pGCL for Isabelle

David Cock

August 16, 2018
# Contents

1 Overview 1

2 Introduction to pGCL 3

2.1 Language Primitives ........................................... 3
  2.1.1 The Basics ........................................... 3
  2.1.2 Assertion and Annotation ................................. 4
  2.1.3 Probability ........................................... 4
  2.1.4 Nondeterminism ......................................... 5
  2.1.5 Properties of Expectations ............................... 5

2.2 Loops .......................................................... 6
  2.2.1 Guaranteed Termination ................................. 6
  2.2.2 Probabilistic Termination ............................... 8

2.3 The Monty Hall Problem ......................................... 9
  2.3.1 The State Space ....................................... 9
  2.3.2 The Game ........................................... 10
  2.3.3 A Brute Force Solution ................................ 11
  2.3.4 A Modular Approach .................................. 11

3 Semantic Structures 17

3.1 Expectations .................................................. 17
  3.1.1 Bounded Functions .................................... 18
  3.1.2 Non-Negative Functions .................... 20
  3.1.3 Sound Expectations .................................. 21
  3.1.4 Unitary expectations .................................. 25
  3.1.5 Standard Expectations ............................... 25
  3.1.6 Entailment ........................................... 27
  3.1.7 Expectation Conjunction .............................. 28
  3.1.8 Rules Involving Conjunction .......................... 29
  3.1.9 Rules Involving Entailment and Conjunction Together 31

3.2 Expectation Transformers ....................................... 33
  3.2.1 Comparing Transformers ............................... 36
  3.2.2 Healthy Transformers .................................. 39
  3.2.3 Sublinearity ........................................... 47
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 ...................... 122
4.4 Continuity and Induction for Loops ......................... 126
4.4.1 The Limit of Iterates .................................. 134
4.5 Sublinearity .................................................. 144
4.5.1 Nonrecursive Primitives ............................... 144
4.5.2 Sublinearity for Loops ................................. 150
4.6 Determinism .................................................. 154
4.6.1 Additivity ............................................... 154
4.6.2 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 ..................................... 176
4.8.4 Unfolding ............................................... 177
4.9 The Algebra of pGCL ......................................... 178
4.9.1 Program Refinement .................................... 178
4.9.2 Simple Identities ...................................... 179
4.9.3 Deterministic Programs are Maximal .................. 184
4.9.4 The Algebraic Structure of Refinement ................ 186
4.9.5 Data Refinement ........................................ 189
4.9.6 The Algebra of Data Refinement ....................... 193
4.9.7 Structural Rules for Correspondence ................... 200
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.9.8 Structural Rules for Data Refinement</td>
<td>201</td>
</tr>
<tr>
<td>4.10 Structured Reasoning</td>
<td>203</td>
</tr>
<tr>
<td>4.10.1 Syntactic Decomposition</td>
<td>204</td>
</tr>
<tr>
<td>4.10.2 Algebraic Decomposition</td>
<td>208</td>
</tr>
<tr>
<td>4.10.3 Hoare triples</td>
<td>209</td>
</tr>
<tr>
<td>4.11 Loop Termination</td>
<td>209</td>
</tr>
<tr>
<td>4.11.1 Trivial Termination</td>
<td>209</td>
</tr>
<tr>
<td>4.11.2 Classical Termination</td>
<td>210</td>
</tr>
<tr>
<td>4.11.3 Probabilistic Termination</td>
<td>213</td>
</tr>
<tr>
<td>4.12 Automated Reasoning</td>
<td>217</td>
</tr>
</tbody>
</table>

### Additional Material

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.13 Miscellaneous Mathematics</td>
<td>219</td>
</tr>
<tr>
<td>4.13.1 Truncated Subtraction</td>
<td>225</td>
</tr>
</tbody>
</table>
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:

\begin{verbatim}
datatype coin = Heads | Tails
record coins =
  a :: coin
  b :: coin
\end{verbatim}

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

\begin{verbatim}
lemma
  Apply (\lambda s. s (# a := Heads \rbrack)) = (a := (\lambda s. Heads))
  by (simp)
\end{verbatim}

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

### 2.1.2 Assertion and Annotation

Assertions in pGCL are real-valued functions of the state, which are often interpreted as a probability distribution over possible outcomes. These functions are termed *expectations*, for reasons which shortly be clear. Initially, however, we need only consider standard expectations: those derived from a binary predicate. A predicate $P: s \Rightarrow \text{bool}$ is embedded as $« P »: s \Rightarrow \text{real}$, such that $P s \rightarrow « P » s = 1 \land \neg P s \rightarrow « P » s = 0$. An annotation consists of an assertion on the initial state and one on the final state, which for standard expectations may be interpreted as ‘if $P$ holds in the initial state, then $Q$ will hold in the final state’. These are in weakest-precondition form: we assert that the precondition implies the weakest precondition: the weakest assertion on the initial state, which implies that the postcondition must hold on the final state. So far, this is identical to the standard approach. Remember, however, that we are working with real-valued assertions. For standard expectations, the logic is nevertheless identical, if the implication $\forall s. P s \rightarrow Q s$ is substituted with the equivalent expectation entailment $« P » \vdash « Q »$, $[ « ?P » \vdash « ?Q » ; ?P ?s] \implies ?Q ?s$. Thus a valid specification of $\text{Apply} (\lambda s. s(a := b s))$ is:

