } Q \rightarrow \text{unitary } (t Q)
with uP have \text{unitary } (t P) by(auto)
thus \lambda s. 0 \vdash t P by(auto)
qed
qed
also {
  from ne have X: \text{(snd \ ?M = \{\}) = False} by(simp)
  have Inf-trans (snd \ ?M) P = Inf-utrans (snd \ ?M) P
    unfolding Inf-utrans-def by(subst X, simp)
}
finally show \lambda s. 0 \vdash \text{Inf-utrans } (\text{snd \ ?M}) P .
qed
qed

have wp-loop-mono: \forall t u. [le-trans t u; \forall P. \text{sound } P \implies \text{sound } (t P);
  \forall P. \text{sound } P \implies \text{sound } (u P)] \implies \text{le-trans } (?X t) (?X u)
proof(intro le-transI le-funI, simp add:wp-eval)
fix t u::'s trans and P::'s expect and s::'s
assume le: le-trans t u
  and st: \forall P. \text{sound } P \implies \text{sound } (t P)
  and su: \forall P. \text{sound } P \implies \text{sound } (u P)
  and sP: \text{sound } P
hence \text{sound } (t P) \text{ sound } (u P) by(auto)
  with healthy-monoD[OF hwp] leP have wp body (t P) \vdash wp body (u P)
  by(auto)
  hence wp body (t P) s \le wp body (u P) s by(auto)
  thus "G" s * wp body (t P) s \le "G" s * wp body (u P) s by(auto intro:nearly-healthy-monoD)
qed

have wlp-loop-mono: \forall t u. [le-utrans t u; \forall P. \text{unitary } P \implies \text{unitary } (t P);
  \forall P. \text{unitary } P \implies \text{unitary } (u P)] \implies \text{le-utrans } (?Y t)
(?Y u)
proof(intro le-utransI le-funI, simp add:wp-eval)
fix t u::'s trans and P::'s expect and s::'s
assume le: le-utrans t u
  and ut: \forall P. \text{unitary } P \implies \text{unitary } (t P)
  and uu: \forall P. \text{unitary } P \implies \text{unitary } (u P)
  and uP: \text{unitary } P
hence \text{unitary } (t P) \text{ unitary } (u P) by(auto)
  with le uP have wlp body (t P) \vdash wlp body (u P)
  by(auto intro:nearly-healthy-monoD[OF hwlp])
  hence wlp body (t P) s \le wlp body (u P) s by(auto)
  thus "G" s * wlp body (t P) s \le "G" s * wlp body (u P) s
    by(auto intro:mult-left-mono)
qed

from hwp have hX: \forall t. \text{healthy } t \implies \text{healthy } (?X t)
4.4. CONTINUITY AND INDUCTION FOR LOOPS

by(auto intro:healthy-intros)

from hwp have hY: \( \forall t. \text{nearly-healthy} t \Rightarrow \text{nearly-healthy} (\ ?Y t) \)
by(auto intro:healthy-intros)

have PLimit: \( P \ (\text{Sup}\trans \ (\text{fst} \ ?M)) \ (\text{Inf}\trans \ (\text{snd} \ ?M)) \)
by(auto intro:Limit)

have feasible-lfp-loop:
feasible (\text{fp-trans} \ ?X)
proof(intro feasibleI bounded-byI2 nnegI2,
    simp-all add:wp-Loop1 [simplified wp-eval] soundI2 hwp)
fix P::'s expect and b::real
assume hP: bounded-by b P and hP: nneg P
hence sP: sound P by(auto)
show lfp-exp (\lambda Q s. <<G>> s * wp body Q s + <<N G>> s * P s) \vdash \lambda.s. b
proof(intro lfp-exp-lowerbound le-funI)
fix s::'s
from hP hP have nnb: 0 \leq b by(auto)
  hence sound (\lambda.s. b) bounded-by b (\lambda.s. b) by(auto)
with hP have bounded-by b (wp body (\lambda.s. b)) by(auto)
with hP have wp body (\lambda.s. b) s \leq b P s \leq b by(auto)
  hence <<G>> s * wp body (\lambda.s. b) s + <<N G>> s * P s \leq <<G>> s * b + <<N G>> s * b
by(auto intro:add-mono mult-left-mono)
thus <<G>> s * wp body (\lambda.s. b) s + <<N G>> s * P s \leq b
  by(simp add:negate-embed algebra-simps)
from nnb show sound (\lambda.s. b) by(auto)
qed

from hwp sP show \lambda.s. 0 \vdash lfp-exp (\lambda Q s. <<G>> s * wp body Q s + <<N G>> s * P s)
  s * P s)
  by(blast intro!:lfp-exp-greatest lfp-loop-fp)
qed

have unitary-gfp:
\( \forall P. \text{unitary} P \Rightarrow \text{unitary} (\text{gfp-trans} ?Y P) \)
proof(intro unitaryI2 nnegI2 bounded-byI2,
    simp-all add:wp-Loop1 [simplified wp-eval] hwp)
fix P::'s expect
assume aP: unitary P
show \lambda.s. 0 \vdash gfp-exp (\lambda Q s. <<G>> s * wp body Q s + <<N G>> s * P s)
proof(rule gfp-exp-upperbound[OF le-funI])
fix s::'s
from hwp aP have 0 \leq wp body (\lambda.s. 0) s 0 \leq P s by(auto dest!:unitary-sound)
thus 0 \leq <<G>> s * wp body (\lambda.s. 0) s + <<N G>> s * P s
  by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)
show unitary (\lambda.s. 0) by(auto)
qed

show gfp-exp (\lambda Q s. <<G>> s * wp body Q s + <<N G>> s * P s) \vdash \lambda.s. 1
have fX:
\[
\forall t. \text{feasible } t \implies \text{feasible } (?X t)
\]
proof (intro feasible I nneg I bounded-by I, simp-all add: wp-eval)
fix t::'s trans and Q: '::expect and b::real and s::'s
assume fI: feasible t and bQ: bounded-by b Q and nQ: nneg Q
hence nneg (t Q) bounded-by b (t Q) by (auto)
moreover hence stQ: sound (t Q) by (auto)
ultimately have wp body (t Q) s ≤ b using hwp by (auto)
moreover from bQ have Q s ≤ b by (auto)
ultimately have «G» s * wp body (t Q) s + (1 − «G» s) * Q s ≤
«G» s * b + (1 − «G» s) * b
by (auto intro: add-mono mult-left-mono)
thus «G» s * wp body (t Q) s + (1 − «G» s) * Q s ≤ b
by (simp add: algebra-simps)
from nQ stQ hwp have 0 ≤ wp body (t Q) s 0 ≤ Q s by (auto)
thus 0 ≤ «G» s * wp body (t Q) s + (1 − «G» s) * Q s
by (auto intro: add-nonneg-nonneg mult-nonneg-nonneg)
qed

have uY:
\[
\forall t P. (\forall P. \text{unitary } P \implies \text{unitary } (t P)) \implies \text{unitary } P \implies \text{unitary } (?Y t P)
\]
proof (intro unitary I2 nneg I bounded-by I, simp-all add: wp-eval)
fix t::'s trans and P::'s expect and s::'s
assume uI: \forall P. \text{unitary } P \implies \text{unitary } (t P)
and uP: \text{unitary } P
hence uP: \text{unitary } (t P) by (auto)
with hwp have ubtP: \text{unitary } (wlp body (t P)) by (auto)
with uP have 0 ≤ P s 0 ≤ wp body (t P) s by (auto)
thus 0 ≤ «G» s * wp body (t P) s + (1 − «G» s) * P s
by (auto intro: add-nonneg-nonneg mult-nonneg-nonneg)
from uP ubtP have P s ≤ 1 wp body (t P) s ≤ 1 by (auto)
hence «G» s * wp body (t P) s + (1 − «G» s) * P s ≤ «G» s * 1 + (1 − «G» s) * 1
by (blast intro: add-mono mult-left-mono)
also have ... = 1 by (simp add: algebra-simps)
finally show «G» s * wp body (t P) s + (1 − «G» s) * P s ≤ 1 .
qed

have fu-lfp: le-trans (Sup-trans (fst ?M)) (lfp-trans ?X)
using feasible-nneg D[OF feasible-lfp-loop]
by (intro le-trans I [OF Sup-trans least2], blast+)
hence le-trans (?X (Sup-trans (fst ?M))) (?X (lfp-trans ?X))
by (auto intro: wp-loop-mono feasible-sound [OF Sup]
feasible-sound [OF feasible-lfp-loop])
also have equiv-trans ... (lfp-trans ?X)

proof (rule iffD1[OF equiv-trans-comm, OF lfp-trans-unfold], iprover intro:wp-loop-mono)

fix t::'s trans and P::'s expect

assume st: \( \exists Q. \) sound \( Q \implies \) sound \( t \) \( Q \)

and sP: sound \( P \)

show sound \( \forall X. t P \)

proof (intro soundD2 bounded-by inegI, simp-all add:wp-eval)

fix s::'s

from sP st hwp have \( 0 \leq P s 0 \leq \) wp body \( (t P) s \) by(auto)

thus \( 0 \leq \langle G \rangle s * \) wp body \( (t P) s + (1 - \langle G \rangle s) * P s \)

by (blast intro:add-ineq add-nonneg-nonneg)

from sP st have bounded-by \( (\text{bound-of} (t P)) (t P) \) by(auto)

with sP st hwp have bounded-by \( (\text{bound-of} (t P)) (\text{wp body} (t P)) \) by(auto)

hence wp body \( (t P) s \leq \) bound-of \( (t P) \) by(auto)

moreover from sP st hwp have \( P s \leq \) bound-of \( P \) by(auto)

moreover have \( \langle G \rangle s \leq 1 \) by(auto)

moreover from sP st hwp have \( 0 \leq \) wp body \( (t P) s \) \( \leq P s \) by(auto)

moreover have \( (0::real) \leq 1 \) by(simp)

ultimately show \( \langle G \rangle s * \) wp body \( (t P) s + (1 - \langle G \rangle s) * P s \leq 1 * \) bound-of \( (t P) + 1 * \) bound-of \( P \)

by (blast intro:add-ineq mono)

qed

next

let \( \bar{f}_P = \lambda R. s. \) 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 \( (?X (\text{Sup-trans} (\text{fst} \ ?M))) (\text{lfp-trans} ?X) \).

have fu-gfp: le-utrans \( (\text{gfp-trans} ?Y) (\text{Inf-utrans} (\text{snd} \ ?M)) \)

by(auto intro:Inf-utrans-greatest unitary-gfp)

have equiv-utrans \( (\text{gfp-trans} ?Y) \ ?Y \) \( (\text{gfp-trans} ?Y) \)

by(auto intro!:gfp-trans-unfold wp-loop-mono u\text{Y})

also from fu-gfp have le-utrans \( (?Y (\text{gfp-trans} ?Y)) \ ?Y \) \( (\text{Inf-utrans} (\text{snd} \ ?M))) \)

by(auto intro:wp-loop-mono u\text{Y} unitary-gfp)

finally have ge-gfp: le-utrans \( (\text{gfp-trans} ?Y) \ ?Y \) \( (\text{Inf-utrans} (\text{snd} \ ?M)) \).

from PLimit fX u\text{Y} fSup u\text{Inf} have \( P \ ?X (\text{Sup-trans} (\text{fst} \ ?M)) \ ?Y \) \( (\text{Inf-utrans} (\text{snd} \ ?M))) \)

by (iprover intro:IH)

moreover from fSup have feasible \( (?X (\text{Sup-trans} (\text{fst} \ ?M))) \) by(rule fX)

moreover have \( \forall P. \) \text{unitary} \( P \implies \text{unitary} \ ?Y \) \( (\text{Inf-utrans} (\text{snd} \ ?M)) \) \( P \)

by (auto intro:u\text{Y} unitary)

moreover note le-lfp ge-gfp

ultimately have pair-in: \( (?X (\text{Sup-trans} (\text{fst} \ ?M)), \ ?Y (\text{Inf-utrans} (\text{snd} \ ?M))) \) \( \in \ ?M \)

by(simp)
have \(?X\) \((\text{Sup-trans} \ (\text{fst} \ ?M))\) \(\in\) \(\text{fst} \ ?M\)
by (rule imageI [OF pair-in, of \text{fst}, simplified])
hence \(\text{le-trans} \ (?X\ (\text{Sup-trans} \ (\text{fst} \ ?M)))\)
(proof (rule le-transI [OF Sup-trans-upper2 [where \(t=?X\) \((\text{Sup-trans} \ (\text{fst} \ ?M))\)
and \(S=\text{fst} \ ?M\)]])

fix \(P::\)'s expect
assume \(sP\); sound \(P\)
thus \(\negneg\ P\) by (auto)

from \(sP\) show bounded-by \((\text{bound-of} \ P\) \(P\) by (auto)

by (auto)

qed

hence \(\text{le-trans} \ (\text{lfp-trans} \ ?X)\ (\text{Sup-trans} \ (\text{fst} \ ?M))\)
by (auto intro : lfp-trans-lowerbound feasible-sound [OF fSup])

with \(\text{fu-lfp}\) have eqt: \(\text{equiv-trans} \ (\text{Sup-trans} \ (\text{fst} \ ?M))\)
(by (rule le-trans-antisym)

have \(?Y\) \((\text{Inf-utrans} \ (\text{snd} \ ?M))\) \(\in\) \(\text{snd} \ ?M\)
by (rule imageI [OF pair-in, of \text{snd}, simplified])
hence \(\text{le-utrans} \ (\text{Inf-utrans} \ (\text{snd} \ ?M))\)
(by intro Inf-utrans-lower, auto)
hence \(\text{le-utrans} \ (\text{Inf-utrans} \ (\text{snd} \ ?M))\)
(by blast intro: gfp-trans-upperbound uInf)

with \(\text{fu-gfp}\) have equ: \(\text{equiv-utrans} \ (\text{Inf-utrans} \ (\text{snd} \ ?M))\)
(by auto intro: le-utrans-antisym)
from \(\text{PLimit eqt equ}\) show \(P\) \((\text{lfp-trans} \ ?X)\) \((\text{gfp-trans} \ ?Y)\)
(by (rule P-equiv)

qed

4.4.1 The Limit of Iterates

The iterates of a loop are its sequence of finite unrollings. We show shortly that this converges on the least fixed point. This is enormously useful, as we can appeal to various properties of the finite iterates (which will follow by finite induction), which we can then transfer to the limit.

definition iterates :: 's prog ⇒ ('s ⇒ bool) ⇒ nat ⇒ 's trans
where iterates body \(G\) \(i\) = ((\(\lambda x\). \(\text{wp} \ (\text{body} ; ; \text{Embed} x \ G \oplus \text{Skip})\)) \(\sim\) \(i\) \(\lambda P\) \(s\).

lemma iterates-0[simp]:
iterates body \(G\) \(0\) = \((\lambda P\) \(s\).
by (simp add: iterates-def)

lemma iterates-Suc[simp]:
iterates body \(G\) \((\text{Suc} \ i)\) = \(\text{wp} \ (\text{body} ; ; \text{Embed} \ (\text{iterates body} \ G\ i) \ G\ i \oplus \text{Skip})\)
by (simp add: iterates-def)

All iterates are healthy.
4.4. CONTINUITY AND INDUCTION FOR LOOPS

**Lemma** *iterates-healthy*:

\[ \text{healthy (wp body)} \Rightarrow \text{healthy (iterates body G i)} \]

*by (induct i, auto intro:healthy-intros)*

The iterates are an ascending chain.

**Lemma** *iterates-increasing*:

*fixes* body: ‘s prog
*assumes* hb: healthy (wp body)
*shows* le-trans (iterates body G i) (iterates body G (Suc i))

*proof* (induct i)

*show* le-trans (iterates body G 0) (iterates body G (Suc 0))

*proof* (simp add: iterates-def, rule le-transI)

*fix* P: ‘s expect
*assume* sound P
*with* hb have sound (wp (body ;; Embed (\(\lambda P \ s . \ 0\) * G \oplus Skip) P))
*by (auto intro:wp-loop-step-sound)

*thus* \(\lambda s. \ 0 \triangleright wp (body ;; Embed (\(\lambda P \ s . \ 0\) * G \oplus 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))

\(wp (body ;; Embed (iterates body G i) * G \oplus Skip))\)
*by (simp)

also from iterates-healthy[OF hb]
*have* le-trans ... (wp (body ;; Embed (iterates body G (Suc i)) * G \oplus Skip))
*by (blast intro:wp-loop-step-mono[OF hb IH])

also have equiv-trans ... (iterates body G (Suc (Suc i)))*
*by (simp)

finally *show* le-trans (iterates body G (Suc i)) (iterates body G (Suc (Suc i))) .

qed

**Lemma** *wp-loop-step-bounded*:

*fixes* t: ‘s trans and Q: ‘s expect
*assumes* nQ: nneg Q
and bQ: bounded-by b Q
and ht: healthy t
and hb: healthy (wp body)
*shows* bounded-by b (wp (body ;; Embed t * G \oplus 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 \leq b Q s \leq b by (auto)
hence \(\langle G\rangle s * wp body (t Q) s + (1 - \langle G\rangle s) * Q s \leq \langle G\rangle s + \langle G\rangle s + (1 - \langle G\rangle s) * b\)
*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 = λP::'s ⇒ real s::real
have HF: ∃i. healthy ((?F ^^ i) ?bot)
proof
  fix i from hb show (?thesis i)
  by (induct i, simp-all add: healthy-intros)
qed

from iterates-healthy[OF hb] have ∃i. feasible (iterates body G i) by (auto)
hence fSup: feasible (Sup-trans (range (iterates body G)))
  by (auto intro: feasible-Sup-trans)

{ fix i
  have le-trans ((?F ^^ i) ?bot) ?X
  proof (induct i)
    show le-trans ((?F ^^ 0) ?bot) ?X
    proof ( simp, intro le-transI)
      fix P::'s expect
      assume sound P
      with hb healthy-wp-loop
      have sound (wp (µ x. body ;; x « G » ⊕ Skip) P)
        by (auto)
      thus λs. 0 ⊢ wp (µ x. body ;; x « G » ⊕ Skip) P
        by (auto)
      qed
  fix i
  assume IH: le-trans ((?F ^^ i) ?bot) ?X
  have equiv-trans ((?F ^^ (Suc i)) ?bot) (?F ((?F ^^ i) ?bot)) by (simp)
  also have le-trans ... (?F ?X)
  proof (rule wp-loop-step-mono[OF hb IH])
    fix P::'s expect
    assume sP: sound P
    with hb healthy-wp-loop
show sound (wp (μ x. body ;; x ⸞ G ⊕ Skip) P)
by(auto)
from sP show sound ((?F ⟨≈ i⟩) ?bot P)
by(rule healthy-sound[OF HF])
qed
also {
from hb have X: le-trans (wp (body ;; Embed (λP s. bound-of P) « G ⊕ Skip))
(λP s. bound-of P)
by(intro le-transI, simp add:wp-eval, auto intro: lfp-loop-fp[unfolded negate-embed])
have equiv-trans (?F ?X) ?X
apply (simp only: wp-eval)
by(intro ifD1[OF equiv-trans-comm, OF lfp-trans-unfold]
wp-loop-step-mono[OF hb] wp-loop-step-sound[OF hb], (blast|rule X)+)
}
finally show le-trans ((?F ⟨≈ (Suc i)⟩) ?bot) ?X.
qed
}
hence ∀i. le-trans (iterates body G i) (wp do G → body od)
by(simp add:iterates-def)
thus le-trans ?Y ?X
by(auto intro! le-transI[OF Sup-trans-least2] sound-nneg
healthy-sound[OF iterates-healthy, OF hb]
healthy-bounded-byD[OF iterates-healthy, OF hb]
healthy-sound[OF healthy-wp-loop hb])

show le-trans ?X ?Y
proof(simp only: wp-eval, rule lfp-loop-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 (λF o (iterates body G))))
by (auto intro! healthy-feasibleD bd-cts-trD cong del: image-cong-simp)
also have le-trans (Sup-trans (range (λF o (iterates body G)))) ?Y
proof(rule le-transI)
fix P::'s expect
assume sP: sound P
show (Sup-trans (range (λF o (iterates body G)))) P |- ?Y P
proof(rule Sup-trans-least2, clarsimp)
show ∀u∈range (λx. wp (body ;; Embed x « G ⊕ Skip)) o iterates body G).
∀R. nneg R ∧ bounded-by (bound-of P) R →
neg (u R) ∧ bounded-by (bound-of P) (u R)
proof(clarsimp, intro conjI)
fix Q::'s expect and i
assume nQ: nneg Q and bQ: bounded-by (bound-of P) Q
hence sound Q by(auto)
moreover from iterates-healthy[OF hb]
have \( P \text{ sound } G \implies \text{ sound (iterates body } G \text{ i } P) \) by(auto)
moreover note \( hb \)
ultimately have \( \text{ sound (wp (body ; Embed (iterates body } G \text{ i} ) \oplus G) \) by(auto)

\[ \begin{align*}
\text{ Skip) } Q) & \quad \text{ by/iprover intro:wp-loop-step-sound) } \\
\text{ thus } \neg \text{ neg} \ (\wp (\text{body ; Embed (iterates body } G \text{ i} ) \oplus G) \oplus \text{ Skip) } Q) & \quad \text{ by/auto) } \\
\text{ from } \neg Q \ bQ \text{ iterates-healthy[OF } hb] \ hb \ & \quad \text{ show bounded-by (bound-of } P) \ (wp (\text{body ; Embed (iterates body } G \text{ i} ) \oplus G) \oplus \text{ Skip) } Q) & \quad \text{ by/auto) }
\end{align*} \]

Thus \( \text{ nneg (wp (body ; Embed (iterates body } G \text{ i} ) \oplus G) \oplus \text{ Skip) } Q) \) by(auto)
from \( \text{ sP show } \neg \text{ neg } P \text{ bounded-by (bound-of } P) \ P \) by(auto)
next
fix \( i \)
show \( \wp (\text{body ; Embed (iterates body } G \text{ i} ) \oplus G) \oplus \text{ Skip) } Q \vdash \\
\text{ Sup-trans (range (iterates body } G)) \ Q \) proof(rule Sup-trans-upper2[OF - - nQ bQ])
from iterates-healthy[OF hb]
show \( \forall u \in \text{range (iterates body } G). \\
\forall R. \neg \text{ neg} R \land \text{ bounded-by (bound-of } P) \ R \implies \\
\neg \text{ neg} (u \ R) \land \text{ bounded-by (bound-of } P) \ (u \ R) \) by(auto)
have \( \wp (\text{body ; Embed (iterates body } G \text{ i} ) \oplus G) \oplus \text{ Skip) } = \text{ iterates body } G \ (\text{Suc } i) \) by(simp)
also have \( \ldots \in \text{range (iterates body } G) \) by(blast)
finally show \( \wp (\text{body ; Embed (iterates body } G \text{ i} ) \oplus G) \oplus \text{ Skip) } \in \\
\text{range (iterates body } G). \)
qed
qed
qed
finally show \( \text{ le-trans (?F } ?Y) ?Y. \)

fix \( P::\text{s expect} \\
assume \text{ sound } P \)
with \( \text{fSup show } \text{ sound (} ?Y P) \) by(auto)
qed
qed

Therefore, evaluated at a given point (state), the sequence of iterates gives
a sequence of real values that converges on that of the loop itself.
**4.4. CONTINUITY AND INDUCTION FOR LOOPS**

**Corollary: Loop-Iterates**

*Fixes body:*'s prog

*Assumes* \( hb: healthy \; (wp \; body) \)

*and ch: bd-cts \; (wp \; body) *

*and \( sP: sound \; P \)

*Shows* \( (λi. \; iterates \; body \; G \; i \; s) \quad \text{---} \quad wp \; (do \; G \quad \text{body} \; od) \; P \; s \)

**Proof**

- *Let* \( \{ f \; s \mid f, \; f \in \{ t \; P \mid t \in range \; (iterates \; body \; G) \} \} \)

- *Have* \( closure-Sup: \; Sup \; \{ f \; s \mid f, \; f \in \{ t \; P \mid t \in range \; (iterates \; body \; G) \} \} \)

  - *Fix* \( i \)

  - From \( sP \) *have* \( bounded-by \; (bound-of \; P) \; P \) *by(auto)*

    - With \( iterates-healthy[OF \; hb] \; sP \) *have* \( \bigwedge j. \; bounded-by \; (bound-of \; P) \; (iterates \; body \; G \; j \; P) \)

      - By(auto)

  - Thus *iterates body G i P s \leq bound-of P* *by(auto)*

*Qed*

- *Have* \( (λi. \; iterates \; body \; G \; i \; s) \quad \text{---} \quad Sup \; \{ f \; s \mid f, \; f \in \{ t \; P \mid t \in range \; (iterates \; body \; G) \} \} \)

  - *Proof* \( (rule \; LIMSEQ-I) \)

    - *Fix r::real* *assume* \( 0 < r \)

      - With \( closure-Sup \) *obtain* \( y \) *where* \( yin: \; y \in \; \{ f \; s \mid f, \; f \in \{ t \; P \mid t \in range \; (iterates \; body \; G) \} \} \) and \( ey: \; dist \; y \; (Sup \; \{ f \; s \mid f, \; f \in \{ t \; P \mid t \in range \; (iterates \; body \; G) \} \} < r \)

        - By(simp only: closure-approachable, blast)

    - From \( yin \) *obtain* \( i \) *where* \( git: \; y = \; iterates \; body \; G \; i \; s \) *by(auto)*

      - *Fix j*

        - *Have* \( i \leq j \quad \text{---} \quad le-trans \; (iterates \; body \; G \; i) \; (iterates \; body \; G \; j) \)

          - *Proof* \( (induct \; j, \; simp, \; clarify) \)

            - *Fix k*

              - *Assume* \( IH: \; i \leq k \quad \text{---} \quad le-trans \; (iterates \; body \; G \; i) \; (iterates \; body \; G \; k) \) and \( le: \; i \leq Suc \; k \)

                - *Show* \( le-trans \; (iterates \; body \; G \; i) \; (iterates \; body \; G \; (Suc \; k)) \)

                    - *Proof* \( (cases \; i = Suc \; k, \; simp) \)

                      - *Assume* \( i \neq Suc \; k \)

                        - With \( le \) *have* \( i \leq k \) *by(auto)*

                          - With \( IH \) *have* \( le-trans \; (iterates \; body \; G \; i) \; (iterates \; body \; G \; k) \) *by(auto)*

                            - Also note \( iterates-increasing[OF \; hb] \)

                              - Finally *show* \( le-trans \; (iterates \; body \; G \; i) \; (iterates \; body \; G \; (Suc \; k)) \).

                      *Qed*

            *Qed*

    - *With* \( sP \) *have* \( \forall j\geq i. \; iterates \; body \; G \; i \; P \; s \leq iterates \; body \; G \; j \; P \; s \) *by(auto)*

      - *Moreover* \{*

        - From \( sP \) *have* \( bounded-by \; (bound-of \; P) \; P \) *by(auto)*

          - With \( iterates-healthy[OF \; hb] \; sP \) *have* \( \bigwedge j. \; bounded-by \; (bound-of \; P) \; (iterates \; body \; G \; j \; P) \)

            - By(auto)

    *Qed*
hence $\forall j. \text{iterates body } G j P s \leq \text{bound-of } P$ by (auto)

hence $\forall j. \text{iterates body } G j P s \leq \text{Sup } ?X$

by (intro cSup-upper bdd-aboveI, auto)

ultimately have $\forall j. i \leq j \implies \text{norm (iterates body } G j P s - \text{Sup } ?X) \leq \text{norm (iterates body } G i P s - \text{Sup } ?X)$

by (auto)

also from ey yit have $\text{norm (iterates body } G i P s - \text{Sup } ?X) < r$

by (simp add: dist-real-def)

finally show $\exists \text{no. } \forall n \geq \text{no. } \text{norm (iterates body } G n P s - \text{Sup } \{f s | f \in \{t P | t \in \text{range (iterates body } G)\}\}) < r$

by (auto)

qed

moreover

from hb cb sP have $\text{wp do } G \to \text{body od } P s = \text{Sup-trans (range (iterates body } G) ) P s}$

by (simp add: equiv-transD [OF lfp-iterates])

moreover have $\ldots = \text{Sup } \{f s | f \in \{t P | t \in \text{range (iterates body } G)\}\}$

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** body::′s prog

**Assumes** hb: healthy (wp body)

**and** cb: bd-cts (wp body)

**Shows** bd-cts (iterates body G i)

**Proof (induct i, simp-all)**

**Have** range $\lambda (n::nat) (s::'s). 0::\text{real}) = \{\lambda s. 0::\text{real}\}$

by (auto)

**Thus** bd-cts ($\lambda P (s::'s). 0$)

by (intro bd-ctsI, simp add: a-def Sup-exp-def)

**Next**

**Fix** i

**Assume** IH: bd-cts (iterates body G i)

**Thus** bd-cts (wp (body ;; Embed (iterates body G i) @ G @ Skip))


qed

Therefore so is the loop itself.