**lemma**

$$\forall x. « \lambda s. b s = x » \vdash wp (a := b) « \lambda s. a s = x »$$

by (pveq, simp add: o-def)

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

### 2.1.3 Probability

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

**lemma**

$$« \lambda s::\text{real}. s \neq 0 » \vdash wp (\text{Apply} ((\ast) 2) :: \text{Apply} (\lambda s. s / s)) « \lambda s. s = 1 »$$

by (pveq, simp add: o-def)

So far, we haven’t done anything that required probabilities, or expectations other than 0 and 1. As an example of both, we show that a single coin toss is fair. We introduce the syntax $x p \oplus y$ for a probabilistic choice between $x$ and $y$. This program behaves as $x$ with probability $p$, and as $y$ with probability $(1::'a) - p$. The probability may depend on the state, and is therefore of
2.1. LANGUAGE PRIMITIVES

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-head-Suc 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-head-Suc 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 (\x. 0 < x) → Apply (\s. 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 = (\x. 0 ≤ x)

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

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

  show well-def ?p
    by(rule ud-intros)
A measure of 0 implies termination.

  show \A 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.
show \( \forall n. (\forall x. \text{nat}(\max x 0)) = \text{Suc} n \) \&\& \( \text{inv-count} \) \leftarrow wp \ ?p \ (\forall x. \text{nat}(\max x 0)) = n \)

unfolding inv-count-def
by(pvcg,
auto simp: o-def exp-conj-std-split[symmetric]
intro: implies-entails)

qed

2.2.2 Probabilistic Termination

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

type-synonym coin = bool
definition Heads = True
definition Tails = False
definition flip :: coin prog
where
flip = Apply (\x. Heads) (\x. 1/2) \oplus Apply (\x. Tails)

We can’t define a measure here, as we did previously, as neither of the two possible states guarantees termination.
definition wait-for-heads :: coin prog
where
wait-for-heads = do ((\#) Heads) \rightarrow flip od

Nonetheless, we can show termination.
lemma wait-for-heads-term:
\( \forall s. 1 \rightarrow wp \ wait-for-heads \ (\forall s. 1) \)
unfolding wait-for-heads-def

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

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

We must show that the loop body is deterministic, meaning that it cannot diverge by itself.
show maximal (wp flip)
unfolding flip-def by(auto intro:max-intros)

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

\[ \text{show } \lambda s. \frac{1}{2} \vdash \text{wp flip \{N ((\neq) Heads)\}} \]
\[ \text{unfolding flip-def} \]
\[ \text{by(pvcg, simp add:o-def Heads-def Tails-def)} \]

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

\[ \text{show } (0::\text{real}) < \frac{1}{2} \text{ by(simp)} \]
\[ \text{qed} \]
\[ \text{end} \]

2.3 The Monty Hall Problem

theory Monty imports ../pGCL begin

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

The setting is a game show: There is a prize hidden behind one of three doors, and the contestant is invited to choose one. Once the guess is made, the host than opens one of the remaining two doors, revealing a goat and showing that the prize is elsewhere. The 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.

\[ \text{record game =} \]
\[ \text{prize :: nat} \]
\[ \text{guess :: nat} \]
\[ \text{clue :: nat} \]

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

\[ \text{definition player-wins :: game } \Rightarrow \text{ bool} \]
\[ \text{where player-wins } g \equiv \text{ guess } g = \text{ prize } g \]
Invariants

We prove explicitly that only valid doors are ever chosen.

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

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

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

\subsection{2.3.2 The Game}

Hide the prize behind door \(D\).

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

Choose door \(D\).

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

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

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

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

\textbf{definition}\ \textit{hide-prize} :: \textit{game} \textit{prog}
\textbf{where}\ \textit{hide-prize} $\equiv$ \textit{hide-behind} 1 \(\cap\) \textit{hide-behind} 2 \(\cap\) \textit{hide-behind} 3

Guess uniformly at random.

\textbf{definition}\ \textit{make-guess} :: \textit{game} \textit{prog}
\textbf{where}\ \textit{make-guess} $\equiv$ \textit{guess-behind} 1 \(\oplus\) \textit{guess-behind} 2 \(\oplus\) \textit{guess-behind} 3

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

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

Switch your guess to the other unopened door.