**Lemma cts-wp-loop:**

**Fixes** body::′s prog

**Assumes** hb: healthy (wp body)

**and** cb: bd-cts (wp body)

**Shows** bd-cts (wp do G -> body od)
**proof** \(\text{rule bd-ctsI}\)

fix \(M::\text{nat} \Rightarrow 's\) expect and \(b::\text{real}\)

assume chain: \(\forall i. M \vdash M (\text{Suc } i)\)

and \(sM: \forall i. \text{sound } (M i)\)

and \(bM: \forall i. \text{bounded-by } b (M i)\)

from \(sM\) \(bM\) iterates-healthy\([\text{OF } hb]\)

have \(\forall j. i. \text{bounded-by } b (\text{iterates body } G i (M j))\) by\(\text{(blast)}\)

hence \(\forall i. \forall j. s. \text{iterates body } G i (M j)\) \(s \leq b\) by\(\text{(auto)}\)

from \(sM\) \(bM\) have \(\text{Sup}: \text{sound } (\text{Sup-exp } (\text{range } M))\)

by\(\text{(auto intro;Sup-exp-sound)}\)

with \(\text{lfp-iterates}\([\text{OF } hb\ cb]\)\)

have wp do \(G \rightarrow\) body od \((\text{Sup-exp } (\text{range } M)) = \)

\(\text{Sup-trans } (\text{range (iterates body } G)) (\text{Sup-exp } (\text{range } M))\)

by\(\text{(simp add:equiv-transD)}\)

also \{ from chain \(sM\) \(bM\)

have \(\forall i. \text{iterates body } G i (\text{Sup-exp } (\text{range } M)) = \text{Sup-exp } (\text{range (iterates body } G i \circ M))\)

by\(\text{(blast intro:bd-ctsD cts-iterates[OF hb cb])}\)

hence \(\{\{ (\text{Sup-exp } (\text{range } M)) | t. t \in \text{range (iterates body } G)\}\}\)

by\(\text{(auto intro:sym)}\)

hence \(\text{Sup-trans } (\text{range (iterates body } G)) (\text{Sup-exp } (\text{range } M)) = \)

\(\text{Sup-exp } (\text{Sup-exp } (\text{range } (t \circ M)) | t. t \in \text{range (iterates body } G)\})\)

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

\}

also \{ have \(\forall s. \{ f | f. \exists t. f = (\lambda s. \text{Sup } \{ f s | f. f \in \text{range } (t \circ M)\}) \land t \in \text{range (iterates body } G)\} = \)

\(\text{range } (\lambda i. \text{Sup } (\lambda j. \text{iterates body } G i (M j) s)))\)

(is \(\forall s. ?X s = ?Y s)\)

**proof**(intro antisym subsetI)

fix \(s x\)

assume \(x \in ?X s\)

then obtain \(t\) where \(\text{rwx: } x = \text{Sup } \{ f s | f. f \in \text{range } (t \circ M)\}\)

and \(t \in \text{range (iterates body } G)\) by\(\text{(auto)}\)

then obtain \(i\) where \(t = \text{iterates body } G i\) by\(\text{(auto)}\)

with \(\text{rwx}\) have \(x = \text{Sup } \{ f s | f. f \in \text{range } (\lambda j. \text{iterates body } G i (M j))\}\)

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

moreover have \(\{ f s | f. f \in \text{range } (\lambda j. \text{iterates body } G i (M j))\} = \)

\(\text{range } (\lambda j. \text{iterates body } G i (M j) s)\) by\(\text{(auto)}\)

ultimately have \(x = \text{Sup } (\text{range } (\lambda j. \text{iterates body } G i (M j) s))\)

by\(\text{(simp)}\)

thus \(x \in \text{range } (\lambda i. \text{Sup } (\lambda j. \text{iterates body } G i (M j) s)))\)

by\(\text{(auto)}\)

next

fix \(s x\)
assume $x \in \exists Y s$
then obtain $i$ where $A$: $x = \text{Sup} (\text{range} (\lambda j. \text{iterates body} G i (M j) s))$
  by(auto)

have $\exists s. \{ f s \mid f \in \text{range} (\lambda j. \text{iterates body} G i (M j)) \} =$
  $\text{range} (\lambda j. \text{iterates body} G i (M j) s)$ by(auto)

hence $B$: $(\lambda s. \text{Sup} (\text{range} (\lambda j. \text{iterates body} G i (M j) s))) =$
  $(\lambda s. \text{Sup} \{ f s \mid f \in \text{range} (\text{iterates body} G i o M) \})$
  by(simp add:o-def)

have $C$: iterates body $G i \in \text{range} (\text{iterates body} G)$ by(auto)

have $\exists f. x = f s \wedge$
  $(\exists t. f = (\lambda s. \text{Sup} \{ f s \mid f \in \text{range} (t \circ M) \}) \wedge$
  $t \in \text{range} (\text{iterates body} G))$
  by(iprove intro:A B C)
thus $x \in \exists X s$ by(simp)

qed

hence Sup-exp $\{ \text{Sup-exp} (\text{range} (t \circ M)) \mid t \in \text{range} (\text{iterates body} G) \} =$
  $(\lambda s. \text{Sup} (\text{range} (\lambda i. \text{Sup} (\text{range} (\lambda j. \text{iterates body} G i (M j) s))))))$
  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 $\exists X = \exists Y$)

proof(rule ext, rule antisym)
  fix s::'s
  show $\exists Y s \leq \exists X s$
    proof(rule cSup-least, blast, clarify)
      fix i::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-above1, 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-above1, (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 $\exists X s \leq \exists Y s$
  by(intro cSup-least, blast, clarify, simp, 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
4.4. CONTINUITY AND INDUCTION FOR LOOPS

have \( \bigwedge j\). iterates body \( G \) \( i (M j) s \leq \Sup (\text{range } (\lambda i. \text{iterates body } G \ i \ (M j) s)) \)

by(rule cSup-upper, auto intro:iB)

thus \( ?Y s \leq ?X s \)

by(intro cSup-least, blast, clarify, simp, blast intro:cSup-least)

show \( ?X s \leq ?Y s \)

proof(rule cSup-least, blast, clarify)

fix \( i j :: \text{nat} \)

from \( iB \) have iterates body \( G \) \( i (M j) s \leq \Sup (\text{range } (\lambda i. \text{iterates body } G \ i \ (M j) s)) \)

by(intro cSup-upper bdd-aboveI, auto)

also from \( iB \) have \( ... \leq \Sup (\text{range } (\lambda j. \text{Sup (range } (\lambda i. \text{iterates body } G \ i \ (M j) s)))) \)

by(intro cSup-upper cSup-least bdd-aboveI, blast, blast intro:cSup-least)

finally show iterates body \( G \) \( i (M j) s \leq \Sup (\text{range } (\lambda j. \text{Sup (range } (\lambda i. \text{iterates body } G \ i \ (M j) s)))) \).

qed

from \( iB \) have iterates body \( G \) \( i (M j) s \leq \Sup (\text{range } (\lambda i. \text{iterates body } G \ i \ (M j) s)) \)

by(auto)

assume \( x \in ?X s \)

then obtain \( j \) where rwx: \( x = \Sup (\text{range } (\lambda i. \text{iterates body } G \ i \ (M j) s)) \)

by(auto)

moreover {

have \( \bigwedge s. \text{range } (\lambda j. \text{Sup (range } (\lambda i. \text{iterates body } G \ i \ (M j) s)))) = \{ f s | f. \exists t. f = t (M j) \land t \in \text{range } (\text{iterates body } G) \} \)

by(auto)

hence \( \bigwedge s. \text{Sup (range } (\lambda i. \text{iterates body } G \ i \ (M j) s))) \in \text{range } (\lambda P s. \text{Sup } \{ f s | f. \exists t. f = t P \land t \in \text{range } (\text{iterates body } G) \}) \circ M \)

by (simp add: o-def cong del: SUP-cong-simp)

}

ultimately show \( x \in ?Y s \) by(auto)

next

fix \( s x \)

assume \( x \in ?Y s \)

then obtain \( P \) where rwx: \( x = P s \)

and \( \text{Pin: } P \in \text{range } ((\lambda P s. \text{Sup } \{ f s | f. \exists t. f = t P \land t \in \text{range } (\text{iterates body } G) \}) \circ M) \)

by(auto)

then obtain \( j \) where \( P = (\lambda s. \text{Sup } \{ f s | f. \exists t. f = t (M j) \land t \in \text{range } (\text{iterates body } G) \}) \)

by(auto)

also {

have \( \bigwedge s. \{ f s | f. \exists t. f = t (M j) \land t \in \text{range } (\text{iterates body } G) \} = \)

ultimately show \( x \in ?Y s \) by(auto)
\[
\text{range } (\lambda_i \text{. iterates body } G \ i \ (M \ j) \ s) \ \text{by(auto)}
\]

\[
\text{hence } (\lambda s. \text{Sup } \{ f \ s \ | \ \exists t. f = t (M \ j) \land t \in \text{range } (\text{iterates body } G)\}) = \\
(\lambda s. \text{Sup } (\text{range } (\lambda_i \text{. iterates body } G \ i \ (M \ j) \ s)))
\]

\[
\text{by(simp)}
\]

finally have \(x = \text{Sup } (\text{range } (\lambda_i \text{. iterates body } G \ i \ (M \ j) \ s))\)

\[
\text{by(simp add:rwx)}
\]

thus \(x \in \emptyset X \ s\) by(simp)

qed

hence \(\lambda s. \text{Sup } (\text{range } (\lambda_j \text{. Sup } (\text{range } (\lambda_i \text{. iterates body } G \ i \ (M \ j) \ s))))\) =

\[
\text{Sup-exp } (\text{range } (\text{iterates body } G) \circ M)
\]

\[
\text{by (simp add: Sup-exp-def Sup-trans-def cong del: SUP-cong-simp)}
\]

also have \(\text{Sup-exp } (\text{range } (\text{iterates body } G) \circ M) = \\
\text{Sup-exp } (\text{range } (\text{wp do } G \rightarrow \text{body od o M}))\)

\[
\text{by(simp add:o-def equiv-transD[OF lfp-iterates, OF hb cb, OF sM])}
\]

finally show \(\text{wp do } G \rightarrow \text{body od } (\text{Sup-exp } (\text{range } M)) \rightarrow \\
\text{Sup-exp } (\text{range } (\text{wp do } G \rightarrow \text{body od o M}))\).

qed

lemmas cts-intros =
cts-wp-Abort cts-wp-Skip
ccts-wp-Seq cts-wp-PC
ccts-wp-DC cts-wp-Embed
ccts-wp-Apply ccts-wp-SetDC
ccts-wp-SetPC ccts-wp-Bind
ccts-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)

\[
\text{by(auto simp:wp-eval)}
\]

lemma sublinear-wp-Abort:
sublinear (wp Abort)

\[
\text{by(auto simp:wp-eval)}
\]

lemma sublinear-wp-Apply:
4.5. SUBLINEARITY

sublinear (wp (Apply f))
by(auto simp:wp-eval)

lemma sublinear-wp-Seq:
  fixes x::'s prog
  assumes slx: sublinear (wp x) and sly: sublinear (wp y)
  and hx: healthy (wp x) and hy: healthy (wp y)
  shows sublinear (wp (x ;; y))
proof(rule sublinearI, simp add:wp-eval)
fix P::'s ⇒ real and Q::'s ⇒ real and s::'s
and a::real and b::real and c::real
assume sP: sound P and sQ: sound Q
and nna: 0 ≤ a and nnb: 0 ≤ b and nnc: 0 ≤ c
with slx hy have a * wp x (wp y P) s + b * wp x (wp y Q) s ⊙ c ≤
  wp x (λs. a * wp y P s + b * wp y Q s ⊙ c) s
by(blast intro:sublinearD)
also { from sP sQ nna nnb nnc sly
  have (∃s. a * wp y P s + b * wp y Q s ⊙ c ≤
    wp y (λs. a * P s + b * Q s ⊙ c) s)
    by(blast intro:sublinearD)
  moreover from sP sQ 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-intrns tminus-sound)
  moreover from sP sQ nna nnb nnc
  have sound (λs. a * P s + b * Q s ⊙ c)
    by(auto intro:sound-intrns tminus-sound)
  moreover with hy have sound (wp y (λs. a * P s + b * Q s ⊙ c))
    by(blast)
  ultimately
  have wp x (λs. a * wp y P s + b * wp y Q s ⊙ c) s ≤
    wp x (wp y (λs. a * P s + b * Q s ⊙ c)) s
    by(blast intro!:le-funD[OF mono-transD[OF healthy-monoD[OF hx]]])
} finally show a * wp x (wp y P) s + b * wp x (wp y Q) s ⊙ c ≤
  wp x (wp y (λs. a * P s + b * Q s ⊙ c)) s.
qed

lemma sublinear-wp-PC:
  fixes x::'s prog
  assumes slx: sublinear (wp x) and sly: sublinear (wp y)
  and uP: unitary P
  shows sublinear (wp (x ⊕ 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 \(\text{sr}: \text{sound } R\) and \(sQ: \text{sound } Q\)
and \(\text{nn}: 0 \leq a\) and \(\text{nnb}: 0 \leq b\) and \(\text{nnn}: 0 \leq c\)

have \(a \cdot (P \cdot s \cdot \text{wp } x \cdot Q \cdot s + (1 - P) \cdot s \cdot \text{wp } y \cdot Q \cdot s) + \)
   \(b \cdot (P \cdot s \cdot \text{wp } x \cdot R \cdot s + (1 - P) \cdot s \cdot \text{wp } y \cdot R \cdot s) \cup c = \)
   \((P \cdot s \cdot a \cdot \text{wp } x \cdot Q \cdot s + (1 - P) \cdot s \cdot a \cdot \text{wp } y \cdot Q \cdot s) + \)
   \((P \cdot s \cdot b \cdot \text{wp } x \cdot R \cdot s + (1 - P) \cdot s \cdot b \cdot \text{wp } y \cdot R \cdot s) \cup c \)
by(simp add:field-simps)
also
have \(\ldots = (P \cdot s \cdot (a \cdot \text{wp } x \cdot Q \cdot s + b \cdot \text{wp } x \cdot R \cdot s) + \)
   \((1 - P) \cdot s \cdot (a \cdot \text{wp } y \cdot Q \cdot s + (1 - P) \cdot s \cdot b \cdot \text{wp } y \cdot R \cdot s) \cup c \)
by(simp add:ac-simps)
also
have \(\ldots \leq (P \cdot s \cdot (a \cdot \text{wp } x \cdot Q \cdot s + b \cdot \text{wp } x \cdot R \cdot s) \cup P \cdot s \cdot c) + \)
   \((1 - P) \cdot s \cdot ((a \cdot \text{wp } y \cdot Q \cdot s + b \cdot \text{wp } y \cdot R \cdot s) \cup (1 - P) \cdot s \cdot c) \)
by(rule tminus-add-mono)
also \{ 
   from \(uP\) have \(0 \leq P\) and \(0 \leq 1 - P\)
   by auto
   hence \((P \cdot s \cdot (a \cdot \text{wp } x \cdot Q \cdot s + b \cdot \text{wp } x \cdot R \cdot s) \cup P \cdot s \cdot c) + \)
      \((1 - P) \cdot s \cdot ((a \cdot \text{wp } y \cdot Q \cdot s + b \cdot \text{wp } y \cdot R \cdot s) \cup (1 - P) \cdot s \cdot c) = \)
      \(P \cdot s \cdot ((a \cdot \text{wp } x \cdot Q \cdot s + b \cdot \text{wp } x \cdot R \cdot s) \cup c) + \)
      
by(simp add:tminus-left-distrib)
\}
also \{ 
   from \(sQ\) \(sR\) \(\text{nn}\) \(\text{nnb}\) \(\text{nnn}\) \(\text{slx}\)
   have \(a \cdot \text{wp } x \cdot Q \cdot s + b \cdot \text{wp } x \cdot R \cdot s \cup c \leq \)
      \(\text{wp } x \cdot (\lambda s. a \cdot Q \cdot s + b \cdot R \cdot s \cup c)\) \(s\)
   by(blast)
moreover
   from \(sQ\) \(sR\) \(\text{nn}\) \(\text{nnb}\) \(\text{nnn}\) \(\text{slx}\)
   have \(a \cdot \text{wp } y \cdot Q \cdot s + b \cdot \text{wp } y \cdot R \cdot s \cup c \leq \)
      \(\text{wp } y \cdot (\lambda s. a \cdot Q \cdot s + b \cdot R \cdot s \cup c)\) \(s\)
   by(blast)
moreover
   from \(uP\) have \(0 \leq P\) and \(0 \leq 1 - P\)
   by auto
ultimately
   have \(P \cdot s \cdot ((a \cdot \text{wp } x \cdot Q \cdot s + b \cdot \text{wp } x \cdot R \cdot s) \cup c) + \)
      
by(blast intro:add-mono mult-left-mono)
4.5. SUBLINEARITY

\[ \text{finally show } \]
\[ a \ast (P \ast wp x Q s + (1 - P) \ast wp y Q s) + \]
\[ b \ast (P \ast wp x R s + (1 - P) \ast wp y R s) \ominus c \leq \]
\[ P \ast wp x (\lambda s. a \ast Q s + b \ast R s \ominus c) s + \]
\[ (1 - P) \ast wp y (\lambda s. a \ast Q s + b \ast R s \ominus c) s . \]

qed

\text{lemma sublinear-wp-DC:}

\text{fixes } x::'s \text{ prog}
\text{assumes slx: sublinear (wp x) and sly: sublinear (wp y)}
\text{shows sublinear (wp (x }\text{ Int } y))
\text{proof(rule sublinearI, simp only: wp-eval)}
\text{fix } R::'s \Rightarrow \text{real and Q::'s }\Rightarrow \text{real and s::'}s
\text{and a::real and b::real and c::real}
\text{assume slR: sound R and sQ: sound Q}
\text{and nna: } 0 \leq a \text{ and nnb: } 0 \leq b \text{ and nnc: } 0 \leq c
\text{from nna nnb}
\text{have } a \ast \min (wp x Q s) (wp y Q s) +
\text{b \ast \min (wp x R s) (wp y R s) } \ominus c =
\text{min (a \ast wp x Q s) (a \ast wp y Q s) +}
\text{min (b \ast wp x R s) (b \ast wp y R s) } \ominus c
\text{by(simp add:min-distrib)}
\text{also}
\text{have } ...
\text{by(auto intro!:tminus-left-mono)}
\text{also}
\text{have } ...
\text{by(rule min-tminus-distrib)}
\text{also }
\text{from slx sQ sR nna nnb nnc}
\text{have } a \ast wp x Q s + b \ast wp x R s \ominus c \leq
\text{wp x (\lambda s. a \ast Q s + b \ast R s \ominus c) s}
\text{by(blast)}
\text{moreover}
\text{from sly sQ sR nna nnb nnc}
\text{have } a \ast wp y Q s + b \ast wp y R s \ominus c \leq
\text{wp y (\lambda s. a \ast Q s + b \ast R s \ominus c) s}
\text{by(blast)}
\text{ultimately}
\text{have min (a \ast wp x Q s + b \ast wp x R s \ominus c) }
\text{(a \ast wp y Q s + b \ast wp y R s \ominus c) } \leq
\text{min (wp x (\lambda s. a \ast Q s + b \ast R s \ominus c) s) }
\text{(wp y (\lambda s. a \ast Q s + b \ast R s \ominus c) s)}
\text{by(auto)}
finally show \( a \ast \min (\wp x Q s) (\wp y Q s) + \\
\min (\wp x (\lambda s. a \ast Q s + b \ast R s \ominus c) s) \\
(\wp y (\lambda s. a \ast Q s + b \ast R s \ominus c) s) \),
qed

As for continuity, we insist on a finite support.

\textbf{lemma} sublinear-wp-SetPC:
\begin{itemize}
  \item \texttt{fixes p':a \Rightarrow 's prog}
  \item \texttt{assumes slp: \( \forall s. a. a \in \text{supp} (P s) \Longrightarrow \text{sublinear} (wp (p a)) \)}
  \item \texttt{and sum: \( \forall s. (\sum a \in \text{supp} (P s). P s a) \leq 1 \)}
  \item \texttt{and nnP: \( \forall s. a. 0 \leq P s a \)}
  \item \texttt{and fin: \( \forall s. \text{finite} (\text{supp} (P s)) \)}
\end{itemize}
\texttt{shows sublinear (wp (SetPC p P))}
\texttt{proof (rule sublinear1, simp add:wp-eval) fix R::'s \Rightarrow real and Q::'s \Rightarrow real and s::'s and a::real and b::real and c::real assume sl: \( \forall R s a. \text{sound} Q s \ominus R s \ominus a \)}
\texttt{and nna: \( 0 \leq a \) and nnb: \( 0 \leq b \) and nnc: \( 0 \leq c \)}
\texttt{have a \ast (\sum a' \in \text{supp} (P s). P s a' \ast wp (p a') Q s) + \}
\texttt{b \ast (\sum a' \in \text{supp} (P s). P s a' \ast wp (p a') R s) \ominus c = \}
\texttt{(\sum a' \in \text{supp} (P s). P s a' \ast ((a \ast wp (p a') Q s + b \ast wp (p a') R s)) \ominus c) by (simp add:field_simps sum-distrib-left sum_distrib) also have ... \leq \}
\texttt{(\sum a' \in \text{supp} (P s). P s a' \ast ((a \ast wp (p a') Q s + b \ast wp (p a') R s)) \ominus c) by (rule tminus-right-antimono) also 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 (rule sum_distrib_right) also from \texttt{fin} and \texttt{nnc} have \( ... \leq 1 \ast c \) by (rule real_right_mono) finally show \( \sum a' \in \text{supp} (P s). P s a' \ast c \) \leq \( c \) by (simp qed also from \texttt{fin} have \( ... \leq (\sum a' \in \text{supp} (P s). P s a' \ast ((a \ast wp (p a') Q s + b \ast wp (p a') R s) \ominus P s a' \ast c) \) by (blast intro: tminus-sum_mono) also have \( \texttt{...} = (\sum a' \in \text{supp} (P s). P s a' \ast ((a \ast wp (p a') Q s + b \ast wp (p a')) R s \ominus c)) \) by (simp add: nnP tminus_left_distrib) also \{ from slp sQ sR nna nnb nnc have \( \forall a'. a' \in \text{supp} (P s) \Longrightarrow a \ast wp (p a') Q s + b \ast wp (p a') R s \ominus c \leq \}
\texttt{wp (p a') (\lambda s. a \ast Q s + b \ast R s \ominus c) s) by (blast) with \texttt{nnP} have \( \sum a' \in \text{supp} (P s). P s a' \ast ((a \ast wp (p a') Q s + b \ast wp (p a') R s \ominus c)) \leq \}
\texttt{(\sum a' \in \text{supp} (P s). P s a' \ast wp (p a') (\lambda s. a \ast Q s + b \ast R s \ominus c) s) \)
4.5. SUBLINEARITY

by(\(\text{blast intro:sum-mono mult-left-mono}\))

}\)

finally

show \(a \ast (\sum a' \in \text{supp} (P \, s). \; P \, s \; a' \ast \wp (p \, a') \; Q \, s) +\)

\(b \ast (\sum a' \in \text{supp} (P \, s). \; P \, s \; a' \ast \wp (p \, a') \; R \, s) \ominus c \leq\)

\((\sum a' \in \text{supp} (P \, s). \; P \, s \; a' \ast \wp (p \, a') (\lambda s. \; a \ast Q \, s + b \ast R \, s \ominus c) \; s) \).\)

qed

lemma \(\text{sublinear-wp-SetDC}:\)

fixes \(p::'a \Rightarrow 's \text{ prog}\)

assumes \(\text{slp}: \bigwedge s \; a. \; a \in S \; s \Rightarrow \text{ sublinear } (\wp (p \, a))\)

and \(\text{hp}: \bigwedge s \; a. \; a \in S \; s \Rightarrow \text{ healthy } (\wp (p \, a))\)

and \(\text{nc}: \bigwedge s. \; S \; s \neq \{\}\)

shows \(\text{ sublinear } (\wp (\text{SetDC} \; p \; S))\)

proof(rule \text{sublinearI}, \sim{\text{simp add:wp-eval}}, rule \text{cInf-greatest})

fix \(P::'s \Rightarrow \text{ real and } Q::'s \Rightarrow \text{ real and } s::'s \text{ and } x \; y\)

and \(a::\text{real and } b::\text{real and } c::\text{real}\)

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

and \(\text{ nna: } 0 \leq a \text{ and } \text{ nnb: } 0 \leq b \text{ and } \text{ nnc: } 0 \leq c\)

from \(\text{ ne } \text{ show } (\lambda p. \; wp (p \; pr) (\lambda s. \; a \ast P \; s + b \ast Q \; s \ominus c) \; s) \; ' \; S \; s \neq \{\} \text{ by(auto)}\)

assume \(\text{ yin: } y \in (\lambda p. \; wp (p \; pr) (\lambda s. \; a \ast P \; s + b \ast Q \; s \ominus c) \; s) \; ' \; S \; s\)

then obtain \(x \; \text{ where } \text{xin: } x \in S \; s \text{ and } \text{rwy: } y = \wp (p \; x) (\lambda s. \; a \ast P \; s + b \ast Q \; s \ominus c) \; s\)

by(auto)

from \(\text{ xin \; hp \; sP \; nna}\)

have \(a \ast \text{Inf} (\lambda a. \; wp (p \; a) \; P \; s) \; ' \; S \; s \leq a \ast \wp (p \; x) \; P \; s\)

by(intro \text{ mult-left-mono}[OF \text{cInf-lower}] \text{bdd-belowI}[\text{where } m=0], \text{blast+})

moreover from \(\text{xin \; hp \; sQ \; nnb}\)

have \(b \ast \text{Inf} (\lambda a. \; wp (p \; a) \; Q \; s) \; ' \; S \; s \leq b \ast \wp (p \; x) \; Q \; s\)

by(intro \text{ mult-left-mono}[OF \text{cInf-lower}] \text{bdd-belowI}[\text{where } m=0], \text{blast+})

ultimately

have \(a \ast \text{Inf} (\lambda a. \; wp (p \; a) \; P \; s) \; ' \; S \; s + \)

\(b \ast \text{Inf} (\lambda a. \; wp (p \; a) \; Q \; s) \; ' \; S \; s \ominus c \leq\)

\(a \ast \wp (p \; x) \; P \; s + b \ast \wp (p \; x) \; Q \; s \ominus c\)

by(\text{blast intro:minus-left-mono add-mono})

also from \(\text{xin \; slp \; sP \; sQ \; nna \; nnb \; nnc}\)

have \(... \leq \wp (p \; x) (\lambda s. \; a \ast P \; s + b \ast Q \; s \ominus c) \; s\)

by(\text{blast})

finally show \(a \ast \text{Inf} (\lambda a. \; wp (p \; a) \; P \; s) \; ' \; S \; s + b \ast \text{Inf} (\lambda a. \; wp (p \; a) \; Q \; s) \; ' \; S \; s \ominus c \leq y\)

by(\text{simp add:rwy})

qed

lemma \(\text{sublinear-wp-Embed}:\)
sublinear \( t \rightarrow \) sublinear \( (wp \ (Embed \ t)) \)
by(simp add:wp-eval)

**Lemma** sublinear-wp-repeat:
\[
\begin{align*}
\text{by (induct n, simp-all add:sublinear-wp-Seq sublinear-wp-Skip healthy-wp-repeat)}
\end{align*}
\]

**Lemma** sublinear-wp-Bind:
\[
\begin{align*}
\text{by (rule sublinearI, simp add:wp-eval, auto)}
\end{align*}
\]

**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:
\[
\begin{align*}
\text{fixes body::s prog}
\text{assumes sdb: sub-distrib (wp body)}
\text{and hb: healthy (wp body)}
\text{and nhb: nearly-healthy (wp body)}
\text{shows sub-distrib (wp (do G \rightarrow body od))}
\text{proof (rule loop-induct[OF hb nhb], safe)}
\text{fix S::(s trans \times s trans) set and P::s expect and s::s}
\text{assume saS: \forall x\in S. \forall P s. sound P \rightarrow \fst x P s \ominus 1 \leq \fst x (\lambda s. P s \ominus 1) s}
\text{and sP: sound P}
\text{and fS: \forall x\in S. feasible (fst x)}
\text{from sP have sPm: sound (\lambda s. P s \ominus 1) by(auto intro:tminus-sound)}
\text{have \&n Sup: \forall s. 0 \leq \Sup-trans (fst ' S) (\lambda s. P s \ominus 1) s}
\text{proof(cases S={}, simp add:Sup-trans-def Sup-exp-def)}
\text{fix s}
\text{assume S \neq {}}
\text{then obtain x where xin: x\in S by(auto)}
\text{with fS sPm have 0 \leq \fst x (\lambda s. P s \ominus 1) s by(auto)}
\text{also from xin fS sPm have ... \leq \Sup-trans (fst ' S) (\lambda s. P s \ominus 1) s}
\text{by(auto intro!: le-funD[OF Sup-trans-upper2])}
\text{finally show \&thesis s .}
\text{qed}
\end{align*}
\]