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

The complete game, either with or without switching guesses.

\textbf{definition}\ \textit{monty} :: \textit{bool} $\Rightarrow$ \textit{game} \textit{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.

**Lemma eval-win** [simp]:

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

by (simp add: embed-bool-def player-wins-def)

**Lemma eval-loss** [simp]:

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

by (simp add: embed-bool-def player-wins-def)

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

**Lemma wp-monty-noswitch**: 

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

**Unfolding**: 

\text{monty-def hide-prize-def make-guess-def reveal-def}

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

\text{switch-guess-def}

by (simp add: wp-eval insert-Diff-if o-def cong del: strong-INF-cong)

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 ∃s. {1, 2, 3} − {prize s, guess s} ≠ {}
  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 ∃s. {1, 2, 3} − {clue s, guess s} ≠ {}
  by (auto simp insert-Diff-if)
- thus ?thesis
  unfolding switch-guess-def guess-behind-def
  by (intro wd-intros, auto)
  qed

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

**Annotations**

We now annotate each step individually, and then combine them to produce an annotation for the entire program.
hide-prize chooses a valid door.

**Lemma wp-hide-prize:**

\[(\lambda s. 1) \vdash wp\ hide-prize\ «\ inv-prize»\]

**Unfolding** hide-prize-def hide-behind-def wp-eval o-def

By (simp add: embed-bool-def inv-prize-def)

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

**Lemma wp-make-guess:**

\[(\lambda s. 2/3 \star \langle \lambda g. \ inv-prize\ g\rangle\ s) \vdash wp\ make-guess\ «\ \lambda g.\ guess\ g\ \neq\ \prize\ g\ \land\ \inv-prize\ g\ \land\ \inv-guess\ g\rangle\]

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

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

**Lemma last-one:**

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

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

Apply (simp add: insert-Diff-if)

Using assms by (auto intro: assms)

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

**Lemma wp-reveal:**

\[«\ \lambda g.\ guess\ g\ \neq\ \prize\ g\ \land\ \inv-prize\ g\ \land\ \inv-guess\ g\rangle\ \vdash wp\ reveal\ «\ \lambda g.\ guess\ g\ \neq\ \prize\ g\ \land\ \clue\ g\ \neq\ \prize\ g\ \land\ \clue\ g\ \neq\ \guess\ g\ \land\ \inv-prize\ g\ \land\ \inv-guess\ g\ \land\ \inv-clue\ g\rangle\]

(Is \(?X\ \vdash wp\ reveal\ ?Y\)?)

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

Fix \(s\)

Assume \(guess\ s\ \neq\ \prize\ s\)

And \(\inv-prize\ s\)

And \(\inv-guess\ s\)

Moreover then obtain \(c\)

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

And \(c\ \neq\ \prize\ s\)

And \(c\ \neq\ \guess\ s\)

And \(c\ \in\ \{Suc\ 0, 2, 3\}\)

**Unfolding** inv-prize-def inv-guess-def

By (force dest:last-one clarsimp!:ex1E)

Ultimately show \(1 \leq wp\ reveal\ ?Y\ s\)

By (simp add: reveal-def open-door-def wp-eval singleton o-def

embed-bool-def inv-prize-def inv-guess-def inv-clue-def)

QED

Showing that the three doors are all district is a largeish first-order problem, for which sledgehammer gives us a reasonable script.
lemma distinct-game:
[ guess g ≠ 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:
«λ g. guess g ≠ prize g ∧ clue g ≠ prize g ∧ clue g ≠ guess g ∧
inv-prize g ∧ inv-guess g ∧ inv-clue g » ⊢ «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])))
(by(auto simp:insert-Diff-if player-wins-def))
also from state have ... = Inf ((λa. « player-wins » (s[guess := a])))
(by(simp add:distinct-game[symmetric]))
also have ... = wp switch-guess «player-wins» s
(by(simp add:switch-guess-def guess-behind-def wp-eval o-def))
finally show 1 ≤ 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:
(λs. 2/3) ⊢ wp (monty True) «player-wins»
  unfolding monty-def
  by(simp, pvcg)
end
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](image)
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 \( \Rightarrow (\alpha \Rightarrow \text{real}) \Rightarrow \text{bool} \)

**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-by1[intro]:

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

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

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

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

by \((\text{blast dest:le-funD})\)

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

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

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

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

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

by \((\text{blast intro:le-funI})\)

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

**definition** bounded :: (\(\alpha \Rightarrow \text{real}\)) \( \Rightarrow \text{bool} \)

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

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

**definition** bound-of :: (\(\alpha \Rightarrow \text{real}\)) \( \Rightarrow \text{real} \)

**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 cInf-greatest)
qed

The least upper bound has the usual properties:

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

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

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

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

This is occasionally easier to apply:

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

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

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

**lemma** bounded-by-mono-alt[intro]:
  \[ \text{bounded-by } b \ Q ; \ P \leq Q \] \( \Rightarrow \text{bounded-by } b \ P \)
  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 □x. P x ≤ (c::real)
sows bound-of P ≤ c
unfolding bound-of-def
using assms by(intro cSup-least, auto)

lemma bound-of-mono[intro]:
□ P ≤ Q; bounded P; bounded Q ▸ bound-of P ≤ bound-of Q
by (blast intro:order-trans dest:le-funD)

lemma bounded-by-o[intro,simp]:
□ b. bounded-by b P ⇒ bounded-by b (P o f)
unfolding o-def by(blast)

lemma le-bound-of[intro]:
□ x. bounded f ⇒ f x ≤ 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 ⇒ 'b::{zero,order}) ⇒ bool
where
nneg P ⟷ (∀ x. 0 ≤ P x)

lemma nnegI[intro]:
□ □ x. 0 ≤ P x ⟹ nneg P
by (simp add:nneg-def)

lemma nnegI2[intro]:
(λs. 0) ≤ P ⇒ nneg P
by (blast dest:le-funD)

lemma nnegD[dest]:
nneg P ⇒ 0 ≤ P x
by (simp add:nneg-def)

lemma nnegD2[dest]:
nneg P ⇒ (λs. 0) ≤ P
by (blast intro:le-funI)

lemma nneg-bdd-below[intro]:
nneg P ⇒ bdd-below (range P)
by(auto)
3.1. EXPECTATIONS

**lemma** nneg-const[iff]:
\[ \text{nneg} (\lambda x. c) \leftrightarrow 0 \leq c \]
**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]:
\[ \text{fixes} P::'s \Rightarrow \text{real} \]
\[ \text{shows} [ \text{bounded-by} b P; \text{nneg} P ] \implies 0 \leq b \]
**by** (blast intro:order-trans)

### 3.1.3 Sound Expectations

**definition** sound :: (real) \Rightarrow bool

\[ \text{where} \quad \text{sound} P \equiv \text{bounded} P \land \text{nneg} P \]

Combining \textit{nneg} and \textit{Expectations.bounded}, we have **sound** expectations. We set up the classical reasoner and the simplifier, such that showing soundess, or deriving a simple consequence (e.g. \textit{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[intro]:
\[ \text{assumes} \quad sP: \text{sound} P \]
shows \( 0 \leq \text{bound-of } P \)

using \textit{assms by(\textit{auto})}

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

\textbf{lemma sound-sum\textdaggerbraceleft simp,intro\textdaggerbraceleft:}

\textit{assumes sP: sound P and sQ: sound Q}

\textit{shows sound (\(\lambda s. P s + Q s\))}

\textbf{proof}

\textit{from sP have }\land s. P s \leq \text{bound-of } P \text{ by(\textit{blast})}

\textit{moreover from sQ have }\land s. Q s \leq \text{bound-of } Q \text{ by(\textit{blast})}

\textit{ultimately have }\land s. P s + Q s \leq \text{bound-of } P + \text{bound-of } Q

\textit{by(rule add-mono)}

\textit{thus bounded-by }\left(\text{bound-of } P + \text{bound-of } Q\right) \text{ by(\textit{blast})}

\textit{from sP have }\land s. \ 0 \leq P s \text{ by(\textit{blast})}

\textit{moreover from sQ have }\land s. \ 0 \leq Q s \text{ by(\textit{blast})}

\textit{ultimately have }\land s. \ 0 \leq P s + Q s \text{ by(sim:add-mono)}

\textit{thus }\text{nneg }\left(\lambda s. P s + Q s\right) \text{ by(\textit{blast})}

\textit{qed}

\textbf{lemma mult-sound:}

\textit{assumes sP: sound P and sQ: sound Q}

\textit{shows sound }\left(\lambda s. P s \ast Q s\right)

\textbf{proof}

\textit{from sP have }\land s. P s \leq \text{bound-of } P \text{ by(\textit{blast})}

\textit{moreover from sQ have }\land s. Q s \leq \text{bound-of } Q \text{ by(\textit{blast})}

\textit{ultimately have }\land s. P s \ast Q s \leq \text{bound-of } P \ast \text{bound-of } Q

\textit{using sP and sQ by(blasm intro:mult-mono)}

\textit{thus bounded-by }\left(\text{bound-of } P \ast \text{bound-of } Q\right) \text{ by(\textit{blast})}

\textit{from sP and sQ show }\text{nneg }\left(\lambda s. P s \ast Q s\right)