**Lemma** subadd-wp-loop:
\[
\begin{align*}
\text{fixes body::s prog}
\text{assumes sdb: sub-distrib (wp body)}
\text{and hb: healthy (wp body)}
\text{and nhb: nearly-healthy (wp body)}
\text{shows subadd (wp (do G \rightarrow body od))}
\text{proof (rule loop-induct[OF hb nhb], safe)}
\text{fix S::(s trans \times s trans) set and P::s expect and s::s}
\text{assume saS: \forall x\in S. \forall P s. sound P \rightarrow \fst x P s \ominus 1 \leq \fst x (\lambda s. P s \ominus 1) s}
\text{and sP: sound P}
\text{and fS: \forall x\in S. feasible (fst x)}
\text{from sP have sPm: sound (\lambda s. P s \ominus 1) by(auto intro:tminus-sound)}
\text{have \&n Sup: \forall s. 0 \leq \Sup-trans (fst ' S) (\lambda s. P s \ominus 1) s}
\text{proof(cases S={}, simp add:Sup-trans-def Sup-exp-def)}
\text{fix s}
\text{assume S \neq {}}
\text{then obtain x where xin: x\in S by(auto)}
\text{with fS sPm have 0 \leq \fst x (\lambda s. P s \ominus 1) s by(auto)}
\text{also from xin fS sPm have ... \leq \Sup-trans (fst ' S) (\lambda s. P s \ominus 1) s}
\text{by(auto intro!: le-funD[OF Sup-trans-upper2])}
\text{finally show \&thesis s .}
\text{qed}
\end{align*}
\]
4.5. SUBLINEARITY

with \( fS \) have \( \forall x. x \in S \Rightarrow \text{fst } (\lambda s. P \preceq 1) s + 1 \leq \text{Sup-trans} \ (\text{fst } \ S) \ (\lambda s. P \preceq 1) s + 1 \)
by (\text{blast intro: add-right-mono le-funD([OF Sup-trans-upper2])})
}
finally have le: \( \forall s. \forall x \in S. \text{fst } x P s \leq \text{Sup-trans} \ (\text{fst } \ S) \ (\lambda s. P \preceq 1) s + 1 \)
by (auto)
moreover from \( \text{nnSup} \) have nn: \( \forall s. 0 \leq \text{Sup-trans} \ (\text{fst } \ S) \ (\lambda s. P \preceq 1) s + 1 \)
by (auto intro: add-nonneg-nonneg)
ultimately have \( \text{leSup} \) Sup-trans (fst ' S) P s \leq \text{Sup-trans} (fst ' S) (\lambda s. P \preceq 1) s + 1
unfolding Sup-trans-def
by (intro le-funD([OF Sup-exp-least], auto)
show Sup-trans (fst ' S) P s \preceq 1 \leq \text{Sup-trans} (fst ' S) (\lambda s. P \preceq 1) s
proof (cases Sup-trans (fst ' S) P s \leq 1, simp-all add:nnSup)
from \( \text{leSup} \) have Sup-trans (fst ' S) P s - 1 \leq \text{Sup-trans} (fst ' S) (\lambda s. P \preceq 1) s + 1 - 1
by (auto)
thus Sup-trans (fst ' S) P s - 1 \leq \text{Sup-trans} (fst ' S) (\lambda s. P \preceq 1) s
by (simp)
qed
next
fix (\_')s trans and P::'s expect and s::'s
assume IH: \( \forall P. s. \text{sound } P \Rightarrow t \ P s \preceq 1 \leq t (\lambda a. P a \preceq 1) s \)
and ft: feasible t
and sP: sound P
from sP ft have sound (\lambda s. P s \preceq 1) by (auto intro: tminus-sound)
with ft have s2: sound (t (\lambda s. P s \preceq 1)) by (auto)
from sP ft have sound (t P) by (auto)
hence s3: sound (\lambda s. t P s \preceq 1) by (auto intro!: tminus-sound)
show wp (body :: Embed t \ a G \oplus Skip) P s \preceq 1 \leq wp (body :: Embed t \ a G \oplus Skip) (\lambda a. P a \preceq 1) s
proof (simp add: wp-eval)
have \( \forall s. wP \text{ body } (\lambda P. t P) s \oplus (1 - \forall G s) \ast P s \preceq 1 = \)
\( \forall G s \ast wP \text{ body } (\lambda P. t P) s + (1 - \forall G s) \ast P s \ominus (\forall G s + (1 - \forall G s)) \)
by (simp)
also have \( \ldots \leq (\forall G s \ast (wP \text{ body } (\lambda P. t P) s \ominus \forall G s) + \)
\( (1 - \forall G s) \ast P s \ominus (1 - \forall G s)) \)
by (rule tminus-add-mono)
also have \( \ldots = (\forall G s \ast (wP \text{ body } (\lambda P. t P) s \ominus 1) + (1 - \forall G s) \ast (P s \preceq 1) \)
by (simp add: tminus-left-distrib)
also { from ft sP have wp body (\lambda P. t P s \preceq 1) \leq wp body (\lambda s. t P s \preceq 1) s
by (auto intro!: sub-distribD([OF sdb]))
also { from IH sP have \lambda s. t P s \preceq 1 \vdash t (\lambda s. P s \preceq 1) by (auto)
with \( sP \) \( \text{ft} \) \( s_2 \) \( s_3 \) have \( \text{wp body} (\lambda s. t P s \odot 1) s \leq \text{wp body} (t (\lambda s. P s \odot 1)) s \)

by\((\text{blast intro:le-funD}[\text{OF mono-transD}, \text{OF healthy-monoD}, \text{OF hh}])\)

\}

finally have \( G s \ast (\text{wp body} (t P s \odot 1) + (1 \ast \langle G \rangle s) \ast (P s \odot 1)) \leq \langle G \rangle s \ast \text{wp body} (t (\lambda s. P s \odot 1)) s + (1 \ast \langle G \rangle s) \ast (P s \odot 1) \)

by\((\text{auto intro:add-right-mono mult-left-mono})\)

\}

finally show \( G s \ast \text{wp body} (t P s \odot 1) \leq \langle G \rangle s \ast \text{wp body} (t (\lambda s. P s \odot 1)) s + (1 \ast \langle G \rangle s) \ast (P s \odot 1) \)

qed

next

fix \( t \) \( t' \) ::'s trans and \( P ::'s \) expect and \( s ::'s \)

assume \( IH :: \forall P s. \text{sound} P \rightarrow t P s \odot 1 \leq t (\lambda a. P a \odot 1) s \)

and \( eq :: \text{equiv-trans} t t' \) and \( sP :: \text{sound} P \)

from \( sP \) have \( t' P s \odot 1 = t P s \odot 1 \) by\((\text{simp add:eqv-transD}[\text{OF eq}])\)

also from \( sP \) \( IH \) have \( \ldots \leq t (\lambda s. P s \odot 1) s \) by\((\text{auto})\)

also { from \( sP \) have \( \text{sound} (\lambda s. P s \odot 1) \) by\((\text{simp add:tminus-sound})\)

hence \( t (\lambda s. P s \odot 1) s = t' (\lambda s. P s \odot 1) s \) by\((\text{simp add:equiv-transD}[\text{OF eq}])\)

}\}

finally show \( t' P s \odot 1 \leq t' (\lambda s. P s \odot 1) s \)

qed

thus \( \? \)thesis by\((\text{auto intro!:sub-distribI})\)

qed

For sub-additivity, we again use the limit-of-iterates characterisation. Firstly, all iterates are sublinear:

**Lemma** sublinear-iterates:

assumes \( hh :: \text{healthy} (\text{wp body}) \)

and \( sb :: \text{sublinear} (\text{wp body}) \)

shows \( \text{sublinear} (\text{iterates body} G i) \)

by\((\text{induct } i, \text{auto intro!:sublinear-wp-PC sublinear-wp-Seq sublinear-wp-Skip sublinear-wp-Embed asms healthy-intros iterates-healthy})\)

From this, sub-additivity follows for the limit (i.e. the loop), by appealing to the property at all steps.

**Lemma** sub-add-wp-loop:

fixes \( body ::'s \) prog

assumes \( sb :: \text{sublinear} (\text{wp body}) \)

and \( cb :: \text{bd-cts} (\text{wp body}) \)

and \( hwp :: \text{healthy} (\text{wp body}) \)

shows \( \text{sub-add} (\text{wp} (\text{do } G \rightarrow \text{body} od)) \)

**Proof**

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

assume \( sP :: \text{sound} P \) and \( sQ :: \text{sound} Q \)
4.5. SUBLINEARITY

from hwp cb sP have \((\lambda i. \text{iterates body } G_i P_s) \rightarrow wp \text{ do } G \rightarrow \text{ body od } P_s\)
by (rule loop-iterates)
moreover
from hwp cb sQ have \((\lambda i. \text{iterates body } G_i Q_s) \rightarrow wp \text{ do } G \rightarrow \text{ body od } Q_s\)
by (rule loop-iterates)
ultimately
have \((\lambda i. \text{iterates body } G_i P_s + \text{ iterates body } G_i Q_s) \rightarrow wp \text{ do } G \rightarrow \text{ body od } P_s + wp \text{ do } G \rightarrow \text{ body od } Q_s\)
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 \(\bigwedge_i \text{iterates body } G_i P_s + \text{ iterates body } G_i Q_s \leq \text{iterates body } G_i (\lambda s. P_s + Q_s)\)
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 \((\lambda i. \text{iterates body } G_i (\lambda s. P_s + Q_s) s) \rightarrow wp \text{ do } G \rightarrow \text{ body od } (\lambda s. P_s + Q_s) s\)
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
4.6 Determinism

theory Determinism imports WellDefined begin

We provide a set of lemmas for establishing that appropriately restricted programs are fully additive, and maximal in the refinement order. This is particularly useful with data refinement, as it implies correspondence.

4.6.1 Additivity

lemma additive-wp-Abort: additive (wp (Abort))
  by(auto simp: wp-eval)

wlp Abort is not additive.

lemma additive-wp-Skip: additive (wp (Skip))
  by(auto simp: wp-eval)

lemma additive-wp-Apply: additive (wp (Apply f))
  by(auto simp: wp-eval)

lemma additive-wp-Seq:
  fixes a::'s prog
  assumes adda: additive (wp a)
    and addb: additive (wp b)
    and wb: well-def b
  shows additive (wp (a ;; b))
proof(rule additiveI, unfold wp-eval o-def)
  fix P::'s ⇒ real and Q::'s ⇒ real and s::'s
  assume sP: sound P and sQ: sound Q

  note hb = well-def-wp-healthy[OF wb]
  from addb sP sQ
  have wp b (λs. P s + Q s) = (λs. wp b P s + wp b Q s)
    by(blast dest:additiveD)
  with adda sP sQ hb
4.6. DETERMINISM

show \( wp \ a \ (wp \ b \ (\lambda s. P \ s + Q \ s)) \ s = \)
\[ wp \ a \ (wp \ b \ P) \ s + (wp \ a \ (wp \ b \ Q)) \ s \]
by(auto intro:fun-cong[OF additiveD])

qed

lemma additive-wp-PC:
[ additive \( wp \ a \); additive \( wp \ b \) ] \( \Rightarrow \) additive \( wp \ (a \ p\oplus b) \)
by(rule additiveI, simp add: additiveD field-simps wp-eval)

DC is not additive.

lemma additive-wp-SetPC:
[ \( \forall x \ s. \ x \in supp \ (p \ s) \Rightarrow \) additive \( wp \ (a \ x) \); \( \forall s. finite \ (supp \ (p \ s)) \] \( \Rightarrow \)
additive \( wp \ (SetPC \ a \ p) \)
by(rule additiveI, simp add: wp-eval additiveD distrib-left sum.distrib)

lemma additive-wp-Bind:
[ \( \forall x. \) additive \( wp \ (a \ (f \ x)) \] \( \Rightarrow \) additive \( wp \ (Bind \ f \ a) \)
by(simp add:wp-eval additive-def)

lemma additive-wp-Embed:
[ \( additive \ t \] \( \Rightarrow \) additive \( wp \ (Embed \ t) \)
by(simp add:wp-eval)

lemma additive-wp-repeat:
\( additive \ (wp \ a) \Rightarrow \) well-def \ a \( \Rightarrow \) additive \( wp \ (repeat \ n \ a) \)
by(induct n, auto simp: additive-wp-Skip intro: additive-wp-Seq wd-intros)

lemmas fa-intros =
additive-wp-Abort additive-wp-Skip
additive-wp-Apply additive-wp-Seq
additive-wp-PC additive-wp-SetPC
additive-wp-Bind additive-wp-Embed
additive-wp-repeat

4.6.2 Maximality

lemma max-wp-Skip:
maximal \( wp \ Skip \)
by(simp add:maximal-def wp-eval)

lemma max-wp-Apply:
maximal \( wp \ (Apply \ f) \)
by(auto simp:wp-eval o-def)

lemma max-wp-Seq:
[ \( maximal \ (wp \ a); maximal \ (wp \ b) \] \( \Rightarrow \) maximal \( wp \ (a ;; b) \)
by(simp add:wp-eval maximal-def)
lemma max-wp-PC: \[
\maximal\ (\wp a); \maximal\ (\wp b) \implies \maximal\ (\wp (a \oplus b))
\]
 by (rule maximalI, simp add: maximalD field-simps wp-eval)

lemma max-wp-DC: \[
\maximal\ (\wp a); \maximal\ (\wp b) \implies \maximal\ (\wp (a \sqcap b))
\]
 by (rule maximalI, simp add: wp-eval maximalD)

lemma max-wp-SetPC: \[
\bigwedge s. a \in \supp (P s) = \implies \maximal\ (\wp (p a)); \\
\bigwedge s. (\sum a \in \supp (P s). P s a) = 1
\]
 by (auto simp: maximalD wp-def SetPC-def sum-distrib-right [symmetric])

lemma max-wp-SetDC: fixes p::'a \Rightarrow 's prog
 assumes mp: \bigwedge s. a \in S s = \implies \maximal\ (\wp (p a))
 and ne: \bigwedge s. S s \neq \{\}
 shows \maximal\ (\wp (SetDC p S))
proof (rule maximalI, rule ext, unfold wp-eval)
 fix c::real and s::'s
 assume 0 \leq c
 hence \Inf ((\lambda a. \wp (p a) (\lambda-. c) s) ' S s) = \Inf ((\lambda-. c) ' S s)
 using mp by (simp add: maximalD cong:image-cong)
 also {
 from ne obtain a where a \in S s by blast
 hence \Inf ((\lambda-. c) ' S s) = c
 by (auto simp add: image-constant-cong cong del: INF-cong-simp)
}
finally show \Inf ((\lambda a. \wp (p a) (\lambda-. c) s) ' S s) = c .
qed

lemma max-wp-Embed: \maximal t = \implies \maximal\ (\wp (Embed t))
 by (simp add: wp-eval)

lemma max-wp-repeat: \maximal\ (\wp a) = \implies \maximal\ (\wp (repeat n a))
 by (induct n, simp-all add: max-wp-Skip max-wp-Seq)

lemma max-wp-Bind: assumes ma: \bigwedge s. \maximal\ (\wp (a (f s)))
 shows \maximal\ (\wp (Bind f a))
proof (rule maximalI, rule ext, simp add: wp-eval)
 fix c::real and s
 assume 0 \leq c
 with ma have \wp (a (f s)) (\lambda-. c) = (\lambda-. c) by (blast)
 thus \wp (a (f s)) (\lambda-. c) s = c by (auto)
qed
4.6. DETERMINISM

lemmas max-intros =
max-wp-Skip  max-wp-Apply
max-wp-Seq  max-wp-PC
max-wp-DC  max-wp-SetPC
max-wp-SetDC max-wp-Embed
max-wp-Bind  max-wp-repeat

A healthy transformer that terminates is maximal.

lemma healthy-term-max:
  assumes ht: healthy t
          and trm: λs. 1 ⊢ t (λs. 1)
  shows maximal t
proof (intro maximalI ext)
  fix c::real and s
  assume nnc: 0 ≤ c

  have t (λs. c) s = t (λs. 1 * c) s by(simp)
  also from nnc healthy-scalingD[OF ht]
  have ... = c * t (λs. 1) s by(simp add:scalingD)
  also { from ht have t (λs. 1) ⊢ λs. 1 by(auto)
          with trm have t (λs. 1) = (λs. 1) by(auto)
          hence c * t (λs. 1) s = c by(simp)
    }
  finally show t (λs. c) s = c .
qed

4.6.3 Determinism

lemma det-wp-Skip:
  determ (wp Skip)
  using max-intros fa-intros by(blast)

lemma det-wp-Apply:
  determ (wp (Apply f))
  by(intro determI fa-intros max-intros)

lemma det-wp-Seq:
  determ (wp a) ⇒ determ (wp b) ⇒ well-def b ⇒ determ (wp (a ;; b))
  by(intro determI fa-intros max-intros, auto)

lemma det-wp-PC:
  determ (wp a) ⇒ determ (wp b) ⇒ determ (wp (a p⊕ b))
  by(intro determI fa-intros max-intros, auto)

lemma det-wp-SetPC:
  (Λx s. x ∈ supp (p s) ⇒ determ (wp (a x))) ⇒
  (Λs. finite (supp (p s))) ⇒
CHAPTER 4. THE PGCL LANGUAGE

\[
(\forall s. \text{sum}(p s) \text{supp}(p s)) = 1 \implies \\
\text{determ}(wp(\text{SetPC} \ a \ p)) \by\text{(intro determI fa-intros max-intros, auto)}
\]

\[\text{lemma det-wp-Bind:} \]
\[
(\forall x. \text{determ}(wp(a(f x)))) \implies \text{determ}(wp(\text{Bind} f a)) \]
\[\by\text{(intro determI fa-intros max-intros, auto)}
\]

\[\text{lemma det-wp-Embed:} \]
\[
\text{determ} t \implies \text{determ}(wp(\text{Embed} t)) \by\text{(simp add:wp-eval)}
\]

\[\text{lemma det-wp-repeat:} \]
\[
\text{determ}(wp a) \implies \text{well-def} a \implies \text{determ}(wp(\text{repeat} n \ a)) \]
\[\by\text{(intro determI fa-intros max-intros, auto)}
\]

\[\text{lemmas determ-intros =} \]
\[
\text{det-wp-Skip det-wp-Apply det-wp-Seq det-wp-PC det-wp-SetPC det-wp-Bind det-wp-Embed det-wp-repeat}
\]

\[\end\]

4.7 Well-Defined Programs.

\[\text{theory WellDefined imports} \]
\[
\text{Healthiness} \quad \text{Sublinearity} \quad \text{LoopInduction}
\]
\[\begin{aligned}
\text{begin} & \\
\text{The definition of a well-defined program collects the various notions of healthiness and well-behavedness that we have so far established: healthiness of the strict and liberal transformers, continuity and sublinearity of the strict transformers, and two new properties. These are that the strict transformer always lies below the liberal one (i.e. that it is at least as strict, recalling the standard embedding of a predicate), and that expectation conjunction is distributed between then in a particular manner, which will be crucial in establishing the loop rules.}
\end{aligned}
\]

4.7.1 Strict Implies Liberal

This establishes the first connection between the strict and liberal interpretations (wp and wlp).

\[\text{definition} \]
\[
\text{wp-under-wlp ::} \ 's \ prog \Rightarrow \ bool
\]
4.7. WELL-DEFINED PROGRAMS.

where
\[ \text{wp-under-wlp prog} \equiv \forall P. \text{unitary } P \rightarrow \text{wp prog } P \vdash \text{wlp prog } P \]

**lemma** \text{wp-under-wlpI}[intro]:
\[
[ \forall P. \text{unitary } P \rightarrow \text{wp prog } P \vdash \text{wlp prog } P ] \rightarrow \text{wp-under-wlp prog}
\]

**unfolding** \text{wp-under-wlp-def} **by**(simp)

**lemma** \text{wp-under-wlpD}[dest]:
\[
[ \text{wp-under-wlp prog}; \text{unitary } P ] \rightarrow \text{wp prog } P \vdash \text{wlp prog } P
\]

**unfolding** \text{wp-under-wlp-def} **by**(simp)

**lemma** \text{wp-under-le-trans}:
\[
\text{wp-under-wlp } a \rightarrow \text{le-utrans } (\text{wp } a) (\text{wlp } a)
\]

**by**(blast)

**lemma** \text{wp-under-wlp-Abort}:
\[
\text{wp-under-wlp } \text{Abort}
\]

**by**(rule \text{wp-under-wlpI}, unfold \text{wp-eval}, auto)

**lemma** \text{wp-under-wlp-Skip}:
\[
\text{wp-under-wlp } \text{Skip}
\]

**by**(rule \text{wp-under-wlpI}, unfold \text{wp-eval}, blast)

**lemma** \text{wp-under-wlp-Apply}:
\[
\text{wp-under-wlp } (\text{Apply } f)
\]

**by**(auto simp:wp-eval)

**lemma** \text{wp-under-wlp-Seq}:
\[
\text{assumes } h\text{-wp-a}: \text{nearly-healthy } (\text{wlp } a)\quad \text{and } h\text{-wp-b}: \text{healthy } (\text{wp } b)\quad \text{and } h\text{-wp-l-p-b}: \text{nearly-healthy } (\text{wlp } b)\quad \text{and } wp\text{-a-a}: \text{wp-under-wlp } a\quad \text{and } wp\text{-a-b}: \text{wp-under-wlp } b\quad \text{shows } \text{wp-under-wlp } (a ;; b)
\]

**proof**(rule \text{wp-under-wlpI}, unfold \text{wp-eval a-def})

**fix** \(P:: a \Rightarrow \text{real assume } uP:: \text{unitary } P\)

**with** \text{h-wp-b} **have** \text{unitary } (wp b P) **by**(blast)

**with** \text{wp-u-a} **have** \text{wp a } (wp b P) \vdash \text{wlp a } (wp b P) **by**(auto)

**also** \{

**from** \text{wp-u-a b} **and** uP **have** \text{wp b P \vdash wlp b P} **by**(blast)

**with** \text{h-wlp-a} **and** h-wlp-b **and** h-wp-b **and** uP

**have** \text{wp a } (wp b P) \n\n**by**(blast intro:nearly-healthy-monoD[OF h-wlp-a])

**\}

**finally** **show** \text{wp a } (wp b P) \vdash \text{wlp a } (wp b P).

**qed**

**lemma** \text{wp-under-wlp-PC}:
\[
\text{assumes } h\text{-wp-a}: \text{healthy } (\text{wp } a)
\]
and \(h\)-\(wp\)-\(a\): nearly-healthy (\(wlp\) \(a\))
and \(h\)-\(wp\)-\(b\): healthy (\(wp\) \(b\))
and \(h\)-\(wp\)-\(h\): nearly-healthy (\(wlp\) \(b\))
and \(wp\)-\(u\)-\(a\): \(wp\)-under-\(wp\) \(a\)
and \(wp\)-\(u\)-\(b\): \(wp\)-under-\(wp\) \(b\)
and \(uP\): unitary \(P\)
shows \(wp\)-under-\(wp\) (\(a \oplus b\))

**proof** (rule \(wp\)-under-\(wp\)-I, unfold \(wp\)-eval, rule le-funI)

fix \(Q:\'a \Rightarrow \) real and \(s\)
assume \(uQ\): unitary \(Q\)
from \(uP\) have \(P \ s \leq 1\) by (blast)
hence \(0 \leq 1 - P \ s\) by (simp)
moreover
from \(uQ\) and \(wp\)-\(u\)-\(b\) have \(wp\) \(b\) \(Q\) \(s\) \(\leq\) \(wlp\) \(b\) \(Q\) \(s\) by (blast)
ultimately
have \((1 - P \ s) \ast \ wp\) \(b\) \(Q\) \(s\) \(\leq\) \((1 - P \ s) \ast \ wlp\) \(b\) \(Q\) \(s\)
by (blast intro: mult-\(\\ast\)-mono)

moreover {
  from \(uQ\) and \(wp\)-\(u\)-\(a\) have \(wp\) \(a\) \(Q\) \(s\) \(\leq\) \(wlp\) \(a\) \(Q\) \(s\) by (blast)
  with \(uP\) have \((P \ s) \ast \ wp\) \(a\) \(Q\) \(s\) \(\leq\) \((P \ s) \ast \ wlp\) \(a\) \(Q\) \(s\)
  by (blast intro: mult-\(\\ast\)-mono)
}

ultimately
show \((P \ s) \ast \ wp\) \(a\) \(Q\) \(s\) + \((1 - P \ s) \ast \ wp\) \(b\) \(Q\) \(s\) \(\leq\)
\((P \ s) \ast \ wlp\) \(a\) \(Q\) \(s\) + \((1 - P \ s) \ast \ wlp\) \(b\) \(Q\) \(s\)
  by (blast intro: add-\(\ast\)-mono)
qed

**lemma** \(wp\)-under-\(wp\)-\(DC\):
assumes \(wp\)-\(u\)-\(a\): \(wp\)-under-\(wp\) \(a\)
and \(wp\)-\(u\)-\(b\): \(wp\)-under-\(wp\) \(b\)
shows \(wp\)-under-\(wp\) (\(a \sqcap b\))

**proof** (rule \(wp\)-under-\(wp\)-I, unfold \(wp\)-eval, rule le-funI)

fix \(Q:\'a \Rightarrow \) real and \(s\)
assume \(uQ\): unitary \(Q\)
from \(wp\)-\(u\)-\(a\) \(uQ\) have \(wp\) \(a\) \(Q\) \(s\) \(\leq\) \(wlp\) \(a\) \(Q\) \(s\) by (blast)
moreover
from \(wp\)-\(u\)-\(b\) \(uQ\) have \(wp\) \(b\) \(Q\) \(s\) \(\leq\) \(wlp\) \(b\) \(Q\) \(s\) by (blast)
ultimately
show \(\min\) (\(wp\) \(a\) \(Q\) \(s\) ) (\(wp\) \(b\) \(Q\) \(s\) ) \(\leq\) \(\min\) (\(wlp\) \(a\) \(Q\) \(s\) ) (\(wlp\) \(b\) \(Q\) \(s\) )
by (auto)
qed

**lemma** \(wp\)-under-\(wp\)-\(SetPC\):
assumes \(wp\)-\(u\)-\(f\): \(\bigwedge a \ a \in \supp\ (P \ s) \Rightarrow \ wp\)-under-\(wp\) (\(f\) \(a\))
and \(nP\): \(\bigwedge a \ a \in \supp\ (P \ s) \Rightarrow 0 \leq P \ s \ a\)
4.7. WELL-DEFINED PROGRAMS.

shows \text{wp-under-wlp} \ (\text{SetPC \ f \ P})
proof (rule \text{wp-under-wlpI}, \ unfold \ \text{wp-eval}, \ rule \ \text{le-funI})

fix Q::'a ⇒ real and s
assume uQ: unitary Q

from \text{wp-u-f} uQ nP
show \left( \sum_{a \in \text{supp} \ (P \ s)} \ P \ s \ a * \text{wp} (f \ a) \ Q \ s \right) ≤ \left( \sum_{a \in \text{supp} \ (P \ s)} \ P \ s \ a * \text{wlp} (f \ a) \ Q \ s \right)
by (auto intro!:\text{sum-mono} \ \text{mult-left-mono})

qed

lemma \text{wp-under-wlp-SetDC}:
assumes \text{wp-u-f}:
\forall s \ a. a ∈ S \ s ⇒ \text{wp-under-wlp} (f \ a)
and hf:
\forall s \ a. a ∈ S \ s ⇒ \text{healthy} (\text{wp} (f \ a))
and nS:
\forall s. S \ s \neq \{\}
shows \text{wp-under-wlp} \ (\text{SetDC \ f \ S})
proof (rule \text{wp-under-wlpI}, \ \text{rule \ le-funI}, \ unfold \ \text{wp-eval})
fix Q::'a ⇒ real and s
assume uQ: unitary Q

show Inf ((\lambda a. \text{wp} (f \ a)) \ Q \ s) ' S s) ≤ Inf ((\lambda a. \text{wlp} (f \ a)) \ Q \ s) ' S s)
proof (rule \text{cInf-mono})
from nS show (\lambda a. \text{wlp} (f \ a) \ Q \ s) ' S s \neq \{\} \ by (\text{blast})

fix x assume xin: x ∈ (\lambda a. \text{wp} (f \ a) \ Q \ s) ' S s
then obtain a where ain: a ∈ S \ s and xrw: x = \text{wlp} (f \ a) \ Q \ s
by (blast)
with \text{wp-u-f} uQ
have \text{wp} (f \ a) \ Q \ s ≤ \text{wlp} (f \ a) \ Q \ s \ by (\text{blast})
moreover from ain have \text{wp} (f \ a) \ Q \ s ∈ (\lambda a. \text{wp} (f \ a) \ Q \ s) ' S s
by (blast)
ultimately show ∃ y∈ (\lambda a. \text{wp} (f \ a) \ Q \ s) ' S s. y ≤ x
by (auto simp:\text{xrw})

next
fix y assume yin: y ∈ (\lambda a. \text{wp} (f \ a) \ Q \ s) ' S s
then obtain a where ain: a ∈ S \ s and yrw: y = \text{wp} (f \ a) \ Q \ s
by (blast)
with hf uQ have unitary (\text{wp} (f \ a) \ Q) \ by (auto)
with yrw show 0 ≤ y \ by (auto)
qed

qed

lemma \text{wp-under-wlp-Embed}:
\text{wp-under-wlp} \ (\text{Embed \ t})
by (rule \text{wp-under-wlpI}, \ unfold \ \text{wp-eval}, \ \text{blast})

lemma \text{wp-under-wlp-loop}:
fixes body::'s prog
assumes \textit{hwp}: healthy (wp body) 
and \textit{hwlp}: nearly-healthy (wlp body) 
and \textit{wp-under}: wp-under-wlp body
shows \textit{wp-under-wlp} (do \textit{G} \rightarrow body od)