\textit{by(blasm intro:mult-nonneg-nonneg)}

\textit{qed}

\textbf{lemma div-sound:}

\textit{assumes sP: sound P and cpos: \(0 < c\)}

\textit{shows sound }\left(\lambda s. P s \div c\right)

\textbf{proof}

\textit{from sP and cpos have }\land s. P s / c \leq \text{bound-of } P / c

\textit{by(blasm intro:divide-right-mono less-imp-le)}

\textit{thus bounded-by }\left(\text{bound-of } P / c\right) \text{ by(\textit{blast})}

\textit{from assms show }\text{nneg }\left(\lambda s. P s / c\right) \text{ by(\textit{blast})}

\textit{by(blasm intro:divide-nonneg-pos)}

\textit{qed}

\textbf{lemma tminus-sound:}

\textit{assumes sP: sound P and nnc: \(0 \leq c\)}
3.1. EXPECTATIONS

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

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

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

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

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

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

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

lemma sum-sound:
  assumes sP: \forall x \in S. \text{sound } (P x)
  shows \text{bounded-by } (\lambda s. \sum x \in S. P x s)
  proof (rule soundI2)
    from sP show \text{bounded-by } (\lambda s. \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.
lemma embed-bool-idem:

\[ \langle P \rangle s \ast \langle P \rangle s = \langle P \rangle s \]

by (simp add:embed-bool-def)

lemma eval-embed-true[simp]:

\[ P s \Rightarrow \langle P \rangle s = 1 \]

by (simp add:embed-bool-def)

lemma eval-embed-false[simp]:

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

by (simp add:embed-bool-def)

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

\[ 0 \leq \langle G \rangle s \]

by (simp add:embed-bool-def)

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

\[ \langle G \rangle s \leq 1 \]

by (simp add:embed-bool-def)

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

\[ 0 \leq 1 - \langle G \rangle s \]

by (subst add-le-cancel-right[where c=\langle G \rangle s, symmetric], simp)

lemma expect-1-I:

\[ P x \Rightarrow 1 \leq \langle P \rangle x \]

by (simp)

lemma standard-sound[intro,simp]:

sound \langle P \rangle

by (blast)

lemma embed-o[simp]:

\[ \langle P \rangle o f = \langle P o f \rangle \]

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

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

definition negate :: ('s ⇒ bool) ⇒ 's ⇒ bool (N)

where negate P = (λs. ¬ P s)

lemma negateI:

\[ \neg P s \Rightarrow N \langle P \rangle s \]

by (simp add:negate-def)

lemma embed-split:

\[ f s = \langle P \rangle s \ast f s + \langle N \rangle P s \ast f s \]

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

**Lemma negate-embed:**

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

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

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

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

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

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

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

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

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

\[ \langle \neg \neg \rangle = (\lambda x. x) \]

*by* (simp add:negate-def)

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

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

*by* (simp add:negate-def)

**Lemma embed-bool-cancel:**

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

*by* (cases G s, simp-all)

### 3.1.6 Entailment

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

**Abbreviation entails:: ('s ⇒ real) ⇒ ('s ⇒ real) ⇒ bool (- ⊢ - 50)**

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

**Lemma entailsI[intro]:**

\[ \forall s. P s \leq Q s \Rightarrow P \vdash Q \]

*by* (simp add:le-funI)

**Lemma entailsD[dest]:**

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

*by* (simp add:le-funD)

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

\[ P = Q \Rightarrow P \vdash Q \]

*by* (blast)

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

\[ \langle P \vdash Q; Q \vdash R \rangle \Rightarrow P \vdash R \]

*by* (blast intro:order-trans)

For standard expectations, both notions of entailment coincide. This result justifies the above claim that our definition generalises predicate entailment:
lemma \textit{implies-entails}:
\[
\forall s. \ P s \Rightarrow Q s \implies «P» \vdash «Q» \\
\text{by (rule entailsI, case-tac P s, simp-all)}
\]

lemma \textit{entails-implies}:
\[
\forall s. \ «P» \vdash «Q»; P s \implies Q s \\
\text{by (rule ccontr, drule-tac s=s in entailsD, simp)}
\]

### 3.1.7 Expectation Conjunction

definition \textit{pconj} :: real \Rightarrow real \Rightarrow real \ (\text{infixl} \ .& \ 71)
where
\[ p \ .& \ q \equiv p + q \ominus 1 \]

definition \textit{exp-conj} :: (′s \Rightarrow real) \Rightarrow (′s \Rightarrow real) \Rightarrow (′s \Rightarrow real) \ (\text{infixl} \ .&\& \ 71)
where
\[ a .&\& b \equiv \lambda s. (a s .\& \ b s) \]