\textbf{proof} (rule \textit{wp-under-wlpI})

fix \(P::s\) expect
assume \(uP\): unitary \(P\) hence \(sP\): sound \(P\) by (auto)

\begin{align*}
\text{let } & ?X Q s = «G» s \ast wp body Q s + «N G» s \ast P s \\
\text{let } & ?Y Q s = «G» s \ast wlp body Q s + «N G» s \ast P s
\end{align*}

\begin{align*}
\text{show } & (do \textit{G} \rightarrow body od) P \vdash wlp (do \textit{G} \rightarrow body od) P \\
\text{proof} & (\text{simp add:} \text{hwp hwlp } sP \ uP \ \text{wp-Loop1 } wlp-Loop1, \text{ rule gfp-exp-upperbound})
\end{align*}

\begin{align*}
\text{thm } & \text{lpf-loop-fp} \\
\text{from } & \text{hwp } sP \ \text{have } \text{lpf-exp } ?X = ?X (\text{lpf-exp } ?X) \\
\text{by} & (\text{rule lfp-wp-loop-unfold})
\end{align*}

\begin{align*}
\text{hence } & \text{lpf-exp } ?X \vdash ?X (\text{lpf-exp } ?X) \text{ by (simp)} \\
\text{also } & \{
\text{from } \text{hwp } uP \ \text{have } \text{wp body} (\text{lpf-exp } ?X) \vdash \text{wp body} (\text{lpf-exp } ?X) \\
\text{by} & (\text{auto intro:wp-under-wlpD[OF wp-under]} \ \text{lpf-loop-unitary})
\end{align*}

\begin{align*}
\text{hence } & ?X (\text{lpf-exp } ?X) \vdash ?Y (\text{lpf-exp } ?X) \\
\text{by} & (\text{auto intro:} \text{add-mono mult-left-mono})
\end{align*}

\begin{align*}
\text{finally show } & \text{lpf-exp } ?X \vdash ?Y (\text{lpf-exp } ?X), \\
\text{from } & \text{hwp } uP \ \text{show } \text{unitary} (\text{lpf-exp } ?X) \\
\text{by} & (\text{auto intro:} \text{lpf-loop-unitary})
\end{align*}

\textbf{qed}

\textbf{lemma} \textit{wp-under-wlp-repeat}:
\[
\begin{align*}
& [ \text{healthy } (wp a); \text{nearly-healthy } (wlp a); \text{wp-under-wlp } a ] \implies \text{wp-under-wlp } (\text{repeat } n a) \\
& \text{by (induct } n, \text{ auto intro:wp-under-wlp-Skip wp-under-wlp-Seq healthy-intros)}
\end{align*}
\]

\textbf{lemma} \textit{wp-under-wlp-Bind}:
\[
\begin{align*}
& [ \lambda s. \text{wp-under-wlp } (a (f s)) ] \implies \text{wp-under-wlp } (\text{Bind } f a) \\
& \text{unfolding wp-under-wlp-def by (auto simp: wp-eval)}
\end{align*}
\]


\section*{4.7.2 Sub-Distributivity of Conjunction}

\textbf{definition}
sub-distrib-pconj :: 's prog ⇒ bool

where
sub-distrib-pconj prog ≡
∀ P Q. unitary P → unitary Q →
wlp prog P & & wp prog Q ⊢ wp prog (P & & Q)

lemma sub-distrib-pconjI[intro]:
[ ∃ P Q. [ unitary P; unitary Q ] → wp prog P & & wp prog Q ⊢ wp prog (P & & Q) ]
sub-distrib-pconj prog

unfolding sub-distrib-pconj-def by(simp)

lemma sub-distrib-pconjD[dest]:
[ ∀ P Q. [ unitary P; unitary Q ] ] →
wlp prog P & & wp prog Q ⊢ wp prog (P & & Q)
unfolding sub-distrib-pconj-def by(simp)

lemma sdp-Abort:
sub-distrib-pconj Abort
by(rule sub-distrib-pconjI; unfold wp-eval, auto intro:exp-conj-rzero)

lemma sdp-Skip:
sub-distrib-pconj Skip
by(rule sub-distrib-pconjI; simp add:wp-eval)

lemma sdp-Seq:
fixes a and b
assumes sdp-a: sub-distrib-pconj a
and sdp-b: sub-distrib-pconj b
and h-wp-a: healthy (wp a)
and h-wp-b: healthy (wp b)
and h-wlp-b: nearly-healthy (wlp b)
shows sub-distrib-pconj (a ;; b)

proof(rule sub-distrib-pconjI, unfold wp-eval o-def)
fix P::'a ⇒ real and Q::'a ⇒ real
assume uP: unitary P and uQ: unitary Q

with h-wp-b and h-wlp-b
have 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)) .
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-wlp-b: nearly-healthy (wlp b)
shows sub-distrib-pconj (a && b)
proof (rule sub-distrib-pconjI,
unfold wp-eval, rule le-funI)
fix P::'s ⇒ real and Q::'s ⇒ real and s::'
s
assume uP: unitary P and uQ: unitary Q
have ((λs. min (wlp a P s) (wlp b P s)) &&
(λs. min (wp a Q s) (wp b Q s))) s ≤
min (wlp a P s & wp a Q s) (wlp b P s & wp b Q s)
unfolding exp-conj-def by (rule min-pconj)
also { have (λs. wlp a P s & wp a Q s) = wlp a P & wp a Q
by (simp add: exp-conj-def)
also from sdp-a uP uQ have ... ⊢ wp a (P & & Q)
by (blast dest: sub-distrib-pconjD)
finally have wlp a P s & wp a Q s ≤ wp a (P & & Q) s
by (rule le-funD)
moreover { have (λs. wlp b P s & wp b Q s) = wlp b P & wp b Q
by (simp add: exp-conj-def)
also from sdp-b uP uQ have ... ⊢ wp b (P & & Q)
by (blast)
finally have wlp b P s & wp b Q s ≤ wp b (P & & Q) s
by (rule le-funD)
}
ultimately
have min (wlp a P s & wp a Q s) (wlp b P s & wp b Q s) ≤
min (wp a (P & & Q) s) (wp b (P & & Q) s) by (auto)
}
finally
show ((λs. min (wlp a P s) (wlp b P s)) &&
(λs. min (wp a Q s) (wp b Q s))) s ≤
min (wp a (P & & Q) s) (wp b (P & & Q) s).
qed

lemma sdp-PC:
fixes a::'s prog and b
4.7. WELL-DEFINED PROGRAMS.

assumes \( sdp-a: \) sub-distrib-pconj \( a \)
and \( sdp-b: \) sub-distrib-pconj \( b \)
and \( h-wp-a: \) healthy \( (wp a) \)
and \( h-wp-b: \) healthy \( (wp b) \)
and \( h-wlp-b: \) nearly-healthy \( (wlp b) \)
and \( uP: \) unitary \( P \)
shows \( sub-distrib-pconj \) \( (a \ p \oplus \ b) \)
proof (\( rule \ sub-distrib-pconjI, \) unfold \( wp-eval, \) rule \( le-funI) \)

fix \( Q::s \Rightarrow real \) and \( R::s \Rightarrow real \) and \( s::s' \)
assume \( uQ: \) unitary \( Q \) and \( uR: \) unitary \( R \)

have \( nnA: \) \( 0 \leq P s \) and \( nnB: \) \( P s \leq 1 \)
using \( uP \) by auto

note \( nn = nnA \) \( nnB \)

have \( ((\lambda \mathbf{s}. \; P s * wp a Q s + (1 - P s) * wp b Q s) \land\land\
(\lambda \mathbf{s}. \; P s * wp a R s + (1 - P s) * wp b R s)) s =\)
\( ((P s * wp a Q s + (1 - P s) * wp b Q s) +\)
\( (P s * wp a R s + (1 - P s) * wp b R s)) \circ 1 \)
by (simp add: exp-conj-def pconj-def)
also have \( \ldots = P s * \)
\( (wp a Q s + wp a R s) +\)
\( (1 - P s) * (wp b Q s + wp b R s) \circ 1 \)
by (simp add: field-simps)
also have \( \ldots = P s * \)
\( (wp a Q s + wp a R s) +\)
\( (1 - P s) * (wp b Q s + wp b R s) \circ (P s + (1 - P s)) \)
by (simp)
also have \( \ldots \leq (P s * (wp a Q s + wp a R s) \circ P s) +\)
\( (1 - P s) * (wp b Q s + wp b R s) \circ (1 - P s)) \)
by (rule tminus-add-mono)
also have \( \ldots = (P s * (wp a Q s + wp a R s \circ 1)) +\)
\( (1 - P s) * (wp b Q s + wp b R s \circ 1)) \)
by (simp add: nn tminus-left-distrib)
also have \( \ldots = P s * \)
\( ((wp a Q \land\land wp a R) s) +\)
\( (1 - P s) * ((wp b Q \land\land wp b R) s) \)
by (simp add: exp-conj-def pconj-def)
also {\{

from \( sdp-a \) \( sdp-b \) \( uQ \) \( uR \)
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 (simp-all add: entailsD mult-left-mono nn sub-distrib-pconjD)

hence \( P s * \)
\( ((wp a Q \land\land wp a R) s) +\)
\( (1 - P s) * ((wp b Q \land\land wp b R) s) \leq\)
\( P s * wp a (Q \land\land R) s + (1 - P s) * wp b (Q \land\land R) s \)
by (auto)
}\}
finally show \( ((\lambda \mathbf{s}. \; P s * wp a Q s + (1 - P s) * wp b Q s) \land\land\
(\lambda \mathbf{s}. \; P s * wp a R s + (1 - P s) * wp b R s)) s \leq\)
\( P s * wp a (Q \land\land R) s + (1 - P s) * wp b (Q \land\land R) s \).
qed

lemma sdp-Embed:
\[ \forall P, Q. \, [\text{unitary } P; \text{unitary } Q] \Rightarrow t \, P \land t \, Q \vdash t \, (P \land Q) \]
sub-distrib-pconj (Embed t)
by(auto simp:wp-eval)

lemma sdp-repeat:
fixes a :: ’s prog
assumes sdpa: sub-distrib-pconj a
and hwp: healthy (wp a) and hwlp: nearly-healthy (wlp a)
shows sub-distrib-pconj (repeat n a) (is ?X n)
proof (induct n)
show ?X 0 by (simp add: sdp-Skip)
fix n assume IH: ?X n
show ?X (Suc n)
proof (rule sub-distrib-pconjI, simp add: wp-eval)
  fix P :: ’s ⇒ real and Q :: ’s ⇒ real
  assume uP: unitary P and uQ: unitary Q
  from assms have hwlpa: nearly-healthy (wlp (repeat n a))
    and hwpa: healthy (wp (repeat n a))
    by(auto intro: healthy-intros)
  from uP and hwlpa have unitary (wlp (repeat n a) P)
    by(blast)
moreover from uQ and hwpa have unitary (wp (repeat n a) Q)
    by(blast)
ultimately have wp a (wp (repeat n a) P) \&\& wp a (wp (repeat n a) Q) \vdash
    wp a (wp (repeat n a) P) \&\& wp (repeat n a) Q
  using sdpa by(blast)
also {
  from hwlpa have nearly-healthy (wp (repeat n a))
    by(rule healthy-intros)
  with uP have sound (wp (repeat n a) P)
    by(auto)
  moreover from hwpa have sound (wp (repeat n a) Q)
    by(auto intro: healthy-intros)
  ultimately have sound (wp (repeat n a) P \&\& wp (repeat n a) Q)
    by(rule exp-conj-sound)
  moreover {
    from uP uQ have sound (P \&\& Q)
      by(auto intro: exp-conj-sound)
    with hwpa have sound (wp (repeat n a) (P \&\& Q))
      by(auto intro: healthy-intros)
  }
moreover from uP uQ IH
have wp (repeat n a) P \&\& wp (repeat n a) Q \vdash wp (repeat n a) (P \&\& Q)
  by(blast)
ultimately
have wp a (wp (repeat n a) P) \&\& wp a (wp (repeat n a) Q) \vdash
  wp a (wp (repeat n a) (P \&\& Q))
  by(rule mono-transD[OF healthy-monoD, OF hwp])
} 
finally show wp a (wp (repeat n a) P) \&\& wp a (wp (repeat n a) Q) \vdash
4.7. WELL-DEFINED PROGRAMS.

\[
wp a (wp (\text{repeat } n \ a) (P && Q)) .
\]
\text{qed}

\text{qed}

\textbf{lemma} \texttt{sdp-SetPC}:
\begin{itemize}
\item \texttt{fixes} \texttt{p::'a} \Rightarrow 's prog
\item \texttt{assumes} \texttt{sdp: } \lambda s. a \in \text{supp} (P s) \rightarrow \text{sub-distrib-pconj} (p a)
\item \texttt{and} \texttt{fin: } \lambda s. \text{finite} (\text{supp} (P s))
\item \texttt{and} \texttt{nnp: } \lambda s. a \leq P s a
\item \texttt{and} \texttt{sub: } \lambda s. \text{sum} (P s) (\text{supp} (P s)) \leq 1
\end{itemize}
\texttt{shows} \texttt{sub-distrib-pconj} (\texttt{SetPC} \ p \ P)
\texttt{proof}\texttt{(rule sub-distrib-pconjI, simp add: wp-eval, rule le-fanI)}
\texttt{fix} \texttt{Q::'s} \Rightarrow \texttt{real and} \texttt{R::'s} \Rightarrow \texttt{real and} \texttt{s::'s}
\texttt{assume} \texttt{uQ: unitary Q and uR: unitary R}
\texttt{have} ((\lambda s. \Sigma a \in \text{supp} (P s). P s a * wp (p a) Q s) \&\&
(\lambda s. \Sigma a \in \text{supp} (P s). P s a * wp (p a) R s)) s =
(\Sigma a \in \text{supp} (P s). P s a * wp (p a) Q s) + (\Sigma a \in \text{supp} (P s). P s a * wp (p a) R s)) \oplus 1
\texttt{by(simp add: exp-conj-def pconj-def)}
\texttt{also have} ...
(\Sigma a \in \text{supp} (P s). P s a * (wp (p a) Q s + wp (p a) R s)) \oplus 1
\texttt{by(simp add: sum.distrib field-simps)}
\texttt{also from} \texttt{sub}
\texttt{have} ...
(\Sigma a \in \text{supp} (P s). P s a * (wp (p a) Q s + wp (p a) R s)) \oplus
(\Sigma a \in \text{supp} (P s). P s a)
\texttt{by(rule tminus-right-antimono)}
\texttt{also from} \texttt{fin}
\texttt{have} ...
(\Sigma a \in \text{supp} (P s). P s a * (wp (p a) Q s + wp (p a) R s) \oplus P s a)
\texttt{by(rule tminus-sum-mono)}
\texttt{also from} \texttt{nnp}
\texttt{have} ...
(\Sigma a \in \text{supp} (P s). P s a * (wp (p a) Q s + wp (p a) R s \oplus 1))
\texttt{by(simp add: tminus-left-distrib)}
\texttt{also have} ...
(\Sigma a \in \text{supp} (P s). P s a * (wp (p a) Q \&\& wp (p a) R s)) s
\texttt{by(simp add: pconj-def exp-conj-def)}
\texttt{also}\{\texttt{from} \texttt{sdp uQ uR}
\texttt{have} \lambda a. a \in \text{supp} (P s) \Rightarrow wp (p a) Q \&\& wp (p a) R \Rightarrow wp (p a) (Q \&\& R)
\texttt{by(blast intro: sub-distrib-pconjD)}
\texttt{with} \texttt{nnp}
\texttt{have} (\Sigma a \in \text{supp} (P s). P s a * (wp (p a) Q \&\& wp (p a) R s)) \leq
(\Sigma a \in \text{supp} (P s). P s a * (wp (p a) (Q \&\& R) s)) s
\texttt{by(blast intro: sum-mono mult-left-mono)}\}
\texttt{finally show} ((\lambda s. \Sigma a \in \text{supp} (P s). P s a * wp (p a) Q s) \&\&
(\lambda s. \Sigma a \in \text{supp} (P s). P s a * wp (p a) R s)) s \leq
(\Sigma a \in \text{supp} (P s). P s a * wp (p a) (Q \&\& R) s) .
\text{qed}

\textbf{lemma} \texttt{sdp-SetDC}:
fixes $p :: 'a \Rightarrow 's$

assumes

sdp: $\forall s. a \in S \Rightarrow \text{sub-distrib-pconj} (p a)$

and hwp: $\forall s. a \in S \Rightarrow \text{healthy} (\text{wp} (p a))$

and hwlp: $\forall s. a \in S \Rightarrow \text{nearly-healthy} (\text{wlp} (p a))$

and ne: $\forall s. S \neq \{\}$

shows $\text{sub-distrib-pconj} (\text{SetDC} p S)$

proof (rule sub-distrib-pconjI, rule le-funI)

fix $P :: 's \Rightarrow \text{real}$ and $Q :: 's \Rightarrow \text{real}$ and $s :: 's$

assume $uP$: unitary $P$ and $uQ$: unitary $Q$

from $uP$ hwp have $\forall x. x \in (\lambda a. \text{wp} (p a) P) \cdot S s \Rightarrow \text{unitary} x$ by (auto)

hence $\forall y. y \in (\lambda a. \text{wp} (p a) P) \cdot S s \Rightarrow 0 \leq y$ by (auto)

hence $\forall a. a \in S s \Rightarrow \text{wp} (\text{SetDC} p S) P s \leq \text{wp} (p a) P s$

unfolding wp-eval by (intro cInf-lower bdd-belowI, auto)

moreover {
  from $uQ$ hwp have $\forall a. a \in S s \Rightarrow 0 \leq \text{wp} (p a) Q s$ by (blast)
  hence $\forall a. a \in S s \Rightarrow \text{wp} (\text{SetDC} p S) Q s \leq \text{wp} (p a) Q s$
  unfolding wp-eval by (intro cInf-lower bdd-belowI, auto)
}

ultimately have $\forall a. a \in S s \Rightarrow \text{wp} (\text{SetDC} p S) P s + \text{wp} (\text{SetDC} p S) Q s \oplus 1 \leq \text{wp} (p a) P s + \text{wp} (p a) Q s \oplus 1$

by (auto intro: tminus-left-mono add-mono)

also have $\forall a. \text{wp} (p a) P s + \text{wp} (p a) Q s \oplus 1 = (\text{wp} (p a) P \&\& \text{wp} (p a) Q) s$

by (simp add: exp-conj-def pconj-def)

also from $sdp$ $uP$ $uQ$

have $\forall a. a \in S s \Rightarrow \ldots a \leq \text{wp} (p a) (P \&\& Q) s$

by (blast)

also have $\forall a. \ldots a = \text{wp} (p a) (\lambda s. P s + Q s \oplus 1) s$

by (simp add: exp-conj-def pconj-def)

finally show $(\text{wp} (\text{SetDC} p S) P \&\& \text{wp} (\text{SetDC} p S) Q) s \leq \text{wp} (\text{SetDC} p S) (P \&\& Q) s$

unfolding exp-conj-def pconj-def wp-eval

using ne by (blast intro: cInf-greatest)

qed

lemma $sdp$-Bind:

$[ \forall s. \text{sub-distrib-pconj} (p (f s)) ] \Rightarrow \text{sub-distrib-pconj} (\text{Bind} f p)$

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.

lemma $sdp$-loop:

fixes $body :: 's \Rightarrow \text{prog}$
assumes sdp-body: sub-distrib-pconj body
and hwlp: nearly-healthy (wp body)
and hwp: healthy (wp body)
shows sub-distrib-pconj (do G −→ body od)

proof (rule sub-distrib-pconjI, rule loop-induct[OF hwp hwlp])

fix P Q:′s expect and S:=′s trans × ′s trans) set

assume uP: unitary P and uQ: unitary Q
and ffst: ∀ x∈S. feasible (fst x)
and usnd: ∀ x∈S. ∀ Q. unitary Q −→ unitary (snd x Q)
and IH: ∀ x∈S. snd x P ⋀ fst x Q ⊢ fst x (P & & Q)

show Inf-utrans (snd ′ S) P & & Sup-trans (fst ′ S) Q ⊢ Sup-trans (fst ′ S) (P & & Q)

proof (cases)

assume S = {} 
thus ?thesis
by (simp add: Inf-trans-def Sup-trans-def Inf-utrans-def
Inf-exp-def Sup-exp-def exp-conj-def)

next 

assume ne: S ≠ {}

let ?f s = 1 + Sup-trans (fst ′ S) (P & & Q) s − Inf-utrans (snd ′ S) P s

from ne obtain t where tin: t ∈ fst ′ S by (auto)
from ne obtain u where uin: u ∈ snd ′ S by (auto)

from tin ffst uP uQ have utPQ: unitary (t (P & & Q))
by (auto intro: exp-conj-unitary)
hence ∃ s. 0 ≤ t (P & & Q) s by (auto)
also {
from ffst tin have le: le-utrans t (Sup-trans (fst ′ S))
by (auto intro: Sup-trans-upper)
with uP uQ have ∃ s. t (P & & Q) s ≤ Sup-trans (fst ′ S) (P & & Q) s
by (auto intro: exp-conj-unitary)
}
finally have nn-rhs: ∃ s. 0 ≤ Sup-trans (fst ′ S) (P & & Q) s .

have ∃ R. Inf-utrans (snd ′ S) P & & R ⊢ Sup-trans (fst ′ S) (P & & Q) ⇒ R ≤ ?f

proof (rule contrapos-pp, assumption)

fix R
assume ¬ R ≤ ?f
then obtain s where ¬ R s ≤ ?f s by (auto)
hence gt: ?f s < R s by (simp)

from nn-rhs have g1: 1 ≤ 1 + Sup-trans (fst ′ S) (P & & Q) s by (auto)
hence Sup-trans (fst ′ S) (P & & Q) s = Inf-utrans (snd ′ S) P s & & if 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 gt have I \leq\: 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 (Inf-utrans (snd ' S) P \&\& R) s \leq\: Sup-trans (fst ' S) (P \&\& Q) s
by (simp add: exp-conj-def)
thus \neg Inf-utrans (snd ' S) P \&\& R \vdash\: Sup-trans (fst ' S) (P \&\& Q)
by (auto)
qed

moreover have \forall t\in\: fst ' S. Inf-utrans (snd ' S) P \&\& t Q \vdash\: Sup-trans (fst ' S) (P \&\& Q)
proof
fix t assume tin: t \in\: fst ' S
then obtain x where xin: x \in\: S and fx: t = fst x by (auto)
from xin have snd x \in\: snd ' S by (auto)
with uP u snd have Inf-utrans (snd ' S) P \vdash\: snd x P
by (auto intro: le-utransD \[ OF Inf-utrans-lower \])
hence Inf-utrans (snd ' S) P \&\& fst x Q \vdash\: snd x P \&\& fst x Q
by (auto intro: entails-frame)
also from xin IH have ... \vdash\: fst x (P \&\& Q)
by (auto)
also from xin \[\[Fst exp-conj-unitary[OF uP uQ]\]
have ... \vdash\: Sup-trans (fst ' S) (P \&\& Q)
by (auto intro: le-utransD \[ OF Sup-trans-upper \])
finally show Inf-utrans (snd ' S) P \&\& t Q \vdash\: Sup-trans (fst ' S) (P \&\& Q)
by (simp add: fx)
qed

ultimately have bt: \forall t\in\: fst ' S. t Q \vdash\: ?f
by (blast)

have Sup-trans (fst ' S) Q = Sup-exp \{ t Q \mid t \in\: fst ' S \}
by (simp add: Sup-trans-def)
also have ... \vdash\: ?f
proof (rule Sup-exp-least)
from bt show \forall R\in\{ t Q \mid t \in\: fst ' S \}. R \vdash\: ?f
by (blast)
from ne obtain t where tin: t \in\: fst ' S by (auto)
with ffst uQ have unitary (t Q) by (auto)
hence \lambda s. 0 \vdash\: t Q by (auto)
also from tin bt have ... \vdash\: ?f
by (auto)
finally show nneg (\lambda s. 1 + Sup-trans (fst ' S) (P \&\& Q) s -
Inf-utrans (snd ' S) P s)
by (auto)
4.7. WELL-DEFINED PROGRAMS.

qed

finally have Inf-utrans (snd · S) P &\& Sup-trans (fst · S) Q \+ Inf-utrans (snd · S) P &\& ?f

by(auto intro:entails-frame)

also from nn-rhs have \+ \+ \+ Sup-trans (fst · S) (P &\& Q)

by(simp add:exp-conj-def pconj-def)

finally show \?thesis .

qed

next

fix P Q::'s expect and t u::'s trans

assume uP: unitary P and uQ: unitary Q

and ft: feasible t

and uu: \&\& Q. unitary Q \+ unitary (u Q)

and IH: u P &\& t Q \+ t (P &\& Q)

show wlp (body ::; Embed u \& G s \& Skip) P &\&

wp (body ::; Embed t \& G s \& Skip) Q \+ wp (body ::; Embed t \& G s \& Skip) (P &\& Q)

proof(rule le-fun1, simp add:wp-eval exp-conj-def pconj-def)

fix s::'s

have « G » s \& wp body (u P) s + (1 - « G » s) \& P s +

(« G » s \& wp body (t Q) s + (1 - « G » s) \& Q s) \& 1 =

(« G » s \& wp body (u P) s + « G » s \& wp body (t Q) s) +

((1 - « G » s) \& P s + (1 - « G » s) \& Q s) \& («G» s + (1 - «G» s))

by(simp add:ac-simps)

also have ... \+ « G » s \& wp body (u P) s + « G » s \& wp body (t Q) s \& («G» s)

((1 - « G » s) \& P s + (1 - « G » s) \& Q s) \& (1 - «G» s))

by(rule tminus-add-mono)

also have ... =

« G » s \& (wp body (u P) s + wp body (t Q) s \& 1) +

(1 - « G » s) \& (P s + Q s \& 1)

by(simp add:tminus-left-distrib distrib-left)

also {

from uP uQ ft uu

have wp body (u P) &\& wp body (t Q) \+ wp body (u P &\& t Q)

by(auto intro:sub-distrib-pecnjD[OF sdp-body])

also from IH unitary-sound[OF uP] unitary-sound[OF uQ] ft

unitary-sound[OF uu[OF uP]]

have ... \+ wp body (t (P &\& Q))

by(blast intro:mono-transD[OF healthy-monoD, OF hwp] exp-conj-sound)

finally have wp body (u P) s + wp body (t Q) s \& 1 \+ wp body (t (\(\lambda s. P s + Q s \& 1\)) s)

by(auto simp:exp-conj-def pconj-def)

hence « G » s \& (wp body (u P) s + wp body (t Q) s \& 1) +

(1 - « G » s) \& (P s + Q s \& 1) \+ « G » s \& wp body (t (\(\lambda s. P s + Q s \& 1\)) s) +

(1 - « G » s) \& (P s + Q s \& 1)

by(auto intro:add-right-mono mult-left-mono)
4.7.3 The Well-Defined Predicate.

definition well-def :: 's prog ⇒ bool

where
well-def ≡ healthy (wp prog) ∧ nearly-healthy (wlp prog)
∧ wp-under-wlp prog ∧ sub-distrib-pconj prog
∧ sublinear (wp prog) ∧ bd-cts (wp prog)

lemma well-defI[intro]:
[ [ healthy (wp prog); nearly-healthy (wlp prog);
wp-under-wlp prog; sub-distrib-pconj prog; sublinear (wp prog);
bd-cts (wp prog) ] ⇒ well-def prog
unfolding well-def-def by(simp)

lemma well-def-wp-healthy[dest]:
well-def prog ⇒ healthy (wp prog)
unfolding well-def-def by(simp)

lemma well-def-wlp-nearly-healthy[dest]:
well-def prog ⇒ nearly-healthy (wlp prog)
unfolding well-def-def by(simp)

lemma well-def-wp-under[dest]:
well-def prog ⇒ wp-under-wlp prog
unfolding well-def-def by(simp)

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. WELL-DEFINED PROGRAMS.

**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]:
well-def prog \( \Rightarrow \) 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:
\[ \text{well-def a; well-def b} \] \( \Rightarrow \) 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:
\[ \text{well-def a; well-def b; unitary P} \] \( \Rightarrow \) well-def (a \( \land \) 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 ] \Rightarrow 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:
[ \[ \forall x \in S \Rightarrow well-def (a x); \forall s. S s \neq \{ \}; \\
\forall s. finite (S s) \] ] \Rightarrow well-def (SetDC a S) \\

lemma wd-SetPC:
[ \[ \forall x \in (\text{supp} (p s)) \Rightarrow well-def (a x); \forall s. unitary (p s); \forall s. finite (\text{supp} (p s)); \\
\forall s. \text{sum} (p s) (\text{supp} (p s)) \leq 1 \] ] \Rightarrow well-def (SetPC a p) \\
by (iprover intro!:well-defI healthy-wp-SetPC nearly-healthy-wlp-SetPC wp-under-wlp-SetPC sdp-SetPC sublinear-wp-SetPC cts-wp-SetPC dest:wd-destrs unitary-sound sound-nneg [OF unitary-sound] nnegD)

lemma wd-Embed:
fixes t :: 's trans \\
assumes ht: healthy t and st: sublinear t and ct: bd-cts t \\
shows well-def (Embed t) \\
proof (intro well-defI) \\
from ht show healthy (wp (Embed t)) nearly-healthy (wlp (Embed t)) \\
by (simp add: wp-def wp-def Embed-def healthy-nearly-healthy)+ \\
from st show sublinear (wp (Embed t)) by (simp add: wp-def Embed-def) \\
show wp-under-wlp (Embed t) by (simp add: wp-under-wlp-def wp-eval) \\
show sub-distrib-pconj (Embed t) \\
by (rule sub-distrib-pconjI, auto intro:le-funI[OF sublinearD[OF st, where a=1 and b=1 and c=1, simplified]]) \\
show simp: exp-conj-def pconj-def wp-def wp-def Embed-def \\
from ct show bd-cts (wp (Embed t)) \\
by (simp add: wp-def Embed-def) \\
qed

lemma wd-repeat: \\
well-def a \Rightarrow well-def (\text{repeat} n a) \\

lemma wd-Bind: \\
[ [ \forall s. well-def (a (f s)) ] ] \Rightarrow well-def (\text{Bind} f a) \\

lemma wd-loop:
4.8. THE LOOP RULES

well-def body \implies well-def (do G \implies body od) 
by(blast intro:healthy-wp-loop nearly-healthy-wlp-loop  
    wp-under-wlp-loop sdp-loop sublinear-wp-loop cts-wp-loop)

lemmas wd-intros =
    wd-Abort wd-Skip  wd-Apply 
    wd-Embed wd-Seq  wd-PC 
    wd-DC  wd-SetPC  wd-SetDC 
    wd-Bind wd-repeat wd-loop

end

4.8 The Loop Rules

theory Loops imports WellDefined begin

Given a well-defined body, we can annotate a loop using an invariant, just
as in the classical setting.

4.8.1 Liberal and Strict Invariants.

A probabilistic invariant generalises a boolean one: it entails itself, given the
loop guard.

definition
wp-inv :: (′s ⇒ bool) ⇒ (′s prog ⇒ (′s ⇒ real) ⇒ bool 
where
wp-inv G body I <-→ (∀ s. «G» s * I s ≤ wp body I s)

lemma wp-invI:
\A I. (∀ s. «G» s * I s ≤ wp body I s) \implies wp-inv G body I 
by(simp add:wp-inv-def)

definition
wlp-inv :: (′s ⇒ bool) ⇒ (′s prog ⇒ (′s ⇒ real) ⇒ bool 
where
wlp-inv G body I <-→ (∀ s. «G» s * I s ≤ wlp body I s)

lemma wlp-invI:
\A I. (∀ s. «G» s * I s ≤ wlp body I s) \implies wlp-inv G body I 
by(simp add:wlp-inv-def)

lemma wlp-invD:
wp-inv G body I \implies «G» s * I s ≤ 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» ⊨ wp body «I»

proof

fix s
show («G» & & «I») s ≤ wp body «I» s
proof(cases G s)
case False
with hb show ?thesis
  by(auto simp:exp-conj-def)
next
case True
hence («G» & & «I») s = «G» s * «I» s
  by(simp add:exp-conj-def)
also from inv have («G» s * «I» s ≤ wp body «I» s 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 ≤ wlp do G → body od (λs. «N G» s * I s)
(is I ≤ wp do G → body od ?P)

proof

let ?f Q s = «G» s * wlp body Q s + «N G» s * ?P s
have I ⊨ gfp-exp ?f
proof(rule gfp-exp-upperbound[OF - uI])
  have I = (λs. («G» s + «N G» s) * I s) by(simp add:negate-embed)
  also have ... = (λs. «G» s * I s + «N G» s * I s)
    by(simp add:algebra-simps)
  also have ... = (λs. «G» s * («G» s * I s) + «N G» s * («N G» s * I s))
    by(simp add:embed-bool-idem algebra-simps)
  also have ... ⊨ (λs. «G» s * wlp body I s + «N G» s * («N G» s * I s))
    using inv by(auto dest:wlp-invD intro:mult-left-mono)
  finally show I ⊨ (λs. «G» s * wlp body I s + «N G» s * («N G» s * I s)).
qed

also from uI well-def-wlp-nearly-healthy[OF wd] have ... = wlp do G → body od ?P
  by(auto intro!:wlp-Loop1[symmetric] unitary-intros)
finally show ?thesis.
qed
4.8. THE LOOP RULES

4.8.3 Total Correctness

The first total correctness lemma for loops which terminate with probability 1 [McIver and Morgan, 2004, Lemma 7.3.1, §7, p. 186].

lemma wp-Loop:
  assumes wd: well-def body
  and inv: wlp-inv G body I
  and unit: unitary I
  shows I & & wp (do G −→ body od) (λs. I) ⊨ wp (do G −→ body od) (λs. «N G» s * I s)
  (is I & & ?T ⊨ wp ?loop ?X)

proof −

We first appeal to the liberal loop rule:

  from assms have I & & ?T ⊨ wp ?loop ?X & & ?T
  by(blast intro:exp-conj-mono-left wlp-Loop)

Next, by sub-conjunctivity:

  also {
    from wd have sdp-loop: sub-distrib-pconj (do G −→ body od)
    by(blast intro:sdp-intros)

    from wd unit have wp ?loop ?X & & ?T ⊨ wp ?loop (?X & & (λs. 1))
    by(blast intro:sub-distrib-pconjD sdp-intros unitary-intros)
  }

Finally, the conjunction collapses:

  finally show ?thesis
  by(simp add:exp-conj-1-right sound-intros sound-nneg unit unitary-sound)

qed

4.8.4 Unfolding

lemma wp-loop-unfold:
  fixes body :: 's prog
  assumes sP: sound P
  and h: healthy (wp body)
  shows wp (do G −→ body od) P =
  (λs. «N G» s * P s + «G» s * wp body (wp (do G −→ body od) P) s)

proof (simp only: wp-eval)
  let ?X t = wp (body ;; Embed t « G » Skip)
  have equi-trans (lfp-trans ?X)
    (wp (body ;; Embed (lfp-trans ?X) « G » Skip))
  proof(intro lfp-trans-unfold)
  fix t::'s trans and P::'s expect
  assume st: A Q. sound Q =⇒ sound (t Q)
  and sP: sound P
  with h show sound (?X t P)
  by(rule wp-loop-step-sound)
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 « G \oplus Skip))
(wp (body ;; Embed u « G \oplus Skip))

by (iprover intro: wp-loop-step-mono)

let ?v = λP s. bound-of P

from h show le-trans (wp (body ;; Embed ?v « G \oplus Skip)) ?v

by (intro le-transI, simp add: wp-eval lfp-loop-fp [unfolded negate-embed])

fix P :: ′s expect

assume sound P thus sound (?v P) by (auto)

d qed

also have equiv-trans ...

(P. «N G» s * P s + «G» s * wp body (wp (Embed (lfp-trans ?X) P) s))

by (rule equiv-transI, simp add: wp-eval algebra-simps negate-embed)

finally show lfp-trans ?X P =

(P. «N G» s * P s + «G» s * wp body (lfp-trans ?X P) s)

using sP unfolding wp-eval by (blast)

d qed

lemma wp-loop-noguard:

[ healthy (wp body); sound P; ¬ G s ] \implies wp do G → body od P s = P s

by (subst wp-loop-unfold, simp-all)

lemma wp-loop-guard:

[ healthy (wp body); sound P; G s ] \implies

wp do G → body od P s = wp (body ;; do G → body od) P s

by (subst wp-loop-unfold, simp-all add: wp-eval)

d end

4.9 The Algebra of pGCL

theory Algebra imports WellDefined begin

Programs in pGCL have a rich algebraic structure, largely mirroring that for GCL. We show that programs form a lattice under refinement, with a \(\sqcap\) and a \(\sqcup\) 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.
4.9. THE ALGEBRA OF PGCL

definition
refines :: 's prog ⇒ 's prog ⇒ bool (infix ⊑ 70)
where
prog ⊑ prog' ≡ ∀P. sound P −→ wp prog P ⊨ wp prog' P

lemma refinesI[intro]:
[ ∃P. sound P −→ wp prog P ⊨ wp prog' P ] ⇒ prog ⊑ prog'
unfolding refines-def by(simp)

lemma refinesD[dest]:
[ prog ⊑ prog'; sound P ] ⇒ wp prog P ⊨ wp prog' P
unfolding refines-def by(simp)

The equivalence relation below will turn out to be that induced by refinement. It is also the application of equiv-trans to the weakest precondition.

definition
pequiv :: 's prog ⇒ 's prog ⇒ bool (infix ≃ 70)
where
prog ≃ prog' ≡ ∀P. sound P −→ wp prog P = wp prog' P

lemma pequivI[intro]:
[ ∃P. sound P −→ wp prog P = wp prog' P ] ⇒ prog ≃ prog'
unfolding pequiv-def by(simp)

lemma pequivD[dest,simp]:
[ prog ≃ prog'; sound P ] ⇒ wp prog P = wp prog' P
unfolding pequiv-def by(simp)

lemma pequiv-equiv-trans:
a ≃ b ↔ 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 ∩ b = b ∩ a
unfolding DC-def by(simp add:ac-simps)

lemma DC-assoc[ac-simps]:
a ∩ (b ∩ c) = (a ∩ b) ∩ c
unfolding DC-def by(simp add:ac-simps)
lemma $DC$-idem:
\[ a \parallel a = a \]
unfolding $DC$-def by(simp)

lemma $AC$-comm[ac-simps]:
\[ a \parallel b = b \parallel a \]
unfolding $AC$-def by(simp add:ac-simps)

lemma $AC$-assoc[ac-simps]:
\[ a \parallel (b \parallel c) = (a \parallel b) \parallel c \]
unfolding $AC$-def by(simp add:ac-simps)

lemma $AC$-idem:
\[ a \parallel a = a \]
unfolding $AC$-def by(simp)

lemma $PC$-quasi-comm:
\[ a \ifp b = b \ifp (\lambda s. 1 - p) \ifp a \]
unfolding $PC$-def by(simp add:algebra-simps)

lemma $PC$-idem:
\[ a \ifp a = a \]
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 $a \Rightarrow$ 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:
\[ (a \parallel b) \sqsubseteq a \]
by(auto intro!:refinesI simp:wp-eval)

lemma right-refines-DC:
\[ (a \parallel b) \sqsubseteq b \]
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 $a \sqsubseteq (b \parallel c)$
proof
fix $P$::'s \Rightarrow real assume $sP$: sound $P$
with assms have $wp$ $a$ $P \not\Rightarrow wp$ $b$ $P$ and $wp$ $a$ $P \not\Rightarrow wp$ $c$ $P$
by (auto dest: refinesD)
thus \( wp \; a \; P \vdash wp \; (b \sqcap c) \; P \)
by (auto simp: wp-eval intro: min.boundedI)
qed

lemma DC-mono:
fixes a::'s prog
assumes rab: \( a \sqsubseteq b \) and rcd: \( c \sqsubseteq d \)
shows \( (a \sqcap c) \sqsubseteq (b \sqcap d) \)
proof (rule refinesI, unfold wp-eval, rule le-funI)
fix \( P::\'s \Rightarrow real \) and \( s::\'s \)
assume \( sP: \text{ sound } P \)
with assms have \( \bigwedge s. \; wp \; a \; P \; s \leq wp \; b \; P \; s \) and
\( \bigwedge s. \; wp \; c \; P \; s \leq wp \; d \; P \; s \)
by (auto)
thus \( \min (wp \; a \; P \; s) \; (wp \; c \; P \; s) \leq \min (wp \; b \; P \; s) \; (wp \; d \; P \; s) \)
by (auto)
qed

Laws relating angelic choice and refinement

lemma left-refines-AC:
\( a \sqsubseteq (a \sqcup b) \)
by (auto intro!: refinesI simp: wp-eval)

lemma right-refines-AC:
\( b \sqsubseteq (a \sqcup b) \)
by (auto intro!: refinesI simp: wp-eval)

lemma AC-refines:
fixes a::'s prog and b and c
assumes rac: \( a \sqsubseteq c \) and rbc: \( b \sqsubseteq c \)
shows \( (a \sqcup b) \sqsubseteq c \)
proof
fix \( P::\'s \Rightarrow real \) assume \( sP: \text{ sound } P \)
with assms have \( \bigwedge s. \; wp \; a \; P \; s \leq wp \; b \; P \; s \) and
\( \bigwedge s. \; wp \; c \; P \; s \leq wp \; d \; P \; s \)
by (auto dest: refinesD)
thus \( wp \; (a \sqcup b) \; P \vdash wp \; c \; P \)
unfolding wp-eval by (auto)
qed

lemma AC-mono:
fixes a::'s prog
assumes rab: \( a \sqsubseteq b \) and rcd: \( c \sqsubseteq d \)
shows \( (a \sqcup c) \sqsubseteq (b \sqcup d) \)
proof (rule refinesI, unfold wp-eval, rule le-funI)
fix \( P::\'s \Rightarrow real \) and \( s::\'s \)
assume \( sP: \text{ sound } P \)
with assms have \( wp \; a \; P \; s \leq wp \; b \; P \; s \) and
\( wp \; c \; P \; s \leq wp \; d \; P \; s \)
by (auto)
thus \( \max (wp a P s) (wp c P s) \leq \max (wp b P s) (wp d P s) \)
by (auto)
qed

Laws depending on the arithmetic of \( a \oplus b \) and \( a \sqcap b \) together

lemma DC-refines-PC:
assumes unit: unitary \( p \)
shows \( (a \sqcap b) \sqsubseteq (a \oplus b) \)
proof (rule refinesI, unfold wp-eval, rule le-funI)
fix \( s \) and \( P :\forall a \Rightarrow \text{real} \)
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 \( \min (wp a P s) (wp b P s) \leq p s \cdot \min (wp a P s) + (1 - p s) \cdot \min (wp b P s) \)
proof (cases \( wp a P s \leq wp b P s \), simp-all add: min.absorb1 min.absorb2)
case True note le = this
have \( wp a P s = (p s + (1 - p s)) \cdot wp a P s \)
  by (simp)
also have \( ... = p s \cdot wp a P s + (1 - p s) \cdot wp a P s \)
  by (simp only: distrib-right)
also {
  from le and nn-np have \( (1 - p s) \cdot wp a P s \leq (1 - p s) \cdot wp b P s \)
    by (rule mult-left-mono)
hence \( p s \cdot wp a P s + (1 - p s) \cdot wp a P s \leq p s \cdot wp b P s + (1 - p s) \cdot wp b P s \)
  by (rule add-left-mono)
}
finally show \( wp a P s \leq p s \cdot wp a P s + (1 - p s) \cdot wp b P s \).
next
case False
then have le: \( wp b P s \leq wp a P s \)
  by (simp)
have \( wp b P s = (p s + (1 - p s)) \cdot wp b P s \)
  by (simp)
also have \( ... = p s \cdot wp b P s + (1 - p s) \cdot wp b P s \)
  by (simp only: distrib-right)
also {
  from le and nn-p have \( p s \cdot wp b P s \leq p s \cdot wp a P s \)
    by (rule mult-left-mono)
hence \( p s \cdot wp b P s + (1 - p s) \cdot wp b P s \leq p s \cdot wp a P s + (1 - p s) \cdot wp b P s \)
    by (rule add-right-mono)
}
finally show \( wp b P s \leq p s \cdot wp a P s + (1 - p s) \cdot wp b P s \).
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 ⊕ b) ⊑ (a ❌ b)
proof(rule refinesI, unfold wp-eval, rule le-funI)
fix s and P::'a ⇒ real
assume sound: sound P
from unit have p s ≤ 1 by(blast)
hence nn-np: 0 ≤ 1 − p s by(simp)
show p s * wp a P s + (1 − p s) * wp b P s ≤ max (wp a P s) (wp b P s)
proof(cases wp a P s ≤ wp b P s)
  case True
  have ... = wp b P s
  by(auto simp: wp-eval)
  also have ... = wp a P s
  by(auto)
  finally show ?thesis .
next
  case False
  have ... = wp a P s
  by(auto)
  also from leab
  have ... = max (wp a P s) (wp b P s)
  by(auto)
  finally show ?thesis .
qed

Laws depending on the arithmetic of a ❌ b and a ❌ b together

lemma DC-refines-AC:
(a ❌ b) ⊑ (a ❌ b)
by(auto intro!:refinesI simp:wp-eval)

Laws Involving Refinement and Equivalence

lemma pr-trans[trans]:
fixes A::'a prog
assumes prAB: A ⊑ B
and prBC: B ⊑ C
shows A ⊑ C
proof
fix \( P : 'a \Rightarrow \text{real} \)
assume \( sP : \text{sound } P \)
with \( \text{prAB} \) have \( \text{wp } A \ P \vdash \text{wp } B \ P \) by(\( \text{blast} \))
also from \( sP \) and \( \text{prBC} \) have \( \vdash \text{wp } C \ P \) by(\( \text{blast} \))
finally show \( \text{wp } A \ P \vdash \ldots \).
qed

lemma pequiv-refl[intro!,simp]:
\( a \simeq a \)
by(\( \text{auto} \))

lemma pequiv-comm[ac-simps]:
\( a \simeq b \iff b \simeq a \)
unfolding pequiv-def
by(\( \text{rule iffI, safe, simp-all} \))

lemma pequiv-pr[dest]:
\( a \simeq b \implies a \sqsubseteq b \)
by(\( \text{auto} \))

lemma pequiv-trans[intro,trans]:
\( \[ a \simeq b ; b \simeq c \] \implies a \simeq c \)
unfolding pequiv-def by(\( \text{intro!:order-trans} \))

lemma pequiv-pr-trans[intro,trans]:
\( \[ a \sqsubseteq b ; b \simeq c \] \implies a \sqsubseteq c \)
unfolding pequiv-def refines-def by(\( \text{simp} \))

Refinement induces equivalence by antisymmetry:

lemma pequiv-antisym:
\( \[ a \sqsubseteq b ; b \sqsubseteq a \] \implies a \simeq b \)
by(\( \text{auto intro:antisym} \))

lemma pequiv-DC:
\( \[ a \simeq c ; b \simeq d \] \implies (a \sqcap b) \simeq (c \sqcap d) \)
by(\( \text{auto intro!:DC-mono pequiv-antisym simp:ac-simps} \))

lemma pequiv-AC:
\( \[ a \simeq c ; b \simeq d \] \implies (a \sqcup b) \simeq (c \sqcup d) \)
by(\( \text{auto intro!:AC-mono pequiv-antisym simp:ac-simps} \))

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)`
  - `wa`: well-def `a`
  - `wb`: well-def `b`
  - `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 `sub`: sub-add `(wp b)`
  - by (auto dest: `sublinear-subadd[OF well-def-wp-sublinear])`
- **Fix** `P` :: `s ⇒ real`
  - **Assume** `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 be 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 = λs. bound-of P − P s`
- **Have** `sPc`: sound `?Pc`

**Proof** (rule `soundI`)

- **From** `sP` have `∀ s. 0 ≤ P s` by (auto)
- **Hence** `∀ s. wp ?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)
  - **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 (λs. bound-of P) s` by (simp add: `maximalD determ-maximalD`)
also have ... = wp a (λs. ?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)
also from less have ... < wp b ?Pc s + wp b P s
  by(auto)
also from sab sP sPc have ... ≤ wp b (λs. ?Pc s + P s) s
  by(blast)
finally have ¬wp b (λs. bound-of P) s ≤ bound-of P
  by(simp)
thus ¬bounded-by (bound-of P) (wp b (λs. bound-of P))
  by(auto)
next

However,

fix P::'s ⇒ real assume sP: sound P
hence nneg (λs. bound-of P) by(auto)
moreover have bounded-by (bound-of P) (λs. bound-of P) by(auto)
ultimately
show bounded-by (bound-of P) (wp b (λs. bound-of P))
  using wb by(auto dest!:well-def-wp-healthy)
qed

4.9.4 The Algebraic Structure of Refinement

Well-defined programs form a half-bounded semilattice under refinement, where Abort is bottom, and a ∪ b is inf. There is no unique top element, but all fully-deterministic programs are maximal.

The type that we construct here is not especially useful, but serves as a convenient way to express this result.

quotient-type 's program =
  's prog / partial : λa b. a ≃ b ∧ well-def a ∧ well-def b
proof(rule part-equivpI)
  have Skip ≃ Skip and well-def Skip by(auto intro:wd-intros)
  thus ∃x. x ≃ x ∧ well-def x ∧ well-def x by(blast)
  show symp (λa b. a ≃ b ∧ well-def a ∧ well-def b)
    proof(rule sympI, safe)
    fix a::'a prog and b
    assume a ≃ b
    hence equiv-trans (wp a) (wp b)
      by(simp add:pequiv-equiv-trans)
    thus b ≃ a by(simp add:ac-simps pequiv-equiv-trans)
  qed
  show transp (λa b. a ≃ b ∧ well-def a ∧ well-def b)
    by(rule transpI, safe, rule pequiv-trans)
  qed
instantiation program :: (type) semilattice-inf begin

proof(safe)
  fix a::'a prog and b c d
  assume a ≃ b hence b ≃ a by(simp add:ac-simps)
  also assume a ⊑ c
  also assume c ≃ d
  finally show b ⊑ d .

proof(safe)
  fix a::'a prog and b c d
  assume a ≃ b hence b ≃ a by(simp add:ac-simps)
  also assume a ⊑ c
  also assume c ≃ d
  finally show b ⊑ d .

next
  fix a::'a prog and b c d
  assume a ≃ b hence b ≃ a by(simp add:ac-simps)
  also assume a ⊑ c
  finally show a ⊑ c .

next
  fix a b c::'a prog and d
  assume c ≃ d
  also assume d ⊑ b
  also assume a ≃ b hence b ≃ a by(simp add:ac-simps)
  finally have c ⊑ a .
  moreover assume ¬ c ⊑ a
  ultimately show False by(auto)

next
  fix a b c::'a prog and d
  assume c ≃ d hence d ≃ c by(simp add:ac-simps)
  also assume c ⊑ a
  also assume a ≃ b
  finally have d ⊑ b .
  moreover assume ¬ d ⊑ b
  ultimately show False by(auto)
qed

lift-definition
  \textit{inf-program} :: \textquote{a program} $\Rightarrow$ \textquote{a program} $\Rightarrow$ \textquote{a program} is \textit{DC}
proof (safe)
  fix a b c d :: \textquote{a program}
  assume a $\simeq$ b and c $\simeq$ d
  thus \((a \sqcap c) \simeq (b \sqcap d)\) by (rule pequiv-DC)
next
  fix a c :: \textquote{a program}
  assume well-def a well-def c
  thus well-def \((a \sqcap c)\) by (rule wd-intros)
next
  fix a c :: \textquote{a program}
  assume well-def a well-def c
  thus well-def \((a \sqcap c)\) by (rule wd-intros)
qed

instance
proof
  fix x y :: \textquote{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 :: \textquote{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 :: \textquote{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 \textit{program} :: \text{(type) bot}
proof
lift-definition
  \textit{bot-program} :: \textquote{a program} is \textit{Abort}
  by (auto intro:wd-intros)
4.9. THE ALGEBRA OF PGCL

instance ..
end

lemma eq-det: \( \forall a \ b::\mathsf{prog}. \ (a \simeq b; \ \mathsf{determ} (\mathsf{wp} \ a) \equiv \mathsf{determ} (\mathsf{wp} \ b)) \)
proof(intro determI additiveI maximalI)
  fix a b::\mathsf{prog} and P::\mathsf{real} and Q::\mathsf{real} and s::\mathsf{real}
  assume da: determ (wp a)
  assume sP: sound P and sQ: sound Q and eq: a \simeq b
  hence wp b (\\lambda s. \ P s + Q s) s = wp a (\\lambda s. \ P s + Q s) s
    by(simp add:sound-intros)
  also from da sP sQ have ...
    = wp a P s + wp a Q s
    by(simp add:additiveD determ-additiveD)
  also from eq sP sQ have ...
    = wp b P s + wp b Q s
    by(simp add:pequivD)
  finally show wp b (\\lambda s. \ P s + Q s) s = wp b P s + wp b Q s
next
  fix a b::\mathsf{prog} and c::\mathsf{real}
  assume da: determ (wp a)
  assume a \simeq b hence b \simeq a
  by(simp add:ac-simps)
  moreover assume nn: \( \theta \leq c \)
  ultimately have wp b (\\lambda s. \ c) = wp a (\\lambda s. \ c)
    by(simp add:pequivD const-sound)
  also from da nn have ...
    = (\\lambda s. \ c)
    by(simp add:determ-maximalD maximalD)
  finally show wp b (\\lambda s. \ c) = (\\lambda s. \ c).
qed

lift-definition
  \mathsf{pdeterm} :: \mathsf{\mathcal{P}rogram} \Rightarrow \mathsf{bool}
  is \lambda a. \ \mathsf{determ} (\mathsf{wp} \ a)
proof(safe)
  fix a b::\mathsf{prog}
  assume a \simeq b and determ (wp a)
  thus determ (wp b) by(rule eq-det)
next
  fix a b::\mathsf{prog}
  assume a \simeq b hence b \simeq a
  by(simp add:ac-simps)
  moreover assume determ (wp b)
  ultimately show determ (wp a) by(rule eq-det)
qed

lemma determ-maximal:
  \[ \mathsf{pdeterm} \ a; \ a \leq x \ \Rightarrow \ a = x \]
  by(transfer, auto intro:refines-determ)
4.9.5 Data Refinement

A projective data refinement construction for pGCL. By projective, we mean that the abstract state is always a function ($\varphi$) of the concrete state. Refinement may be predicated ($G$) on the state.

**Definition**

\[
\text{drefines} :: (b \Rightarrow a) \Rightarrow 'a \text{ prog} \Rightarrow 'b \text{ prog} \Rightarrow \text{ bool}
\]

where

\[
drefines \varphi G A B \equiv \forall P Q. (\text{unitary } P \land \text{ unitary } Q \land (P \vdash wp A Q)) \rightarrow (G \&\& (P \circ \varphi) \vdash wp B (Q \circ \varphi))
\]

**Lemma** \text{drefinesD}[dest]:

\[
\begin{align*}
\text{drefines} \varphi G A B; \text{unitary } P; \text{unitary } Q; P \vdash wp A Q & \implies \\
(G \&\& (P \circ \varphi) \vdash wp B (Q \circ \varphi))
\end{align*}
\]

**Unfolding** \text{drefines-def} by (\text{blast})

We can alternatively use $G$ as an assumption:

**Lemma** \text{drefinesD2}:

\begin{align*}
\text{assumes } &dr: \text{drefines } \varphi G A B \\
&\text{and } uP: \text{unitary } P \\
&\text{and } uQ: \text{unitary } Q \\
&\text{and } wpA: P \vdash wp A Q \\
&\text{and } G: G s \\
\text{shows } & (P \circ \varphi) s \leq wp B (Q \circ \varphi) s
\end{align*}

**Proof** –

from $uP$ have $0 \leq (P \circ \varphi) s$ unfolding $o$-def by (\text{blast})

with $G$ have $(P \circ \varphi) s = (G \&\& (P \circ \varphi)) s$

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

also from \text{assms} have $\ldots \leq wp B (Q \circ \varphi) s$ by (\text{blast})

finally show $(P \circ \varphi) s \leq \ldots$.