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

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

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

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

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

lemma \textit{pconj-bconj}:
\[
«a» s .\& \ «b» s = «\lambda s. a s \land b s» s \\
\text{unfolding embed-bool-def pconj-def tminus-def by (force)}
\]

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

lemma \textit{pconj-assoc}:
\[
\[ 0 \leq a; a \leq 1; \theta \leq b; b \leq 1; \theta \leq c; c \leq 1 \] \implies
\]

3.1. EXPECTATIONS

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

\textbf{unfolding pconj-def tminus-def by(simp)}

\textbf{lemma pconj-mono:}
\[
[ a \leq b; c \leq d ] \implies a \land c \leq b \land d
\]

\textbf{unfolding pconj-def tminus-def by(simp)}

\textbf{lemma pconj-nneg[intro,simp]:}
\[
0 \leq a \land b
\]

\textbf{unfolding pconj-def tminus-def by(auto)}

\textbf{lemma min-pconj:}
\[
\text{min}(a \land b \land c \land d) \leq \text{min}(a \land c) \land (b \land d)
\]

\textbf{by(cases a \leq b,}
\begin{itemize}
\item simp-all add: \text{min.absorb1 min.absorb2 pconj-mono}]
\end{itemize}

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

\textbf{unfolding pconj-def by(simp)}

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

\textbf{unfolding pconj-def by(simp)}

\textbf{lemma pconj-idem[simp]:}
\[
\langle P \rangle s \land \langle P \rangle s = \langle P \rangle s
\]

\textbf{unfolding pconj-def by(cases P s, simp-all)}

3.1.8 Rules Involving Conjunction.

\textbf{lemma exp-conj-mono-left:}
\[
P \vdash Q \implies P \land R \vdash Q \land R
\]

\textbf{unfolding exp-conj-def pconj-def}

\textbf{by(auto intro:tminus-left-mono add-right-mono)}

\textbf{lemma exp-conj-mono-right:}
\[
Q \vdash R \implies P \land Q \vdash P \land R
\]

\textbf{unfolding exp-conj-def pconj-def}

\textbf{by(auto intro:tminus-left-mono add-left-mono)}

\textbf{lemma exp-conj-comm[ac-simps]:}
\[
a \land b = b \land a
\]

\textbf{by(simp add:exp-conj-def ac-simps)}

\textbf{lemma exp-conj-bounded-by[intro,simp]:}
\[
\text{assumes} \; bP: \text{bounded-by 1 P}
\]
CHAPTER 3. SEMANTIC STRUCTURES

and \( bQ \) bounded-by 1 \( Q \)
shows bounded-by 1 \((P \& \& Q)\)

proof (rule bounded-byI, unfold exp-conj-def pconj-def)

fix \( x \)
from \( bP \) have \( P \leq 1 \) by (blast)
moreover from \( bQ \) have \( Q \leq 1 \) by (blast)
ultimately have \( P + Q \leq 2 \) by (auto)

unfolding tminus-def by (simp)

qed

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

unfolding exp-conj-def o-def by (simp)

lemma exp-conj-assoc:
assumes unitary P and unitary Q and unitary R
shows \( P \& \& (Q \& \& R) = (P \& \& Q) \& \& R \)

unfolding exp-conj-def

proof (rule ext)

fix \( s \) from assms have \( 0 \leq P s \) 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 \( P s \& (Q s \& R s) = (P s \& Q s) \& R s \)

by (simp add: pconj-assoc)

qed

lemma exp-conj-top-left[simp]:
sound \( P \Rightarrow \lambda s. \text{True} \) \& \& \( P = P \)

unfolding exp-conj-def by (force)

lemma exp-conj-top-right[simp]:
sound \( P \Rightarrow P \& \& \lambda s. \text{True} = P \)

unfolding exp-conj-def by (force)

lemma exp-conj-idem[simp]:
\( \lambda s. \lambda s \) \& \& \( \lambda s \) = \( \lambda s \)

unfolding exp-conj-def

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

lemma exp-conj-nneg[intro,simp]:
\( \lambda s. 0 \) \leq \( P \& \& Q \)

unfolding exp-conj-def

by (blast intro:le-funI)
3.1. EXPECTATIONS

**lemma** exp-conj-sound[~intro,simp]:

**assumes** s-P; sound P

and s-Q; sound Q

**shows** sound (P && Q)

**unfolding** exp-conj-def

**proof**(rule soundI)

from s-P and s-Q have \( \forall s. 0 \leq P s + Q s \) by (blast intro: add-nonneg-nonneg)

**hence** \( \forall s. P s \& Q s \leq P s + Q s \)

**unfolding** pconj-def by (force intro:tminus-less)

also from assms have \( \forall s. \ldots s \leq \text{bound-of } P + \text{bound-of } Q \)

by (blast intro: add-mono)

**finally have** bounded-by (bound-of P + 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** pconj-def tminus-def by (force)

**qed**

**lemma** exp-conj-rzero[simp]:

\( \text{bounded-by } 1 \Rightarrow P \equiv (\lambda s. 0) \equiv (\lambda s. 0) \)

**unfolding** exp-conj-def by (force)

**lemma** exp-conj-1-right[simp]:

**assumes** nn: nneg A

**shows** A \& (\lambda. 1) = A

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

**proof**(rule ext, simp)

fix s

from nn have \( 0 \leq A s \) by (blast)

**thus** max (A s) \( 0 = A s \) by (force)

**qed**

**lemma** exp-conj-std-split:

\( \langle \lambda s. P s \& Q s \rangle = \langle P \rangle \& \langle Q \rangle \)

**unfolding** exp-conj-def embed-bool-def pconj-def

by (auto)

3.1.9 Rules Involving Entailment and Conjunction Together

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

**lemma** entails-frame:

**assumes** ePR: P \( \vdash \) R

and eQS: Q \( \vdash \) S

**shows** P \& Q \( \vdash \) R \& S

**proof**(rule le-funI)

fix s
from $ePR$ have $P \ s \ \leq \ R \ s \ \text{by(\text{blast})}$
moreover from $eQS$ have $Q \ s \ \leq \ S \ s \ \text{by(\text{blast})}$
ultimately have $P \ s + Q \ s \ \leq \ R \ s + S \ s \ \text{by(\text{rule add-mono})}$
hence $(P \ \&\& \ Q) \ s \ \leq \ (R \ \&\& \ S) \ s$
thus 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 $PQe$: $\forall x. \ P(x) \implies Q(x)$
and exhaust: $\forall s. \ \exists x. \ P(x \ s) = 1$
and framed: $\forall x. \ P(x) \ \&\& \ R \implies Q(x) \ \&\& \ S$
and $sR$: sound $R$
and $sS$: sound $S$
and $bQ$: $\forall x. \ \text{bounded-by} \ 1 \ (Q(x))$
shows $R \implies S$

proof (\text{rule le-funI})
fix $s$
from exhaust obtain $x$ where $P\text{-xs}: P(x \ s) = 1$ by (\text{blast})
moreover {
  hence $1 = P(x) \ s$ by (\text{simp})
  also from $PQe$ have $P(x) \ s \ \leq \ Q(x) \ s$ \text{by(\text{blast dest:le-funD})}
  finally have $Q(x) \ s = 1$
  using $bQ$ by (\text{blast intro:antisym})
}
moreover note $\text{le-funD}[\text{OF \ framed[where } x=x\text{], where } x=s]$
moreover from $sR$ have $0 \ \leq \ R \ s$ \text{by(\text{blast})}
moreover from $sS$ have $0 \ \leq \ S \ s$ \text{by(\text{blast})}
ultimately show $R \ s \ \leq \ S \ s$ \text{by(\text{simp add:exp-conj-def})}

qed

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

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

lemma unitary-embed[iff]:
unitary $(\text{«P»})$
by (\text{auto})

lemma unitary-const[iff]:
$[ \ 0 \ \leq \ c; \ c \ \leq \ 1 \ ] \implies \ \text{unitary} \ (\lambda s. \ c)$
by (\text{auto})

lemma unitary-mult:
assumes \( uA \): unitary \( A \) and \( uB \): unitary \( B \)
show\( s \) \( (\lambda s . A \cdot B s) \)
proof\( (\text{intro unitaryI2 nnegI bounded-byI}) \)
fix \( s \)
\( \text{from \ assms \ have \ } \text{nnA: } 0 \leq A s \) \( \text{and \ } \text{nnB: } 0 \leq B s \) \( \text{by(auto)} \)
\( \text{thus } 0 \leq A s \cdot B s \) \( \text{by(\text{rule \ mult-nonneg-nonneg})} \)
\( \text{from \ assms \ have \ } A s \leq 1 \) \( \text{and \ } B s \leq 1 \) \( \text{by(auto)} \)
\( \text{with \ nnB \ have \ } A s \cdot B s \leq 1 \cdot 1 \) \( \text{by(\text{intro \ mult-mono, \ auto})} \)
\( \text{also \ have \ } ... = 1 \) \( \text{by(simp)} \)
\( \text{finally \ show \ } A s \cdot B s \leq 1 \).
qed

lemma \( \text{exp-conj-unitary:} \)
\( [ \text{unitary } P; \text{unitary } Q ] \implies \text{unitary } (P \& \& Q) \)
\( \text{by(\text{intro \ unitaryI2 \ nnegI, \ auto})} \)

lemma \( \text{unitary-comp[simp]:} \)
\( \text{unitary } P \implies \text{unitary } (P \circ f) \)
\( \text{by(\text{intro \ unitaryI2 \ nnegI \ bounded-byI, \ auto \ simp:o-def})} \)