**QED**

This additional form is sometimes useful:

**Lemma** \text{drefinesD3}:

\begin{align*}
\text{assumes } &dr: \text{drefines } \varphi G a b \\
&\text{and } G: G s \\
&\text{and } uQ: \text{unitary } Q \\
&\text{and } wa: \text{well-def } a \\
\text{shows } & wp a Q (\varphi s) \leq wp b (Q \circ \varphi) s
\end{align*}

**Proof** –

let $?L s' = wp a Q s'$

from $uQ wa$ have $sL$: sound $?L$ by (\text{blast})

from $uQ wa$ have $bL$: bounded-by 1 $?L$ by (\text{blast})

have $?L \vdash ?L$ by (\text{simp})

with $sL$ and $bL$ and \text{assms}

show $?thesis$

by (\text{blast intro:drefinesD2}[OF dr, where $P=?L$, simplified])
4.9. THE ALGEBRA OF PGCL

qed

lemma drefinesI[intro]:
\[ \forall P \ Q. \ \text{unitary } P; \ \text{unitary } Q; \ P \vdash wp A Q \] \implies
\{G\} && (P \circ \varphi) \vdash wp B (Q \circ \varphi) \]
drefines \varphi G A B

unfolding drefines-def by (blast)

Use G as an assumption, when showing refinement:

lemma drefinesI2:
fixes \ A :: 'a prog
and \ B :: 'b prog
and \ \varphi :: 'b \Rightarrow 'a
and \ G :: 'b \Rightarrow bool
assumes \ wB : well-def B
and withAs: \ \forall s. \ G s \Rightarrow (P \circ \varphi) s \leq wp B (Q \circ \varphi) s
shows drefines \varphi G A B

proof
fix \ P and Q
assume \ uP : unitary P
and \ uQ : unitary Q
and \ wpA : P \vdash wp A Q

hence \ \forall s. \ G s \Rightarrow (P \circ \varphi) s \leq wp B (Q \circ \varphi) s
using withAs by (blast)

moreover
from \ uQ have unitary (Q o \varphi)
unfolding o-def by (blast)

moreover
from \ uP have unitary (P o \varphi)
unfolding o-def by (blast)

ultimately
show \{G\} && (P \circ \varphi) \vdash wp B (Q \circ \varphi)
using \ wB by (blast intro:entails-pconj-assumption)

qed

lemma dr-strengthen-guard:
fixes \ a :: 's prog and \ b :: 't prog
assumes \ fg: \ \forall s. \ F s \Rightarrow G s
and \ drab: drefines \varphi G a b
shows drefines \varphi F a b

proof (intro drefinesI)
fix \ P :: 's expect
assume \ uP : unitary P and \ uQ : unitary Q
and \ wp: P \vdash wp a Q

from \ fg have \ \forall s. \ «F» s \leq «G» s by (simp add: embed-bool-def)
hence \ («F» \&\& (P \circ \varphi)) \vdash («G» \&\& (P \circ \varphi)) by (auto intro:pconj-mono le-funI)
Probabilistic correspondence, \textit{pcorres}, is equality on distribution transformers, modulo a guard. It is the analogue, for data refinement, of program equivalence for program refinement.

\textbf{definition}

\textit{pcorres} :: \((b \Rightarrow a) \Rightarrow (b \Rightarrow \text{bool}) \Rightarrow a \text{ prog} \Rightarrow b \text{ prog} \Rightarrow \text{bool}

where

\textit{pcorres} \varphi G A B \leftarrow\rightarrow

\((\forall Q. \text{unitary} Q \rightarrow \langle G \rangle \&\& (wp A Q \circ \varphi) = \langle G \rangle \&\& wp B (Q \circ \varphi))\)

\textbf{lemma} \textit{pcorresI}:

\begin{align*}
\langle Q. \text{unitary} Q \rightarrow (wp A Q \circ \varphi) = wp B (Q \circ \varphi) \rangle \Rightarrow \\
\text{pcorres} \varphi G A B
\end{align*}

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

Often easier to use, as it allows one to assume the precondition.

\textbf{lemma} \textit{pcorresI2}[intro]:

\begin{align*}
\text{fixes} A &\::'a \text{ prog} \text{ and } B &\::'b \text{ prog} \\
\text{assumes} \text{withG} &\::(\forall Q s. [\text{unitary} Q; G s] \Rightarrow wp A Q (\varphi s)= wp B (Q \circ \varphi) s) \\
\text{and} wA &\::\text{well-def} A \\
\text{and} wB &\::\text{well-def} B \\
\text{shows} \text{pcorres} \varphi G A B
\end{align*}

\text{proof}(\text{rule pcorresI, rule ext})

\begin{align*}
\text{fix} Q &\::'a \Rightarrow \text{real} \text{ and } s &\::'b \\
\text{assume} uQ &\::\text{unitary} Q \\
\text{hence} uQ&\varphi; \text{unitary} (Q \circ \varphi) \text{ by(auto)} \\
\text{show} (\langle G \rangle \&\& wp A Q (\varphi s)) s = (\langle G \rangle \&\& wp B (Q \circ \varphi)) s
\end{align*}

\text{proof}(\text{cases G s})

\begin{align*}
\text{case} \text{True note this} \\
\text{moreover} \\
\text{from} \text{well-def-wp-healthy[OF wA]} uQ \text{ have} 0 \leq wp A Q (\varphi s) \text{ by(blast)} \\
\text{moreover} \\
\text{from} \text{well-def-wp-healthy[OF wB]} uQ\varphi \text{ have} 0 \leq wp B (Q \circ \varphi) s \text{ by(blast)} \\
\text{ultimately show} \text{thesis}
\end{align*}

\text{using} uQ \text{ by(simp add:exp-conj-def withG)}

\text{next}

\begin{align*}
\text{case} \text{False note this} \\
\text{moreover} \\
\text{from} \text{well-def-wp-healthy[OF wA]} uQ \text{ have} wp A Q (\varphi s) \leq 1 \text{ by(blast)} \\
\text{moreover} \\
\text{from} \text{well-def-wp-healthy[OF wB]} uQ\varphi \text{ have} wp B (Q \circ \varphi) s \leq 1 \\
\text{by(blast dest!:healthy-bounded-byD intro:sound-nneg)} \\
\text{ultimately show} \text{thesis by(simp add:exp-conj-def)}
\end{align*}

\textbf{qed}

\textbf{qed}
4.9. THE ALGEBRA OF PGCL

lemma pcorresD:
[ pcorres ϕ G A B; unitary Q ] ⇒ «G» & & (wp A Q o ϕ) = «G» & & wp B (Q o ϕ)

unfolding pcorres-def by(simp)

Again, easier to use if the precondition is known to hold.

lemma pcorresD2:
assumes pc: pcorres ϕ G A B
and uQ: unitary Q
and wA: well-def A and wB: well-def B
and G: G s
shows wp A Q (ϕ s) = wp B (Q o ϕ) s

proof
from uQ well-def-wp-healthy[OF wA] have θ ≤ 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 pe uQ have «G» & & (wp A Q o ϕ) = «G» & & wp B (Q o ϕ) by pcorresD)
  hence «G» s & wp A Q (ϕ s) = «G» s & wp B (Q o ϕ) s
  unfolding exp-conj-def o-def by(rule fun-cong)
}
also { from uQ have sound Q by(auto)
  hence sound (Q o ϕ) by(auto intro:sound-intros)
  with well-def-wp-healthy[OF wB] have θ ≤ 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 $\text{drAB}$: $\text{drefines } \varphi \ G \ A \ B$
and $\text{drBC}$: $\text{drefines } \varphi' \ G' \ B \ C$
and $\text{Gimp}$: $\forall s. \ G' \ s \Rightarrow G \ (\varphi' \ s)$
shows $\text{drefines } (\varphi \ o \ \varphi') \ G' \ A \ C$

proof (rule drefinesI)
fix $P::'a \Rightarrow \text{real}$ and $Q::'a \Rightarrow \text{real}$ and $s::'a$
assume $uP$: unitary $P$ and $uQ$: unitary $Q$
and $\text{wpA}$: $P \vdash \text{wp } A \ Q$

have $\langle G' \rangle \ & \ & \langle G \ o \ \varphi' \rangle = \langle G' \rangle$
proof (rule ext, unfold exp-conj-def)
fix $x$
show $\langle G' \rangle \ x \ & \ & \langle G \ o \ \varphi' \rangle \ x = \langle G' \rangle \ x \ (\text{is } ?X)$
proof (cases $G' \ x$
  case False then show $?X$ by (simp)
next
  case True
moreover
with $\text{Gimp}$ have $(G \ o \ \varphi') \ x$ by (simp add: o-def)
ultimately
show $?X$ by (simp)
qed

with $uP$
have $\langle G' \rangle \ & \ & (P \ o \ (\varphi \ o \ \varphi')) = \langle G' \rangle \ & \ & ((\langle G \rangle \ & \ & (P \ o \ \varphi)) \ o \ \varphi')$
  by (simp add: exp-conj-assoc o-assoc)
also {
  from $uP \ uQ \ \text{wpA}$ and $\text{drAB}$
  have $\langle G \rangle \ & \ & (P \ o \ \varphi) \vdash \text{wp } B \ (Q \ o \ \varphi)$
    by (blast intro: drefinesD)
  with $\text{drBC}$ and $uP \ uQ$
  have $\langle G' \rangle \ & \ & ((\langle G \rangle \ & \ & (P \ o \ \varphi)) \ o \ \varphi') \vdash \text{wp } C \ ((Q \ o \ \varphi) \ o \ \varphi')$
    by (blast intro: unitary-intros drefinesD)
}
finally
show $\langle G' \rangle \ & \ & (P \ o \ (\varphi \ o \ \varphi')) \vdash \text{wp } C \ (Q \ o \ (\varphi \ o \ \varphi'))$
  by (simp add: o-assoc)
qed

Data refinement composes with program refinement:

lemma $\text{pr-dr-trans}$[trans]:
assumes $\text{prAB}$: $A \sqsubseteq B$
and $\text{drBC}$: $\text{drefines } \varphi \ G \ B \ C$
shows $\text{drefines } \varphi \ G \ A \ C$
4.9. THE ALGEBRA OF PGCL

proof (rule drefinesI)
fix P and Q
assume uP: unitary P
and uQ: unitary Q
and wpA: P ⊢ wp A Q

note wpA
also from uQ and prAB have wp A Q ⊢ wp B Q by (blast)
finally have P ⊢ wp B Q.

with uP uQ drBC
show «G» & & (P o ϕ) ⊢ wp C (Q o ϕ) by (blast intro: drefinesD)
qed

lemma dr-pr-trans[trans]:
assumes drAB: drefines ϕ G A B
assumes prBC: B ⊑ C
shows drefines ϕ G A C
proof (rule drefinesI)
fix P and Q
assume uP: unitary P
and uQ: unitary Q
and wpA: P ⊢ wp A Q

with drAB have «G» & & (P o ϕ) ⊢ wp B (Q o ϕ) by (blast intro: drefinesD)
also from uQ prBC have ... ⊢ wp C (Q o ϕ) by (blast)
finally show «G» & & (P o ϕ) ⊢ ...
qed

If the projection ϕ commutes with the transformer, then data refinement is reflexive:

lemma dr-refl:
assumes wa: well-def a
and comm: A Q. unitary Q =⇒ wp a Q o ϕ ⊢ wp a (Q o ϕ)
shows drefines ϕ G a a
proof (intro drefinesI2 wa)
fix P and Q and s
assume wp: P ⊢ wp a Q
assume uQ: unitary Q

have (P o ϕ) s = P (ϕ s) by (simp)
also from wp have ... ⊢ wp a Q (ϕ s) by (blast)
also { from comm uQ have wp a Q o ϕ ⊢ wp a (Q o ϕ) by (blast)
    hence (wp a Q o ϕ) s ≤ wp a (Q o ϕ) s by (blast)
    hence wp a Q (ϕ s) ≤ ... by (simp)
}
finally show (P o ϕ) s ≤ wp a (Q o ϕ) s.
qed
Correspondence implies data refinement

**Lemma pcorres-drefine:**

**Assumes**
- $\text{pcorres } \varphi \ G \ A \ C$
- $wC: \text{well-def } C$

**Shows**
- $\text{drefines } \varphi \ G \ A \ C$

**Proof**

- Fix $P$ and $Q$
- Assume $uP: \text{unitary } P$ and $uQ: \text{unitary } Q$
- and $wpA: \vdash wp A Q$

  - From $wpA$ have $P \circ \varphi \vdash wp A Q \circ \varphi$ by (simp add: $o$-def le-fun-def)

  - Hence $\langle G \rangle \&\& (P \circ \varphi) \vdash \langle G \rangle \&\& (wp A Q \circ \varphi)$ by (rule exp-conj- mono-right)

  - Also from $\text{corres } uQ$ have $\vdash wp C (Q \circ \varphi)$ by (rule pcorresD)

  - Also have $\vdash wp C (Q \circ \varphi)$

**Proof** (rule le-funI)

- Fix $s$

  - From $uQ$ have $\text{unitary } (Q \circ \varphi)$ by (rule unitary-intros)

  - With $\text{well-def-wp-healthy} [OF wC]$ have $\text{nn-wpC}: 0 \leq wp C (Q \circ \varphi) s$ by (blast)

  - Show $\langle G \rangle \&\& wp C (Q \circ \varphi) s \leq wp C (Q \circ \varphi) s$

**Proof** (cases $G \ s$)

  - Case True

    - With $\text{nn-wpC}$ show $\text{thesis}$ by (simp add: exp-conj-def)

  - Next

    - Case False note this

    - Moreover { from $uQ$ have $\text{unitary } (Q \circ \varphi)$ by (simp)

      - With $\text{well-def-wp-healthy} [OF wC]$ have $wp C (Q \circ \varphi) s \leq 1$ by (auto)

    } moreover note $\text{nn-wpC}$

    - Ultimately show $\text{thesis}$ by (simp add: exp-conj-def)

    qed

  qed

  - Finally show $\langle G \rangle \&\& (P \circ \varphi) \vdash wp C (Q \circ \varphi)$.

**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**
- $a::'a \text{ prog}$ and $b::'b \text{ prog}$

**Assumes**
- $\text{determ } (wp \ a)$
- and $wa: \text{well-def } a$
- and $wb: \text{well-def } b$
- and $dr: \text{drefines } \varphi \ G \ a \ b$

**Shows**
- $\text{pcorres } \varphi \ G \ a \ b$

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)

**note** ha = well-def-wp-healthy[OF wa]

**note** hb = well-def-wp-healthy[OF wb]

**from** wb have sublinear (wp b) by(auto)

**moreover from** hb have feasible (wp b) by(auto)

**ultimately have** sub: sub-add (wp b) by(rule sublinear-subadd)

**fix** Q::’a ⇒ real

**assume** uQ: unitary Q

**hence** uQ: unitary (Q o ϕ) by(auto)

**assume** ne: «G» & wp a Q o ϕ ≠ «G» & wp b (Q o ϕ)

**hence** ne': wp a Q o ϕ ≠ wp b (Q o ϕ) by(auto)

From refinement, «G» & wp a Q 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 b (Q o ϕ) ⊢ «G» & wp a 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 & (wp a Q o ϕ) s ≤ «G» s & wp b (Q o ϕ) s

by(auto intro:pconj-mono)

**moreover from** uQ ha have wp a Q (ϕ s) ≤ 1

by(auto dest:healthy-bounded-byD)

**moreover from** uQ ha have 0 ≤ wp a Q (ϕ s)

by(auto)

**ultimately**

**show** «G» s & wp a Q o ϕ) s ≤ «G» s & wp b (Q o ϕ) s

by(simp add:pconj-assoc)

**qed**

If the programs do not correspond, the terms must differ somewhere, and given the previous result, the second must be somewhere strictly larger than the first:

**have** rule: ∃s. («G» & wp a Q o ϕ)) s < («G» & wp b (Q o ϕ)) s

**proof** (rule contrapos-pp[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:

$$(\langle G \rangle \land (wp a Q \circ \varphi)) \ s < (\langle G \rangle \land wp b (Q \circ \varphi)) \ s$$
by(blast)

The transformers themselves must differ at this point:

hence larger: $wp a Q (\varphi \ s) < wp b (Q \circ \varphi) \ s$

proof(cases $G \ s$)

case True
moreover from ha uQ have $0 \leq wp a Q (\varphi \ s)$
by(blast)
moreover from hb uQ have $0 \leq wp b (Q \circ \varphi) \ s$
by(blast)
moreover note less-s
ultimately show ?thesis by(simp add:exp-conj-def)

next
case False
moreover from ha uQ have $wp a Q (\varphi \ s) \leq 1$
by(blast)
moreover {
from uQ have bounded-by 1 $(Q \circ \varphi)$
by(blast)
moreover from unitary-sound[OF uQ]
have sound $(Q \circ \varphi)$ by(auto)
ultimately have $wp b (Q \circ \varphi) \ s \leq 1$
using hb by(auto)
}
moreover note less-s
ultimately show ?thesis by(simp add:exp-conj-def)
qed

from less-s have $(\langle G \rangle \land wp a Q \circ \varphi)) \ s \neq (\langle G \rangle \land wp b (Q \circ \varphi)) \ s$
by(force)

$G$ must also hold, as otherwise both would be zero.

hence $G\ s$: $G \ s$

proof(rule contrapos-np)
assume $\lnot G \ s$
moreover from ha uQ have $wp a Q (\varphi \ s) \leq 1$
by(blast)
moreover {
from uQ have bounded-by 1 $(Q \circ \varphi)$
by(blast)
moreover from unitary-sound[OF uQ]
have sound $(Q \circ \varphi)$ by(auto)
ultimately have $wp b (Q \circ \varphi) \ s \leq 1$
using hb by(auto)
}
ultimately
4.9. THE ALGEBRA OF PGCL

show \((\Box G \land \Box (wp a \circ \varphi)) \) \(s = (\Box G \land \Box wp b \circ (Q \circ \varphi)) \) \(s\)
by\((\text{simp add:exp-conj-def})\)
qed

Take a carefully constructed expectation:

let \(?Qc = \lambda s. \text{bound-of} \ Q - Q \ s\)
have \(bQc: \text{bounded-by} \ 1 \ ?Qc\)
proof\((\text{rule bounded-byI})\)
fix \(s\)
from \(uQ\) have \(\text{bound-of} \ Q \leq 1 \ \text{and} \ 0 \leq Q \ s\) by(auto)
thus \(\text{bound-of} \ Q - Q \ s \leq 1\) by(auto)
qed
have \(sQc: \text{sound} \ ?Qc\)
proof\((\text{rule soundI})\)
from \(bQc\) show \(\text{bounded} \ ?Qc\) by(auto)
show \(\text{nneg} \ ?Qc\)
proof\((\text{rule nnegI})\)
fix \(s\)
from \(uQ\) have \(Q \ s \leq \text{bound-of} \ Q\) by(auto)
thus \(0 \leq \text{bound-of} \ Q - Q \ s\) by(auto)
qed
qed

By the maximality of \(wp a, wp b\) must violate feasibility, by mapping \(s\) to something strictly greater than \(\text{bound-of} \ Q\).

from \(uQ\) have \(0 \leq \text{bound-of} \ Q\) by(auto)
with \(da\) have \(\text{bound-of} \ Q = \ wp a (\lambda s. \text{bound-of} \ Q) (\varphi \ s)\)
by\((\text{simp add:maximalD determ-maximalD})\)
also have \(wp a (\lambda s. \text{bound-of} \ Q) (\varphi \ s) = wp a (\lambda s. Q \ s + \ ?Qc \ s) (\varphi \ s)\)
by\((\text{simp})\)
also \{
from \(da\) have \(\text{additive} \ (wp a)\) by(blast)
with \(uQ\) \(sQc\)
have \(wp a (\lambda s. Q \ s + \ ?Qc \ s) (\varphi \ s) = wp a Q (\varphi \ s) + wp a \ ?Qc (\varphi \ s)\) by\((\text{subst additiveD, blast+})\)
\}
also \{
from \(ha\) and \(sQc\) and \(bQc\)
have \(\Box G \land (wp a \ ?Qc \circ \varphi) \vdash wp b \ ?Qc \circ \varphi\)
by\((\text{blast intro!:drefinesD[OF dr]})\)
hence \((\Box G \land (wp a \ ?Qc \circ \varphi)) \ s \leq wp b \ ?Qc \circ \varphi \ s\)
by\((\text{blast})\)
moreover from \(sQc\) and \(ha\)
have \(0 \leq wp a (\lambda s. \text{bound-of} \ Q - Q \ s) (\varphi \ s)\)
by\((\text{blast})\)
ultimately
have \(wp a \ ?Qc (\varphi \ s) \leq wp b \ ?Qc \circ \varphi \ s\)
using \(G\)-s by\((\text{simp add:exp-conj-def})\)
hence \( \text{wp a } Q (\varphi s) + \text{wp a } ?Qc (\varphi s) \leq \text{wp a } Q (\varphi s) + \text{wp b } (?Qc o \varphi) s \)
\( \text{by (rule add-left-mono)} \)
also with larger
have \( \text{wp a } Q (\varphi s) + \text{wp b } (?Qc o \varphi) s < \text{wp b } (Q o \varphi) s + \text{wp b } (?Qc o \varphi) s \)
\( \text{by (auto)} \)
finally
have \( \text{wp a } Q (\varphi s) + \text{wp a } ?Qc (\varphi s) < \text{wp b } (Q o \varphi) s + \text{wp b } (?Qc o \varphi) s \).
\}
also from \( \text{sab and unitary-sound[OF } uQ] \) and \( sQc \)
have \( \text{wp b } (Q o \varphi) s + \text{wp b } (?Qc o \varphi) s \leq \text{wp b } (\lambda s. (Q o \varphi) s + (?Qc o \varphi) s) s \)
\( \text{by (blast)} \)
also have \( ... = \text{wp b } (\lambda s. \text{bound-of } Q) s \)
\( \text{by (simp)} \)
finally
show \( \neg \text{feasible } (\text{wp b}) \)
\( \text{proof (rule contrapos-ppn)} \)
\( \text{assume } fb: \text{feasible } (\text{wp b}) \)
\( \text{have bounded-by } (\text{bound-of } Q) (\lambda s. \text{bound-of } Q) \text{ by (blast)} \)
\( \text{hence bounded-by } (\text{bound-of } Q) (\text{wp b } (\lambda s. \text{bound-of } Q)) \)
\( \text{using } uQ \text{ by (blast intro:feasible-boundedD[OF } fb]} \)
\( \text{hence } \text{wp b } (\lambda s. \text{bound-of } Q) s \leq \text{bound-of } Q \text{ by (blast)} \)
\( \text{thus } \neg \text{bound-of } Q < \text{wp b } (\lambda s. \text{bound-of } Q) s \text{ by (simp)} \)
\( \text{qed} \)
\( \text{qed} \)

### 4.9.7 Structural Rules for Correspondence

#### lemma pcорres-Skip:

\[ \text{pcорres } \varphi G \text{ Skip Skip} \]

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

Correspondence composes over sequential composition.

#### lemma pcорres-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 \( \text{pcAB: pcорres } \varphi G A B \)
and \( \text{pcCD: pcорres } \varphi H C D \)
and \( \text{waA: well-def } A \) and \( \text{wbB: well-def } B \)
and \( \text{wcC: well-def } C \) and \( \text{wdD: well-def } D \)
and \( \text{p3p2: } \bigwedge Q. \text{ unitary } Q \Rightarrow \langle I \rangle & \& \text{ wp } B Q = \text{ wp } B \langle \langle H \rangle & \& Q \rangle \)
and \( \text{p1p3: } \bigwedge s. \text{ G s } \Rightarrow I s \)
shows \( \text{pcорres } \varphi G (A::C) (B::D) \)

\( \text{proof (rule pcорresI)} \)

fix \( Q::'b \Rightarrow \text{ real} \)
assume \( uQ:: \text{ unitary } Q \)
4.9. THE ALGEBRA OF PGCL

with well-def-wp-healthy[OF wC] have uCQ: unitary (wp C Q) by(auto)
from uQ well-def-wp-healthy[OF wD] have uDQ: unitary (wp D (Q o φ))
  by(auto dest:unitary-comp)

have p3p1: \( \forall R. \exists [unitary R; unitary S; \langle I \rangle \& \& R = \langle I \rangle \& \& S] \implies \langle G \rangle \& \& R = \langle G \rangle \& \& S \)
proof(rule ext)
  fix \( R::'c \Rightarrow real \) and \( S::'c \Rightarrow real \) and \( s::'c \)
  assume a3: \( \langle I \rangle \& \& R = \langle I \rangle \& \& S \)
  and uR: unitary R and uS: unitary S
  show \( \langle G \rangle \& \& R \) s = \( \langle G \rangle \& \& S \) s
proof(simp add:exp-conj-def, cases G s)
  case False note this
  moreover from uR have \( R \) s ≤ 1 by(blast)
  moreover from uS have \( S \) s ≤ 1 by(blast)
  ultimately show \( \langle G \rangle \& \& R \) s = \( \langle G \rangle \& \& S \) 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 \( \langle G \rangle \& \& R \) s = \( \langle G \rangle \& \& S \) s
  by(simp)
qed

qed

show \( \langle G \rangle \& \& (wp (A;\langle C \rangle) Q o φ) = \langle G \rangle \& \& wp (B;\langle D \rangle) (Q o φ) \)
proof(simp add:wp-eval)
  from uCQ pcAB have \( \langle G \rangle \& \& (wp A (wp C Q) o φ) = \langle G \rangle \& \& wp B ((wp C Q) o φ) \)
  by(auto dest:pcorresD)
also have \( \langle G \rangle \& \& wp B ((wp C Q) o φ) = \langle G \rangle \& \& wp B (wp D (Q o φ)) \)
proof(rule p3p1)
  from uCQ well-def-wp-healthy[OF wB] show unitary (wp B (wp C Q o φ))
  by(auto intro:unitary-comp)
from uDQ well-def-wp-healthy[OF wB] show unitary (wp B (wp D (Q o φ)))
  by(auto)
from uQ have \( \langle H \rangle \& \& (wp C Q o φ) = \langle H \rangle \& \& wp D (Q o φ) \)
  by(blast intro:pcorresD[OF pcCD])
thus \( \langle I \rangle \& \& wp B (wp C Q o φ) = \langle I \rangle \& \& wp B (wp D (Q o φ)) \)
  by(simp add:p3p2 uCQ uDQ)
qed

finally show \( \langle G \rangle \& \& (wp A (wp C Q) o φ) = \langle G \rangle \& \& wp B (wp D (Q o φ)) \)
  .
qed
4.9.8 Structural Rules for Data Refinement

**lemma dr-Skip:**

fixes $\varphi :: c \Rightarrow b$

shows $\text{drefines } \varphi \ G \text{ Skip Skip}$

**proof** (intro drefinesI2 wd-intros)

fix $P :: b \Rightarrow \text{real}$ and $Q :: b \Rightarrow \text{real}$ and $s :: c$

assume $P \vdash \wp \text{ Skip } Q$

hence $(P \circ \varphi) s \leq \wp \text{ Skip } Q (\varphi s)$ by(simp, blast)

thus $(P \circ \varphi) s \leq \wp \text{ Skip } (Q \circ \varphi) s$ by(simp add:wp-eval)

qed

**lemma dr-Abort:**

fixes $\varphi :: c \Rightarrow b$

shows $\text{drefines } \varphi \ G \text{ Abort Abort}$

**proof** (intro drefinesI2 wd-intros)

fix $P :: b \Rightarrow \text{real}$ and $Q :: b \Rightarrow \text{real}$ and $s :: c$

assume $P \vdash \wp \text{ Abort } Q$

hence $(P \circ \varphi) s \leq \wp \text{ Abort } Q (\varphi s)$ by(auto)

thus $(P \circ \varphi) s \leq \wp \text{ Abort } (Q \circ \varphi) s$ by(simp add:wp-eval)

qed

**lemma dr-Apply:**

fixes $\varphi :: c \Rightarrow b$

assumes commutes: $f \circ \varphi = \varphi \circ g$

shows $\text{drefines } \varphi \ G \text{ (Apply } f \text{) (Apply } g\text{)}$

**proof** (intro drefinesI2 wd-intros)

fix $P :: b \Rightarrow \text{real}$ and $Q :: b \Rightarrow \text{real}$ and $s :: c$

assume $\wp: P \vdash \wp (\text{Apply } f) Q$

hence $P \vdash (Q \circ f) \text{ by(simp add:wp-eval)}$

hence $P (\varphi s) \leq (Q \circ f) (\varphi s) \text{ by(blast)}$

also have $\ldots = Q ((f \circ \varphi) s) \text{ by(simp)}$

also with commutes

have $\ldots = ((Q \circ \varphi) \circ g) s \text{ by(simp)}$

also have $\ldots = \wp (\text{Apply } g) (Q \circ \varphi) s$

by(simp add:wp-eval)

finally show $(P \circ \varphi) s \leq \wp (\text{Apply } g) (Q \circ \varphi) s \text{ by(simp)}$

qed

**lemma dr-Seq:**

assumes drAB: $\text{drefines } \varphi \ P A B$

and drBC: $\text{drefines } \varphi \ Q C D$

and wpB: «$P \vdash \wp B \quad Q»$

and wB: well-def B

and wC: well-def C

and wD: well-def D

shows $\text{drefines } \varphi \ P (A;;;;C) (B;;;;D)$

**proof**

fix $R$ and $S$
assume \( uR: \text{unitary } R \) and \( uS: \text{unitary } S \)

and \( \text{wpAC: } R \vdash \text{wp } (A;;C) S \)

from \( uR \)

have \( «P» \&\& (R o \varphi) = «P» \&\& («P» \&\& (R o \varphi)) \)

by \((\text{simp add:exp-conj-assoc})\)

also \{

from \( \text{well-def-wp-healthy}[OF wC]\) \( uR uS \)

and \( \text{wpAC}[\text{unfolded eval-wp-Seq o-def}] \)

have \( «P» \&\& (R o \varphi) \vdash \text{wp } B (wp C S o \varphi) \)

by \((\text{auto intro:drefinesD[of drAB]})\)

with \( \text{wpB well-def-wp-healthy}[OF wC]\) \( uS \)

\( \text{sublinear-sub-conj}[OF \text{well-def-wp-sublinear}, OF wB] \)

have \( «P» \&\& («P» \&\& (R o \varphi)) \vdash \text{wp } B («Q» \&\& (wp C S o \varphi)) \)

by \((\text{auto intro!:entails-combine dest!:unitary-sound})\)

\}

also \{

from \( uS\) \( \text{well-def-wp-healthy}[OF wC]\)

have \( «Q» \&\& (wp C S o \varphi) \vdash \text{wp } D (S o \varphi) \)

by \((\text{auto intro!:drefinesD[of drBC]})\)

with \( \text{well-def-wp-healthy}[OF wB]\) \( \text{well-def-wp-healthy}[OF wC]\) \( \text{and unitary-sound}[OF uS]\)

have \( \text{wp } B («Q» \&\& (wp C S o \varphi)) \vdash \text{wp } B (wp D (S o \varphi)) \)

by \((\text{blast intro!:mono-transD})\)

\}

finally

show \( «P» \&\& (R o \varphi) \vdash \text{wp } (B;;D) (S o \varphi) \)

unfolding \( \text{wp-eval o-def} \).

qed

lemma \( \text{dr-repeat}: \)

fixes \( \varphi: 'a \Rightarrow 'b \)

assumes \( \text{dr-ab: } \text{drefines } \varphi G a b \)

and \( \text{Gpr: } «G» \vdash \text{wp } b «G» \)

and \( \text{wa: } \text{well-def } a \)

and \( \text{wb: } \text{well-def } b \)

shows \( \text{drefines } \varphi G (\text{repeat } n a) (\text{repeat } n b) (\text{is } ?X n) \)

proof \((\text{induct } n)\)

show \( ?X 0 \) by \((\text{simp add:dr-Skip})\)

fix \( n \)

assume \( \text{IH: } ?X n \)

thus \( ?X (\text{Suc } n) \) by \((\text{auto intro!:dr-Seq Gpr assms wd-intros})\)

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:
(λs. 0) ⊢ wp Abort Q
unfolding wp-eval by(simp)

lemma wlp-Abort:
(λs. 1) ⊢ wlp Abort Q
unfolding wp-eval by(simp)

lemma wp-Skip:
P ⊢ wp Skip P
unfolding wp-eval by(blast)

lemma wlp-Skip:
P ⊢ wlp Skip P
unfolding wp-eval by(blast)

lemma wp-Apply:
Q o f ⊢ wp (Apply f) Q
unfolding wp-eval by(simp)

lemma wlp-Apply:
Q o f ⊢ wlp (Apply f) Q
unfolding wp-eval by(simp)

lemma wp-Seq:
assumes ent-a: P ⊢ wp a Q
  and ent-b: Q ⊢ wp b R
  and wa:  well-def a
  and wb:  well-def b
  and s-Q:  sound Q
  and s-R:  sound R
shows P ⊢ wp (a ;; b) R
proof –

note ha = well-def-wp-healthy[OF wa]
note hb = well-def-wp-healthy[OF wb]
note ent-a
also from ent-b ha hb s-Q s-R have wp a Q ⊢ wp a (wp b R)
  by(blast intro:healthy-monoD2)
4.10. STRUCTURED REASONING

finally show \( ?thesis \) by (simp add: wp-eval)
qed

lemma wlp-Seq:
assumes ent-a: \( P \vdash wlp\ a\ Q \)
and ent-b: \( Q \vdash wlp\ b\ R \)
and wa: \( \text{well-def}\ a \)
and wb: \( \text{well-def}\ b \)
and u-Q: \( \text{unitary}\ Q \)
and u-R: \( \text{unitary}\ R \)
sows \( P \vdash wlp\ (a;;b)\ R \)
proof
  note ha = well-def-wlp-nearly-healthy[OF wa]
  note hb = well-def-wlp-nearly-healthy[OF wb]
  note ent-a
  also from ent-b ha hb u-Q u-R have wlp a Q \( \vdash wlp\ a\ (wlp\ b\ R) \)
  finally show \( ?thesis \) by (simp add: wp-eval)
qed

lemma wp-PC:
\((\lambda s. P\ s \ast wlp\ a\ Q\ s + (1 - P\ s) \ast wp\ b\ Q\ s) \vdash wp\ (a \oplus b)\ Q\) by (simp add: wp-eval)

lemma wlp-PC:
\((\lambda s. P\ s \ast wlp\ a\ Q\ s + (1 - P\ s) \ast wlp\ b\ Q\ s) \vdash wlp\ (a \oplus b)\ Q\) by (simp add: wp-eval)

A simpler rule for when the probability does not depend on the state.

lemma PC-fixed:
assumes wpa: \( P \vdash a\ ab\ R \)
and wpb: \( Q \vdash b\ ab\ R \)
and np: \( 0 \leq p \) and bp: \( p \leq 1 \)
sows \((\lambda s. p \ast P\ s + (1 - p) \ast Q\ s) \vdash (a\ (\lambda s. p) \oplus b)\ ab\ R\)
unfolding PC-def
proof (rule le-funI)
  fix s
  from wpa and np have \( p \ast P\ s \leq p \ast a\ ab\ R\ s \)
  by (auto intro: mult-left-mono)
  moreover {
    from bp have \( 0 \leq 1 - p \) by (simp)
    with wpb have \( (1 - p) \ast Q\ s \leq (1 - p) \ast b\ ab\ R\ s \)
    by (auto intro: mult-left-mono)
  }
  ultimately show \( p \ast P\ s + (1 - p) \ast Q\ s \leq p \ast a\ ab\ R\ s + (1 - p) \ast b\ ab\ R\ s \)
  by (rule add-mono)
qed
lemma wp-PC-fixed:
\[
P \vdash wp \ a \ R; \ Q \vdash wp \ b \ R; \ 0 \leq p; \ p \leq 1 \implies (\lambda s. p * P s + (1 - p) * Q s) \vdash wp \ (a \ (\lambda s. p) \oplus b) \ R
\]
by (simp add: wp-def PC-fixed)

lemma wlp-PC-fixed:
\[
P \vdash wlp \ a \ R; \ Q \vdash wlp \ b \ R; \ 0 \leq p; \ p \leq 1 \implies (\lambda s. p * P s + (1 - p) * Q s) \vdash wlp \ (a \ (\lambda s. p) \oplus b) \ R
\]
by (simp add: wlp-def PC-fixed)

lemma wp-DC:
\[
(\lambda s. \min (wp \ a \ Q s) \ (wp \ b \ Q s)) \vdash wp \ (a \sqcap b) \ Q
\]
unfolding wp-eval by (simp)

lemma wlp-DC:
\[
(\lambda s. \min (wlp \ a \ Q s) \ (wlp \ b \ Q s)) \vdash wlp \ (a \sqcap b) \ Q
\]
unfolding wp-eval by (simp)

Combining annotations for both branches:

lemma DC-split:
fixes a::'s prog and b
assumes wpb: P \vdash a \ ab \ R
and wpb: Q \vdash b \ ab \ R
shows (\lambda s. \min (P s) (Q s)) \vdash (a \sqcap b) \ ab \ R
unfolding DC-def
proof (rule le-funI)
fix s
from wpb have P s \leq a \ ab \ R s and Q s \leq b \ ab \ R s by (auto)
thus min (P s) (Q s) \leq \min (a \ ab \ R s) (b \ ab \ R s) by (auto)
qed

lemma wp-DC-split:
\[
P \vdash wpprog \ R; \ Q \vdash wpprog' \ R \implies (\lambda s. \min (P s) (Q s)) \vdash wp \ (\prod prog) \ R
\]
by (simp add: wp-def DC-split)

lemma wlp-DC-split:
\[
P \vdash wlpprog \ R; \ Q \vdash wlpprog' \ R \implies (\lambda s. \min (P s) (Q s)) \vdash wlp \ (\prod prog') \ R
\]
by (simp add: wlp-def DC-split)

lemma wp-DC-split-same:
\[
P \vdash wp \ Q; \ P \vdash wp \ prog' \ Q \implies P \vdash wp \ (\prod prog') \ Q
\]
unfolding wp-eval by (blast intro:min.boundedI)

lemma wlp-DC-split-same:
\[
P \vdash wlp \ Q; \ P \vdash wlp \ prog' \ Q \implies P \vdash wlp \ (\prod prog') \ Q
\]
unfolding wp-eval by (blast intro:min.boundedI)
lemma SetPC-split:
  fixes f::'x ⇒ 'y prog
  and p:'y ⇒ 'x ⇒ real
  assumes rec: \( \forall x s. x \in \text{supp} \, (p \, s) \implies P \, x \vdash f \, x \, \mathit{ab} \, Q \)
  and \( \text{nnp}: \forall s. \text{nneg} \, (p \, s) \)
  shows \( (\lambda s. \sum x \in \text{supp} \, (p \, s). p \, s \, x \ast P \, x \, s) \vdash \text{SetPC} \, f \, p \, \mathit{ab} \, Q \)

unfolding SetPC-def
proof (rule le-fun\_I)
  fix \( s \)
  from \( \text{rec} \) have \( \forall x. x \in \text{supp} \, (p \, s) \implies P \, x \, s \leq f \, x \, s \, \mathit{ab} \, Q \, s \) by (blast)

moreover from \( \text{nnp} \) have \( \forall x. 0 \leq p \, s \, x \) by (blast)

ultimately have \( \forall x. x \in \text{supp} \, (p \, s) \implies p \, s \, x \ast P \, x \, s \leq p \, s \, x \ast f \, x \, \mathit{ab} \, Q \, s \)
  by (blast intro: mult-left-mono)

thus \( (\sum x \in \text{supp} \, (p \, s). p \, s \, x \ast P \, x \, s) \leq (\sum x \in \text{supp} \, (p \, s). p \, s \, x \ast f \, x \, \mathit{ab} \, Q \, s) \)
  by (rule sum-mono)

qed

lemma wp-SetPC-split:
  \([ \forall x \, s. x \in \text{supp} \, (p \, s) \implies P \, x \, s \leq f \, x \, s \, \mathit{ab} \, Q \, s; (\lambda s. \, \text{nneq} \, (p \, s)) \] \implies

(\lambda s. \sum x \in \text{supp} \, (p \, s). p \, s \, x \ast P \, x \, s) \vdash \text{wp} \, (\text{SetPC} \, f \, p) \, Q \)

by (simp add: wp-def SetPC-split)

lemma wlp-SetPC-split:
  \([ \forall x \, s. x \in \text{supp} \, (p \, s) \implies P \, x \vdash \text{wlp} \, (f \, x) \, Q; (\lambda s. \, \text{nneq} \, (p \, s)) \] \implies

(\lambda s. \sum x \in \text{supp} \, (p \, s). p \, s \, x \ast P \, x \, s) \vdash \text{wlp} \, (\text{SetPC} \, f \, p) \, Q \)

by (simp add: wp-def SetPC-split)

lemma wp-SetDC-split:
  \([ \forall s. x \, x \in s \, s \implies P \vdash wp \, (f \, x) \, Q; \forall s. \, s \neq \{\} \] \implies

P \vdash wp \, (SetDC \, f \, s) \, Q \)

by (rule le-fun\_I, unfold wp-eval, blast intro!: cInf-greatest)

lemma wlp-SetDC-split:
  \([ \forall s. x \, x \in s \, s \implies P \vdash wlp \, (f \, x) \, Q; \forall s. \, s \neq \{\} \] \implies

P \vdash wlp \, (SetDC \, f \, s) \, Q \)

by (rule le-fun\_I, unfold wp-eval, blast intro!: cInf-greatest)

lemma wp-SetDC:
  assumes wp: \( \forall s. x \, x \in s \, s \implies P \, x \vdash wp \, (f \, x) \, Q \)
  and nc: \( \forall s. \, s \neq \{\} \)
  and \( \text{sp}: \forall x. \text{sound} \, (P \, x) \)
  shows \(((\lambda x. \text{Inf} \, ((\lambda x. \, P \, x \, s) \cdot S \, s)) \vdash wp \, (SetDC \, f \, s) \, Q) \)

using assms by (intro le-fun\_I, simp add: wp-eval, blast intro!: cInf-mono)

lemma wlp-SetDC:
  assumes wp: \( \forall s. x \, x \in s \, s \implies P \, x \vdash wlp \, (f \, x) \, Q \)
  and nc: \( \forall s. \, s \neq \{\} \)
  and \( \text{sp}: \forall x. \text{sound} \, (P \, x) \)
shows \((\lambda s. \text{Inf} ((\lambda x. P \ x \ s) \ S \ s)) \vdash \text{wlp} (\text{SetDC} f S) \ Q\)

using \text{assms by}(\text{intro le-funI, simp add:wp-eval, blast intro!:cInf-mono)}

lemma \text{wp-Embed}:
\[
P \vdash t \ Q \Rightarrow P \vdash \text{wp} (\text{Embed} \ t) \ Q
\]
by \((\text{simp add:wp-def Embed-def})\)

lemma \text{wlp-Embed}:
\[
P \vdash t \ Q \Rightarrow P \vdash \text{wlp} (\text{Embed} \ t) \ Q
\]
by \((\text{simp add:wlp-def Embed-def})\)

lemma \text{wp-Bind}:
\[
[ \forall s. P \ s \vdash \text{wp} (a \ (f \ s)) \ Q \ s ] \Rightarrow P \vdash \text{wp} (\text{Bind} \ f \ a) \ Q
\]
by \((\text{auto simp:wp-def Bind-def})\)

lemma \text{wlp-Bind}:
\[
[ \forall s. P \ s \vdash \text{wlp} (a \ (f \ s)) \ Q \ s ] \Rightarrow P \vdash \text{wlp} (\text{Bind} \ f \ a) \ Q
\]
by \((\text{auto simp:wlp-def Bind-def})\)

lemma \text{wp-repeat}:
\[
[ P \vdash \text{wp} a \ Q; Q \vdash \text{wp} (\text{repeat} \ n \ a) \ R; \ 
\text{well-def} \ a; \text{sound} \ Q; \text{sound} \ R ] \Rightarrow P \vdash \text{wp} (\text{repeat} \ (\text{Suc} \ n) \ a) \ R
\]
by \((\text{auto intro!:wp-Seq wd-intros})\)

lemma \text{wlp-repeat}:
\[
[ P \vdash \text{wlp} a \ Q; Q \vdash \text{wlp} (\text{repeat} \ n \ a) \ R; \ 
\text{well-def} \ a; \text{unitary} \ Q; \text{unitary} \ R ] \Rightarrow P \vdash \text{wlp} (\text{repeat} \ (\text{Suc} \ n) \ a) \ R
\]
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 \text{wp-strengthen-post}=
\text{entails-strengthen-post}[\text{where } t=\text{wp} \ a \ \text{for} \ a]

lemma \text{wlp-strengthen-post}:
\[
P \vdash \text{wlp} a \ Q \Rightarrow \text{nearly-healthy} (\text{wlp} a) \Rightarrow \text{unitary} \ R \Rightarrow Q \vdash R \Rightarrow \text{unitary}
\]
Q \Rightarrow
\]
by \((\text{blast intro:entails-trans})\)

lemmas \text{wp-weaken-pre}=
\text{entails-weaken-pre}[\text{where } t=\text{wp} \ a \ \text{for} \ a]

lemmas \text{wlp-weaken-pre}=
\text{entails-weaken-pre}[\text{where } t=\text{wlp} \ a \ \text{for} \ a]

lemmas \text{wp-scale}=
4.11. LOOP TERMINATION

entails-scale[where $t=wp\ a$ for $a$, OF - well-def-wp-healthy]

4.10.2 Algebraic Decomposition

Refinement is a powerful tool for decomposition, belied by the simplicity of the rule. This is an axiomatic formulation of refinement (all annotations of the $a$ are annotations of $b$), rather than an operational version (all traces of $b$ are traces of $a$).

lemma wp-refines:
\[ [ a \sqsubseteq b; P \not\vdash wp\ a\ Q; sound\ Q ] \Rightarrow P \vdash wp\ b\ Q \]
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 :: $(\forall a:\ real) \Rightarrow (\forall prog \Rightarrow (\forall a:\ real) \Rightarrow bool (\{\|\} - \{\|\})p)$
where
wp-valid P prog Q $\equiv$ $P \vdash \vdash wp\ prog\ Q$

lemma wp-validI:
$P \vdash wp\ prog\ Q \Rightarrow \{P\} prog \{Q\}p$
unfolding wp-valid-def by(assumption)

lemma wp-validD:
$\{P\} prog \{Q\}p \Rightarrow P \not\vdash wp\ prog\ Q$
unfolding wp-valid-def by(assumption)

lemma valid-Seq:
\[ [ \{P\} a \{Q\}p; \{Q\} b \{R\}p; well-def\ a; well-def\ b; sound\ Q; sound\ R ] \Rightarrow \{P\} a :: b \{R\}p \]
unfolding wp-valid-def by(rule wp-Seq)

We make it available to the computational reasoner:

declare valid-Seq[trans]

end

4.11 Loop Termination

theory Termination imports Embedding StructuredReasoning Loops begin
Termination for loops can be shown by classical means (using a variant, or a measure function), or by probabilistic means: We only need that the loop terminates with probability one.

### 4.11.1 Trivial Termination

A maximal transformer (program) doesn’t affect termination. This is essentially saying that such a program doesn’t abort (or diverge).

**Lemma** `maximal-Seq-term`:

`fixes r::'s prog and s::'s prog
assumes mr: maximal (wp r)
and ws: well-def s
and ts: (λs. 1) ⊢ wp s (λs. 1)
shows (λs. 1) ⊢ wp (r ;; s) (λs. 1)
proof –
  note hs = well-def-wp-healthy[OF ws]
  have wp s (λs. 1) = (λs. 1)
  proof (rule antisym)
    show (λs. 1) ⊢ wp s (λs. 1) by(rule ts)
    have bounded-by 1 (wp s (λs. 1))
      by(auto intro!:healthy-bounded-byD[OF hs])
    thus wp s (λs. 1) ⊬ (λs. 1) by(auto intro!:le-funI)
  qed
with mr show ?thesis
by(simp add:wp-eval embed-bool-def maximalD)
qed

From any state where the guard does not hold, a loop terminates in a single step.

**Lemma** `term-onestep`:

`fixes wb: well-def body
shows «N G» ⊢ wp do G → body od (λs. 1)
proof(rule le-funI)
  note hb = well-def-wp-healthy[OF wb]
  fix s
  show «N G» s ≤ wp do G → body od (λs. 1) s
  proof(cases G s, simp-all add:wp-loop-nguard hb)
    from hb have sound (wp do G → body od (λs. 1))
      by(auto intro!:healthy-sound[OF healthy-wp-loop])
    thus 0 ≤ wp do G → body od (λs. 1) s by(auto)
  qed
qed

### 4.11.2 Classical Termination

The first non-trivial termination result is quite standard: If we can provide a natural-number-valued measure, that decreases on every iteration, and implies termination on reaching zero, the loop terminates.


lemma loop-term-nat-measure-noinv:

fixes m :: 's ⇒ nat and body :: 's prog
assumes wb: well-def body
and guard: \( \forall s, m \cdot s = 0 \rightarrow \neg G s \)
and variant: \( \forall n. (\forall s \cdot m s = Suc n) \vdash wp body (\forall s \cdot m s = n) \)
shows \( \forall s. 1 \vdash wp do G \rightarrow body od (\forall s. 1) s \)

proof

note hb = well-def-wp-healthy[OF wb]

have \( \forall n. (\forall s \cdot m s = n \rightarrow 1 \leq wp do G \rightarrow body od (\forall s. 1) s) \)

proof (induct-tac n)

fix n

show \( \forall s. m s = 0 \rightarrow 1 \leq wp do G \rightarrow body od (\forall s. 1) s \)

proof (clarify)

fix s

assume \( m s = 0 \)

with guard have \( \neg G s \) by (blast)

with hb show \( 1 \leq wp do G \rightarrow body od (\forall s. 1) s \)

by (simp add: wp-loop-nguard)

qed

assume \( IH: \forall s. m s = n \rightarrow 1 \leq wp do G \rightarrow body od (\forall s. 1) s \)

hence \( IH': \forall s. m s = n \rightarrow 1 \leq wp do G \rightarrow body od (\forall s. 1) s \)

by (simp add: embed-booldef)

have \( \forall s. m s = Suc n \rightarrow 1 \leq wp do G \rightarrow body od (\forall s. 1) s \)

proof (intro fold-premise healthy-intros hb, rule le-funI)

fix s

show \( (\forall s. m s = Suc n) \cdot s \leq wp do G \rightarrow body od (\forall s. 1) s \)

proof (cases G s)

case False

hence \( 1 \equiv \{ \forall s. G s \} s \) by (auto)

also from wb have \( \ldots \leq wp do G \rightarrow body od (\forall s. 1) s \)

by (rule le-funD[OF term-onestep])

finally show \( \text{thesis} \) by (simp add: embed-booldef)

next

case True note \( G = \text{this} \)

from \( IH' \) have \( (\forall s. m s = n) \vdash wp do G \rightarrow body od (\forall s. 1) s \)

by (blast intro:use-premise healthy-intros hb)

with variant wb

have \( (\forall s. m s = Suc n) \vdash wp (\text{body} ; ; do G \rightarrow body od) (\forall s. True) s \)

by (blast intro: wp-Seq wd-intros)

hence \( (\forall s. m s = Suc n) \cdot s \leq wp (\text{body} ; ; do G \rightarrow body od) (\forall s. True) s \)

by (auto)

also from \( hb \) have \( \ldots = wp do G \rightarrow body od (\forall s. True) s \)

by (simp add: wp-loop-guard)

finally show \( \text{thesis} \) .

qed

qed

thus \( \forall s. m s = Suc n \rightarrow 1 \leq wp do G \rightarrow body od (\forall s. 1) s \)

by (simp add: embed-booldef)

qed
thus \texttt{thesis by\texttt{(auto)}}

\texttt{qed}

This version allows progress to depend on an invariant. Termination is then
determined by the invariant’s value in the initial state.

\textbf{lemma loop-term-nat-measure:}

\texttt{fixes m :: 's ⇒ nat and body :: 's prog}

\texttt{assumes \texttt{wb: well-def body}}

\texttt{and \texttt{guard: \(\forall s. m s = 0 \rightarrow \neg G s\)}}

\texttt{and \texttt{variant: \(\forall n. \langle \lambda s. m s = \text{Suc } n \rangle \& \& \langle I \rangle \vdash wp body \langle \lambda s. m s = n \rangle\)}}

\texttt{and \texttt{inv: wp-inv body \langle I \rangle}}

\texttt{shows \(\langle I \rangle \vdash wp do G \rightarrow body od (\langle \lambda s. \text{True} \rangle \langle I \rangle)\)}}

\texttt{proof –}

\texttt{note \texttt{hb = well-def-wp-healthy[OF wb]}}

\texttt{note \texttt{scb = sublinear-sub-conj[OF well-def-wp-sublinear, OF wb]}}

\texttt{have \(\langle I \rangle \vdash wp do G \rightarrow body od (\langle \lambda s. \text{True} \rangle \langle I \rangle)\)}}

\texttt{proof\texttt{(rule use-premise, intro healthy-intros hb)}}

\texttt{fix s}

\texttt{have \(\forall n. \langle \forall s. m s = n \& \& I s \rightarrow 1 \leq wp do G \rightarrow body od (\langle \lambda s. \text{True} \rangle \langle I \rangle) \rangle\)}}

\texttt{proof\texttt{(clarify)}}

\texttt{fix s}

\texttt{assume \texttt{m s = 0}}

\texttt{with \texttt{guard have \(\neg G s\) by\texttt{(blast)}}

\texttt{with \texttt{hb show \(I \leq wp do G \rightarrow body od (\langle \lambda s. \text{True} \rangle \langle I \rangle)\)}}

\texttt{by\texttt{(simp add\texttt{: wp-loop-nguard)}}

\texttt{qed}

\texttt{assume \texttt{IH: \(\forall s. m s = n \& \& I s \rightarrow 1 \leq wp do G \rightarrow body od (\langle \lambda s. \text{True} \rangle \langle I \rangle)\)}}

\texttt{show \(\langle \lambda s. m s = \text{Suc } n \& \& I s \rangle s \leq wp do G \rightarrow body od (\langle \lambda s. \text{True} \rangle \langle I \rangle)\)}}

\texttt{proof\texttt{(rule fold-premise healthy-intros hb le-fun)}}

\texttt{fix s}

\texttt{show \(\langle \lambda s. \text{Suc } n \& \& I s \rangle s \leq wp do G \rightarrow body od (\langle \lambda s. \text{True} \rangle \langle I \rangle)\)}}

\texttt{proof\texttt{(cases G s)}}

\texttt{case False with \texttt{hb show \(\text{thesis}\)}}

\texttt{by\texttt{(simp add\texttt{: wp-loop-nguard)}}

\texttt{next}

\texttt{case True note \texttt{G = this}}

\texttt{have \(\langle \lambda s. m s = \text{Suc } n \rangle \& \& \langle I \rangle \& \& \langle G \rangle = \langle \lambda s. m s = \text{Suc } n \rangle \& \& \langle I \rangle \& \& \langle G \rangle \)}

\texttt{by\texttt{(simp)}}

\texttt{also have \(\text{... = (\langle \lambda s. m s = \text{Suc } n \rangle \& \& \langle I \rangle \rangle \& \& \langle G \rangle)}\)}}

\texttt{by\texttt{(simp add\texttt{: exp-conj-assoc exp-conj-unitary del\texttt{-exp-conj-ident)}}}

\texttt{also have \(\text{... = (\langle \lambda s. m s = \text{Suc } n \rangle \& \& \langle I \rangle \rangle \& \& \langle G \rangle \& \& \langle I \rangle)}\)}}

\texttt{by\texttt{(simp only\texttt{: exp-conj-comm)}}}

\texttt{also \{ \}}

\texttt{from \texttt{inv hb have \(\langle G \rangle \& \& \langle I \rangle \vdash wp body \langle I \rangle\)}}

\texttt{by\texttt{(rule wp-inv-stdD)}}
4.11. LOOP TERMINATION

with variant
have \( (\lambda s. \text{m s} = \text{Suc } n) \& \& (\text{I s}) \& \& (\text{G s} \& \& (\text{I s})) \) \vdash
wp body \( (\lambda s. \text{m s} = n) \& \& wp body (\text{I s}) \)
by (rule entails-frame)
\}
also from \( \text{s cb} \)
have \( wp body (\lambda s. \text{m s} = n) \& \& wp body (\text{I s}) \)
by (blast)
finally have \( (\lambda s. \text{m s} = \text{Suc } n) \& \& (\text{I s}) \& \& (\text{G s} \& \& (\text{I s})) \) \vdash
wp body \( (\lambda s. \text{m s} = n) \& \& (\text{I s}) \).
moreover \{ from \( \text{IH} \) have \( (\lambda s. \text{m s} = n \land (\text{I s}) s \vdash wp do G \rightarrow body od (\lambda s. \text{True}) \)
by (blast intro: use-premise healthy-intros \( \text{hh} \) )
hence \( (\lambda s. \text{m s} = n) \& \& (\text{I s}) \) \vdash wp do G \rightarrow body od (\lambda s. \text{True})
by (simp add: exp-conj-std-split)
\}
ultimately
have \( (\lambda s. \text{m s} = \text{Suc } n) \& \& (\text{I s}) \& \& (\text{G s}) \) \vdash
wp \( (\text{body} \&\& \text{do G \rightarrow body od} (\lambda s. \text{True}) s \)
using \( \text{wb} \) by (blast intro: wp-Seq \( \text{wd-intros} \) \( \text{hh} \) )
hence \( (\lambda s. \text{m s} = \text{Suc } n \land (\text{I s}) \& \& (\text{G s}) s \leq
wp (\text{body} \&\& \text{do G \rightarrow body od} (\lambda s. \text{True}) s \)
by (auto simp add: exp-conj-std-split)
with \( \text{G} \) have \( (\lambda s. \text{m s} = \text{Suc } n \land (\text{I s}) \& \& (\text{G s}) \leq
wp (\text{body} \&\& \text{do G \rightarrow body od} (\lambda s. \text{True}) s \)
by (simp add: exp-conj-def)
also from \( \text{hh G} \) have \( \ldots \) \( \vdash wp do G \rightarrow body od (\lambda s. \text{True}) s \)
by (simp add: wp-loop-guard)
finally show \( \text{thesis} \).
qed
qed
qed
moreover assume \( \text{I s} \)
ultimately show \( \text{I} \leq wp do G \rightarrow body od (\lambda s. \text{True}) s \)
by (auto)
qed
thus \( \text{thesis} \) 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 \( \text{wb: well-def body} \)
--- The loop terminates in one step with nonzero probability
and onestep: \( (\lambda s. \text{p}) \vdash wp body \in s \text{ G} \)
and \( nzp \): \( 0 < p \)
— The body is maximal i.e. it terminates absolutely.

and \( mb \): maximal \((wp \ body)\)

shows \( \lambda s. \ 1 \vdash wp \ do \ G \longrightarrow \ body \ od \ (\lambda s. \ 1) \)

proof —

note \( hb = \) well-def-wp-healthy\([OF \ wb]\)

note \( sb = \) healthy-scaling\([OF \ hb]\)

note \( sab = \) sublinear-subadd\([OF \ well-def-wp-sublinear, \ OF \ wb, \ OF \ healthy-feasibleD, \ OF \ hb]\)

from \( hb \) have \( hloop: \) healthy \((wp \ do \ G \longrightarrow \ body \ od)\) by \((rule \ healthy-intros)\)

hence \( swp: \) sound \((wp \ do \ G \longrightarrow \ body \ od \ (\lambda s. \ 1))\) by \((blast)\)

\( p \) is no greater than \( 1 \), by feasibility.

from \( onestep \) have \( onestep': \lambda s. \ p \leq \ wp \ body \ «N G» \ s \ by(auto) \)

also \{ 
  from \( hb \) have \( unitary \ (wp \ body \ «N G») \) by \((auto)\)
  hence \( \lambda s. \ wp \ body \ «N G» \ s \leq 1 \) by \((auto)\)
\}

finally have \( p1: \ p \leq 1 \).

This is the crux of the proof: that given a lower bound below \( 1 \), we can find another, higher one.

have \( new-bound: \lambda k. \ 0 \leq k \Rightarrow k \leq 1 \Rightarrow (\lambda s. \ k) \vdash wp \ do \ G \longrightarrow \ body \ od \ (\lambda s. \ 1) \ \Rightarrow \)

\( (\lambda s. \ p \ast (1-k) + k) \vdash wp \ do \ G \longrightarrow \ body \ od \ (\lambda s. \ 1) \)

proof \((rule \ le-funI)\)

fix \( k \ s \)

assume \( X: \lambda s. \ k \vdash wp \ do \ G \longrightarrow \ body \ od \ (\lambda s. \ 1) \)

and \( k0: \ 0 \leq k \ and \ k1: \ k \leq 1 \)

from \( k1 \) have \( nz1k: \ 0 \leq 1 - k \ by(auto) \)

with \( p1 \) have \( p \ast (1-k) + k \leq 1 \ast (1-k) + k \)

by \((blast \ intro:mult-right-mono \ add-mono)\)

hence \( p \ast (1-k) + k \leq 1 \)

by \((simp)\)

The new bound is \( p \ast (1-k) + k \).

hence \( p \ast (1-k) + k \leq «N G» \ s + «G» \ s \ast (p \ast (1-k) + k) \)

by \((cases \ G \ s, \ simp-all)\)

By the one-step termination assumption:

also from \( onestep' \ nz1k \)

have \( \ldots \leq «N G» \ s + «G» \ s \ast (wp \ body \ «N G» \ s \ast (1-k) + k) \)

by \((simp \ add: \ mult-right-mono \ ordered-comm-semiring-class.\ comm-mult-left-mono)\)

By scaling:

also from \( nz1k \)
Lastly, by folding two loop iterations:

\[
\begin{align*}
 & \text{have } \ldots = \langle N \rangle s + \langle G \rangle s \ast (wp \text{ body } (\lambda s. \langle N \rangle s \ast (1-k)) s + k) \\
& \quad \text{by (simp add: right-scalingD[OF sb])}
\end{align*}
\]

By the maximality (termination) of the loop body:

\[
\begin{align*}
 & \text{also from } mb k0 \\
 & \text{have } \ldots = \langle N \rangle s + \langle G \rangle s \ast (wp \text{ body } (\lambda s. \langle N \rangle s \ast (1-k)) s + wp \text{ body } (\lambda s. k) s) \\
& \quad \text{by (simp add: maximalD)}
\end{align*}
\]

By sub-additivity of the loop body:

\[
\begin{align*}
 & \text{also from } k0 \text{ nz}1k \\
 & \text{have } \ldots \leq \langle N \rangle s + \langle G \rangle s \ast (wp \text{ body } (\lambda s. \langle N \rangle s + \langle G \rangle s \ast k) s) \\
& \quad \text{by (auto intro!: add-left-mono mult-left-mono sub-addD[OF sb] sound-intros)} \\
& \quad \text{also have } \ldots = \langle N \rangle s + \langle G \rangle s \ast (wp \text{ body } (\lambda s. \langle N \rangle s + \langle G \rangle s \ast k) s) \\
& \quad \quad \text{by (simp add: negate-embed algebra-simps)}
\end{align*}
\]

By monotonicity of the loop body, and that \( k \) is a lower bound:

\[
\begin{align*}
 & \text{also from } k0 \text{ hloop le-fanD[OF X]} \\
 & \text{have } \ldots \leq \langle N \rangle s + \langle G \rangle s \ast (wp \text{ body } (\lambda s. \langle N \rangle s \ast wp \text{ do } G \longrightarrow body \text{ od } (\lambda s. 1) s) s) \\
& \quad \text{by (iprover intro: add-left-mono mult-left-mono le-fanI embed-ge-0)} \\
& \quad \quad \text{le-fanD[OF mono-transD, OF healthy-monoD, OF hb]} \\
& \quad \quad \quad \text{sound-sum standard sound-intros wpv}
\end{align*}
\]

Unrolling the loop once and simplifying:

\[
\begin{align*}
 & \text{also}\{ \\
 & \quad \text{have } \bigwedge s. \langle N \rangle s + \langle G \rangle s \ast wp \text{ body } (wp \text{ do } G \longrightarrow body \text{ od } (\lambda s. 1)) s = \\
& \quad \quad \langle N \rangle s \ast \langle G \rangle s \ast (\langle N \rangle s + \langle G \rangle s \ast wp \text{ body } (wp \text{ do } G \longrightarrow body \text{ od } (\lambda s. 1)) s) \\
& \quad \quad \text{by (simp only: distrib-left mult.assoc[symmetric] embed-bool-idem embed-bool-cancel)} \\
& \quad \quad \text{also have } \bigwedge s. ... s = \langle N \rangle s + \langle G \rangle s \ast wp \text{ do } G \longrightarrow body \text{ od } (\lambda s. 1) s \\
& \quad \quad \quad \text{by (simp add: fun-cong[OF wp-loop-unfold[symmetric, where P=\lambda s. 1, simplified, OF hb]])} \\
& \quad \text{finally have } X: \bigwedge s. \langle N \rangle s + \langle G \rangle s \ast wp \text{ body } (wp \text{ do } G \longrightarrow body \text{ od } (\lambda s. 1)) s = \\
& \quad \quad \langle N \rangle s + \langle G \rangle s \ast wp \text{ do } G \longrightarrow body \text{ od } (\lambda s. 1) s. \\
& \quad \text{have } \langle N \rangle s \ast \langle G \rangle s \ast (wp \text{ body } (\lambda s. \langle N \rangle s + \langle G \rangle s \ast wp \text{ do } G \longrightarrow body \text{ od } (\lambda s. 1)) s) = \\
& \quad \quad \langle N \rangle s + \langle G \rangle s \ast (wp \text{ body } (\lambda s. \langle N \rangle s + \langle G \rangle s \ast wp \text{ body } (wp \text{ do } G \longrightarrow body \text{ od } (\lambda s. 1)) s) s) \\
& \quad \quad \text{by (simp only:X)}
\end{align*}
\]

Lastly, by folding two loop iterations:

\[
\begin{align*}
 & \text{also have } \langle N \rangle s + \langle G \rangle s \ast (wp \text{ body } (\lambda s. \langle N \rangle s + \langle G \rangle s \ast wp \text{ body } (wp \text{ do } G \longrightarrow body \text{ od } (\lambda s. 1)) s) s) =
\end{align*}
\]
\[
\text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s
\]

by\((\text{simp add: wp-loop-unfold[OF - hb], where } P = \lambda s. 1, \text{ simplified, symmetric}] \)

fun-cong[OF wp-loop-unfold[OF - hb], where \(P = \lambda s. 1\), simplified, symmetric]]

finally show \(p * (1 - k) + k \leq \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s\).

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.

\[
\text{from nzp have inc: } \forall k. 0 \leq k \Rightarrow k < 1 \Rightarrow k < p * (1 - k) + k
\]

by\((auto intro: mult-pos-pos)\)

The result follows by contradiction.

\[
\text{show } ?\text{thesis}
\]

proof\((\text{rule ccontr})\)

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 \(\neg ?\text{thesis}\)

hence \(\neg (\forall s. 1 \leq \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s)\) by\((auto)\)

then obtain \(s\) where point: \(\neg 1 \leq \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s\) by\((auto)\)

let \(?k = \text{Inf (range (wp do } G \rightarrow \text{body od } (\lambda s. 1)))\)

from bloop

have Inf\(lb\): \(\forall s. ?k \leq \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s\)

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 \(k1: ?k < 1\).

hence \(?k \leq 1\) by\((auto)\)

moreover from bloop have \(k0: 0 \leq ?k\)

by\((intro cInf-greatest, auto)\)

The infimum is, naturally, a lower bound.

moreover from Inf\(lb\) have \((\lambda s. ?k) \vdash wp do G \rightarrow \text{body od } (\lambda s. 1)\) by\((auto)\)

ultimately

We can therefore use the previous result to find a new bound, ...

have \(\forall s. p * (1 - ?k) + ?k \leq \text{wp do } G \rightarrow \text{body od } (\lambda s. 1) s\)

by\((\text{blast intro: le-funD[OF new-bound]})\)

...which is lower than the infimum, by minimality, ...

hence \(p * (1 - ?k) + ?k \leq ?k\)

by\((\text{blast intro: cInf-greatest})\)

...yet also strictly greater than it.
moreover from \( k0 \) \( k1 \) have \( ?k < p \ast (1 - ?k) + ?k \) by(rule inc)

We thus have a contradiction.

ultimately show False by(simp)

qed

end

4.12 Automated Reasoning

theory Automation imports StructuredReasoning
begin

This theory serves as a container for automated reasoning tactics for pGCL, implemented in ML. At present, there is a basic verification condition generator (VCG).

named-theorems wd

theorems to automatically establish well-definedness

named-theorems pwp-core

core probabilistic wp rules, for evaluating primitive terms

named-theorems pwp

user-supplied probabilistic wp rules

named-theorems wlp

user-supplied probabilistic wlp rules

ML-file ⟨pVCG.ML⟩

method-setup pvcg =

⟨Scan.succeed (fn ctxt => SIMPLE-METHOD' (pVCG.pVCG-tac ctxt))⟩

Probabilistic weakest preexpectation tactic

declare wd-intros[wd]

lemmas core-wp-rules =

wp-Skip wlp-Skip
wp-Abort wlp-Abort
wp-Apply wlp-Apply
wp-Seq wlp-Seq
wp-DC-split wlp-DC-split
wp-PC-fixed wlp-PC-fixed
wp-SetDC wlp-SetDC
wp-SetDC-split wlp-SetDC-split

declare core-wp-rules[pwp-core]

end
Additional Material

4.13 Miscellaneous Mathematics

theory Misc
imports 
  HOL–Analysis.Multivariate-Analysis
begin

lemma sum-UNIV:
  fixes S :: 'a::finite set
  assumes complete: \( \forall x. x \notin S \implies f x = 0 \)
  shows \( \sum f S = \sum f \text{UNIV} \)

proof –
  from complete have \( \sum f S = \sum f (\text{UNIV} - S) + \sum f S \)
  by (simp)
  also have \( \ldots = \sum f \text{UNIV} \)
  by (auto intro: sum.subset-diff[symmetric])
  finally show ?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: \( a \leq b \)
  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 \( c \cdot \text{max} a b = \text{max} (c \cdot a) (c \cdot b) \)

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 ?thesis by (simp add:max.absorb2)
next
case False then have b ≤ a by(auto)
moreover with nn have c * b ≤ c * a by(auto intro:mult-left-mono)
ultimately show ?thesis by(simp add:max.absorb1)
qed

lemma mult-div-mono-left:
  fixes c::real
  assumes nnc: 0 ≤ c and nzc: c ≠ 0
  and inv: a ≤ inverse c * b
  shows c * a ≤ b
proof –
  from nnc inv have c * a ≤ (c * inverse c) * b
    by(auto simp:mult.assoc intro:mult-left-mono)
  also from nzc have ... = b by(simp)
  finally show c * a ≤ b.
qed

lemma mult-div-mono-right:
  fixes c::real
  assumes nnc: 0 ≤ c and nzc: c ≠ 0
  and inv: inverse c * a ≤ b
  shows a ≤ c * b
proof –
  from nzc have a = (c * inverse c) * a by(simp)
  also from nnc inv have (c * inverse c) * a ≤ c * b
    by(auto simp:mult.assoc intro:mult-left-mono)
  finally show a ≤ c * b.
qed

lemma min-distrib:
  fixes c::real
  assumes nnc: 0 ≤ c
  shows c * min a b = min (c * a) (c * b)
proof(cases a ≤ b)
case True moreover with nnc have c * a ≤ c * b
  by(blast intro:mult-left-mono)
  ultimately show ?thesis by(auto)
next
case False hence b ≤ a by(auto)
moreover with nnc have c * b ≤ c * a
  by(blast intro:mult-left-mono)
ultimately show ?thesis by(simp add:min.absorb2)
qed

lemma finite-set-least:
  fixes S::'a::linorder set
  assumes finite: finite S
  and ne: S ≠ {}
4.13. MISCELLANEOUS MATHEMATICS

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 \( a \) 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 True then show \( \text{thesis} \) by (auto)

next

  case False with \( IH \) have \( \exists x \in S. \forall y \in S. \ x \leq y \) by (auto)

  then obtain \( z \) where \( z \in S \) and \( z \leq y \in S \) by (auto)

  thus \( \text{thesis} \) by (cases \( z \leq x \), auto)

qed

with \( ne \) show \( \text{thesis} \) by (auto)

qed

lemma \( cSup-add \):

fixes \( c::real \)

assumes \( ne: S \neq \{\} \)

  and \( bS: \forall x. \ x \in S \Rightarrow x \leq b \)

shows \( \sup S + c = \sup \{ x + c | x \in S \} \)

proof (rule antsym)

  from \( ne \) \( bS \) show \( \sup \{ x + c | x \in S \} \leq \sup S + c \)

  by (auto intro!:cSup-least add-right_mono cSup-upper bdd-aboveI)

  have \( \sup S \leq \sup \{ x + c | x \in S \} - c \)

  proof (intro cSup-least ne)

    fix \( x \) assume \( zin: x \in S \)

    from \( bS \) have \( \forall x. \ x \in S \Rightarrow x + c \leq b + c \) by (auto intro:add-right_mono)

    hence \( bdd-above \{ x + c | x \in S \} \) by (intro bdd-aboveI, blast)

    with \( zin \) have \( x + c \leq \sup \{ x + c | x \in S \} \) by (auto intro:cSup-upper)

    thus \( x \leq \sup \{ x + c | x \in S \} - c \) by (auto)

    qed

  thus \( \sup S + c \leq \sup \{ x + c | x \in S \} \) by (auto)

  qed

lemma \( cSup-mult \):

fixes \( c::real \)

assumes \( ne: S \neq \{\} \)

  and \( bS: \forall x. \ x \in S \Rightarrow x \leq b \)

  and \( nnc: 0 \leq c \)

shows \( c \cdot \sup S = \sup \{ c \cdot x | x \in S \} \)

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 \( \text{thesis} \)
proof (rule antisym)
  from \( bS \) have \( baS: \text{bdd-above } S \) by (intro bdd-aboveI, auto)
  with \( ne \) nnc show \( \sup \{ c \times x | x \in S \} \leq c \times \sup S \)
    by (blast intro : cSup-least mult-left-mono[OF cSup-upper])
have \( \sup S \leq \frac{1}{c} \sup \{ c \times x | x \in S \} \)
proof (intro cSup-least ne)
  fix \( x \) assume \( x \in S \)
  moreover from \( bS \) \( \forall x \in S. x \leq B \)
ultimately have \( c \times x \leq \sup \{ c \times x | x \in S \} \)
by (auto intro : mult-left-mono)
have \( \sup S \leq \frac{1}{c} \sup \{ c \times x | x \in S \} \)
proof (intro cSup-least ne)
  fix \( x \) assume \( x \in S \)
  moreover from \( nnc \) have \( 0 \leq \frac{1}{c} \)
ultimately have \( \frac{1}{c} \times c \times x \leq \sup \{ c \times x | x \in S \} \)
by (auto intro : mult-left-mono)
have \( \frac{1}{c} \times \frac{1}{c} \times x \leq \sup \{ c \times x | x \in S \} \)
by (simp add : mult.assoc[symmetric])
qed
with \( nnc \) have \( c \times \sup S \leq c \times (\sup \{ c \times x | x \in S \}) \)
by (auto intro : mult-left-mono)
with \( cnz \) show \( c \times \sup S \leq c \times (\sup \{ c \times x | x \in S \}) \)
by (simp add : mult.assoc[symmetric])
qed

lemma closure-contains-Sup:
fixes \( S :: \text{real set} \)
assumes \( neS : S \neq \{ \} \) and \( bS : \forall x \in S. x \leq B \)
sows \( \sup S \in \text{closure } S \)
proof (rule antisym)
  let \( ?T = \text{uminus } ^\cdot S \)
  from \( neS \) have \( \forall x \in ?T. ?T \neq \{ \} \) by (auto)
  from \( bS \) have \( \exists x \in ?T. -B \leq x \) by (auto)
  hence \( \exists x \in ?T. -1 \times -x \leq -1 \times \inf (\text{uminus } ^\cdot S) \)
by (rule mult-left-mono-neg, auto)
  hence \( \inf \{ x \times S \} \leq x \leq \inf (\text{uminus } ^\cdot S) \)
by (simp)
  with \( neS \) \( bS \) show \( \sup S \leq - \inf \{ x \times S \} \)
by (blast intro : cSup-least)
  have \( - \sup S \leq \inf \{ x \times S \} \)
by (rule antisym)
  from \( neT \) \( \exists x \in S \Rightarrow \inf (\text{uminus } ^\cdot S) \leq -x \)
by (blast intro : cInf-lower)
  hence \( \exists x \in S \Rightarrow -1 \times -x \leq -1 \times \inf (\text{uminus } ^\cdot S) \)
by (rule mult-left-mono-neg, auto)
  hence \( \inf \{ x \times S \} \leq x \leq - \inf (\text{uminus } ^\cdot S) \)
by (simp)
  with \( neS \) \( bS \) show \( \sup S \leq - \inf (\sup \{ x \times S \}) \)
by (blast intro : cSup-least)
  have \( - \sup S \leq \inf (\sup \{ x \times S \}) \)
by (rule antisym)
proof\( (\text{rule cInf-greatest}[\text{OF neT}]) \)

fix \( x \) assume \( x \in \text{uminus} \cdot S \)
then obtain \( y \) where \( \text{yin}: y \in S \) and \( \text{rew}: x = -y \) by(\text{auto})
from \( \text{yin bS} \) have \( y \leq \text{Sup S} \)
by(\text{intro cSup-upper bdd-belowI, auto})

hence \(-1 \cdot \text{Sup S} \leq -1 \cdot y\)
with \( \text{rew show} \) \(-\text{Sup S} \leq x \) by(\text{simp})

qed

hence \(-1 \cdot \text{Inf } ?T \leq -1 \cdot (- \text{Sup S})\)
by(\text{simp add: mult-left-mono-neg})

thus \(-\text{Inf } ?T \leq \text{Sup S} \) by(\text{simp})

qed

also {
from \( \text{neT bbT} \) have \( \text{Inf } ?T \in \text{closure } ?T \)
by(\text{rule closure-contains-Inf})

hence \(-\text{Inf } ?T \in \text{uminus} \cdot \text{closure } ?T \) by(\text{auto})
}

also {
have \text{linear} unminus by(\text{auto intro: linearI})

hence \text{uminus} \cdot \text{closure } ?T \subseteq \text{closure } \text{uminus} \cdot ?T \)
by(\text{rule closure-linear-image-subset})
}

also {
have \text{uminus} \cdot ?T \subseteq S by(\text{auto})

hence \text{closure (uminus} \cdot ?T \subseteq \text{closure S by(rule closure-mono})
}

finally show \( \text{Sup S} \in \text{closure S} \).

qed

lemma \text{tendsto-min}:

fixes \( x \ y::\text{real} \)
assumes \( \text{ta}: a \longrightarrow x \)
and \( \text{tb}: b \longrightarrow y \)
shows \( (\lambda i. \text{min } (a \ i) \ (b \ i)) \longrightarrow \text{min } x \ y \)
proof(\text{rule LIMSEQ-I, simp})
fix \( e::\text{real} \) assume \( \text{pe}: 0 < e \)

from \( \text{ta pe obtain noa where balla: } \forall n \geq \text{noa}. \text{abs } (a \ n - x) < e \)
by(\text{auto dest:LIMSEQ-D})

from \( \text{tb pe obtain nob where ballb: } \forall n \geq \text{nob}. \text{abs } (b \ n - y) < e \)
by(\text{auto dest:LIMSEQ-D})

{
fix \( n \)
assume \( \text{ge}: \text{max } \text{noa} \ \text{nob} \leq n \)

hence \( \text{gea: } \text{noa} \leq n \) and \( \text{geb: } \text{nob} \leq n \) by(\text{auto})

have \( \text{abs } (\text{min } (a \ n) \ (b \ n) - \text{min } x \ y) < e \)
proof cases
assume \( \text{le: } \text{min } (a \ n) \ (b \ n) \leq \text{min } x \ y \)
show ?thesis
proof cases 
  assume a n ≤ b n
  hence rwmin: min (a n) (b n) = a n by(auto)
  with le have a n ≤ min x y by(simp)
  moreover from gea balla have abs (a n - x) < e by(simp)
  moreover have min x y ≤ x by(auto)
  ultimately have abs (a n - min x y) < e by(auto)
  with rwmin show abs (min (a n) (b n) - min x y) < e by(simp)
next 
  assume ¬ a n ≤ b n
  hence b n ≤ a n by(auto)
  with le have b n ≤ min x y by(simp)
  moreover from geb ballb have abs (b n - y) < e by(simp)
  moreover have min x y ≤ y by(auto)
  ultimately have abs (b n - min x y) < e by(auto)
  with rwmin show abs (min (a n) (b n) - min x y) < e by(simp)
qed
next 
  assume ¬ min (a n) (b n) ≤ min x y
  hence le: min x y ≤ min (a n) (b n) by(auto)
  show ?thesis
proof cases 
  assume x ≤ y
  hence rwmin: min x y = x by(auto)
  with le have x ≤ min (a n) (b n) by(simp)
  moreover from gea balla have abs (a n - y) < e by(simp)
  moreover have min (a n) (b n) ≤ a n by(auto)
  ultimately have abs (min (a n) (b n) - y) < e by(auto)
  with rwmin show abs (min (a n) (b n) - min x y) < e by(simp)
next 
  assume ¬ x ≤ y
  hence y ≤ x by(auto)
  hence rwmin: min x y = y by(auto)
  with le have y ≤ min (a n) (b n) by(simp)
  moreover from geb ballb have abs (b n - y) < e by(simp)
  moreover have min (a n) (b n) ≤ b n by(auto)
  ultimately have abs (min (a n) (b n) - y) < e by(auto)
  with rwmin show abs (min (a n) (b n) - min x y) < e by(simp)
qed
qed

} 
thus ∃ no. ∀ n≥no. |min (a n) (b n) - min x y| < e by(blast)
qed

definition supp :: ('s ⇒ real) ⇒ 's set
where supp f = {x. f x ≠ 0}
definition dist-remove :: ('s ⇒ real) ⇒ 's ⇒ 's ⇒ real
where dist-remove p x = (λy. if y=x then 0 else p y / (1 − p x))

lemma supp-dist-remove:
  p x ≠ 0 ⇒ p x ≠ 1 ⇒ supp (dist-remove p x) = supp p − {x}
  by (auto simp:dist-remove-def supp-def)

lemma supp-empty:
  supp f = {} ⇒ f x = 0
  by (simp add: supp-def)

lemma nsupp-zero:
  x /∈ supp f ⇒ f x = 0
  by (simp add: supp-def)

lemma sum-supp:
  fixes f :: 'a :: finite ⇒ real
  shows sum f (supp f) = sum f UNIV
proof
  have sum f (UNIV − supp f) = 0
    by (simp add: supp-def)
  hence sum f (supp f) = sum f (UNIV − supp f) + sum f (supp f)
    by (simp)
  also have ... = sum f UNIV
    by (simp add: sum.subset-diff[symmetric])
  finally show ?thesis.
qed

4.13.1 Truncated Subtraction

definition tminus :: real ⇒ real ⇒ real (infixl ⊖ 60)
where
  x ⊖ y = max (x − y) 0

lemma minus-le-tminus[intro!,simp]:
  a − b ≤ a ⊖ b
  unfolding tminus-def by (auto)

lemma tminus-cancel-1:
  0 ≤ a ⇒ a + 1 ⊖ 1 = a
  unfolding tminus-def by (simp)

lemma tminus-zero-imp-le:
  x ⊖ y ≤ 0 ⇒ x ≤ y
  by (simp add: tminus-def)

lemma tminus-zero[simp]:
  0 ≤ x ⇒ x ⊖ 0 = x
by (simp add: tminus-def)

lemma tminus-left-mono:
  \( a \leq b \implies a \ominus c \leq b \ominus c \)
unfolding tminus-def
by (case-tac a \leq c, simp-all)

lemma tminus-less:
  \[ \begin{array}{l}
    0 \leq a; 0 \leq b \\
  \end{array} \implies a \ominus b \leq a \]
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 \( a \ast \max (b - c) \Theta = 0 \) by (simp add: max.absorb2)
  also
    from nna le have \( a \ast b \leq a \ast c \) by (blast intro: mult-left-mono)
  hence \( 0 = \max (a \ast b - a \ast c) \Theta \) by (simp add: max.absorb1)
  \}
  finally show ?thesis by (simp add: tminus-def)
next
  case False
  hence le: \( c \leq b \) by (auto)
  hence \( a \ast \max (b - c) = a \ast (b - c) \) by (simp only: max.absorb1)
  also
    from nna le have \( a \ast c \leq a \ast b \) by (blast intro: mult-left-mono)
  hence \( 0 = \max (a \ast b - a \ast c) \Theta \) 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 (cases \( 0 \leq a - c \))
  case True
  note pac = this
show \( \text{thesis} \)
proof (cases \( 0 \leq b - d \))
case True note \( pbd = \text{this} \)
from \( \text{pac and pbd have} \ (c + d) \leq (a + b) \) by (simp)
with \( \text{pac and pbd show} \ \text{thesis} \) by (simp)
next
case False with \( \text{pac} \) show \( \text{thesis} \)
by (cases \( c + d \leq a + b \), auto)
qed
next
case False note \( nac = \text{this} \)
show \( \text{thesis} \)
proof (cases \( 0 \leq b - d \))
case True with \( nac \) show \( \text{thesis} \)
by (cases \( c + d \leq a + b \), auto)
next
case False note \( nbd = \text{this} \)
with \( nac \) have \( \neg (c + d) \leq (a + b) \) by (simp)
with \( nac \) and \( nbd \) show \( \text{thesis} \) by (simp)
qed
qed

lemma \( \text{tminus-sum-mono} \):
assumes \( fS : \text{finite} \ S \)
shows \( \sum f \ S \ominus \sum g \ S \leq \sum (\lambda x. f x \ominus g x) \ S \)
(is \( \exists X \ S \))
proof (rule finite-induct)
from \( fS \) show \( \text{finite} \ S \).
show \( \exists X \ \{ \} \) by (simp)
fix \( x \) and \( F \)
assume \( fF : \text{finite} \ F \) and \( x \notin F \)
and \( IH : \exists X F \)
have \( f x + \sum f \ F \ominus g x + \sum g \ F \leq \)
\( (f x \ominus g x) + (\sum f \ F \ominus \sum g \ F) \)
by (rule tminus-add-mono)
also from \( IH \) have \( \ldots \leq (f x \ominus g x) + (\sum x \in F. f x \ominus g x) \)
by (rule add-left-mono)
finally show \( \exists X (\text{insert} \ x \ F) \)
by (simp add: sum.insert[OF fF xniF])
qed

lemma \( \text{tminus-nneg} \)[simp,intro]:
\( 0 \leq a \ominus b \)
by (cases \( b \leq a \), auto)

lemma \( \text{tminus-right-antimono} \):
assumes \( clb : c \leq b \)
shows $a \ominus b \leq a \ominus c$
proof\( (\text{cases } b \leq a) \)
\begin{itemize}
  \item case True
    \begin{itemize}
      \item moreover with \( clb \) have \( c \leq a \) by(auto)
      \item moreover note \( clb \)
      \item ultimately show ?thesis by(simp)
    \end{itemize}
  \item case False then show ?thesis by(simp)
\end{itemize}
qed

lemma min-tminus-distrib:
\begin{align*}
  \min a b \ominus c & = \min (a \ominus c) (b \ominus c) \\
  \text{unfolding } \text{tminus-def} & \text{ by(auto)}
\end{align*}
end
Bibliography