lemmas \( \text{unitary-intros =} \)
\( \text{unitary-bot \ unitary-top \ unitary-embed \ unitary-mult \ exp-conj-unitary} \)
\( \text{unitary-comp \ unitary-const} \)

lemmas \( \text{sound-intros =} \)
\( \text{mult-sound \ div-sound \ const-sound \ sound-o \ sound-sum} \)
\( \text{tminus-sound \ sc-sound \ exp-conj-sound \ sum-sound} \)

end

3.2 Expectation Transformers

theory Transformers imports Expectations begin

\( \text{type-synonym 's trans = 's expect } \rightarrow \text{'s expect} \)

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

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: $P_{\text{prior}}(a) = 0.7 \cdot P_{\text{post}}(b) + 0.3 \cdot P_{\text{post}}(c)$, 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 hence 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: $P_{\text{prior}}(e) = 0.5 \cdot P_{\text{post}}(b) + 0.3 \cdot P_{\text{post}}(c)$.

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 $a \rightarrow a \rightarrow \ldots$ thus has probability 0, and the automaton terminates with probability 1. We formalise such probabilistic termination...
3.2. EXPECTATION TRANSFORMERS

Figure 3.3: A diverging automaton.

arguments in Section 4.11.

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

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

Healthiness, again, is the combination of three properties: feasibility, monotonicity and scaling. Feasibility requires that a transformer take non-negative expectations to non-negative expectations, and preserve bounds. Thus, starting with an expectation bounded between 0 and some bound, $b$, after applying any number of feasible transformers, the result will still be bounded between 0 and $b$. This closure property allows us to treat expectations almost as a complete lattice. Specifically, for any $b$, the set of expectations bounded by $b$ is a complete lattice ($\bot = (\lambda s.0)$, $\top = (\lambda s.b)$), and is closed under the action of feasible transformers, including $\cap$ 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**

\[
\text{le-trans} :: \ 's \text{ trans} \Rightarrow 's \text{ trans} \Rightarrow \text{bool}
\]

**where**

\[
\text{le-trans } t \ u \equiv \forall P. \text{ sound } P \rightarrow t \ P \leq u \ P
\]

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

**Definition**

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

**where**

\[
\text{le-utrans } t \ u \longleftrightarrow (\forall P. \text{ unitary } P \rightarrow t \ P \leq u \ P)
\]

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

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

by (simp add: le-trans-def)

**Lemma** *le-utransI [intro]*:

\[
[ \forall P. \text{ unitary } P \Rightarrow t \ P \leq u \ P ] \Rightarrow \text{le-utrans } t \ u
\]

by (simp add: le-utrans-def)

**Lemma** *le-transD [dest]*:

\[
[ \text{le-trans } t \ u \text{; sound } P ] \Rightarrow t \ P \leq u \ P
\]

by (simp add: le-trans-def)

**Lemma** *le-utransD [dest]*:

\[
[ \text{le-utrans } t \ u \text{; unitary } P ] \Rightarrow t \ P \leq u \ P
\]

by (simp add: le-utrans-def)

**Lemma** *le-trans-trans [trans]*:

\[
[ \text{le-trans } x \ y \text{; le-trans } y \ z ] \Rightarrow \text{le-trans } x \ z
\]

unfolding le-trans-def by (blast dest: order-trans)

**Lemma** *le-utrans-trans [trans]*:

\[
[ \text{le-utrans } x \ y \text{; le-utrans } y \ z ] \Rightarrow \text{le-utrans } x \ z
\]

unfolding le-utrans-def by (blast dest: order-trans)

**Lemma** *le-trans-refl [iff]*:

\[
\text{le-trans } x \ x
\]

by (simp add: le-trans-def)

**Lemma** *le-utrans-refl [iff]*:

\[
\text{le-utrans } x \ x
\]

by (simp add: le-utrans-def)
3.2. **EXPECTATION TRANSFORMERS**

**Lemma** \( \text{le-trans-le-utrans}[\text{dest}]: \)
\[ \text{le-trans} \ t \ u \implies \text{le-utrans} \ t \ u \]

**Unfolding** \( \text{le-trans-def le-utrans-def} \) **by**(auto)

**Definition**
\( \text{l-trans} :: 's \ \text{trans} \Rightarrow 's \ \text{trans} \Rightarrow \text{bool} \)
**Where**
\( \text{l-trans} \ t \ u \leftarrow\rightarrow \text{le-trans} \ t \ u \land \neg \text{le-trans} \ u \ t \)

Transformer equivalence is induced by comparison:

**Definition**
\( \text{equiv-trans} :: 's \ \text{trans} \Rightarrow 's \ \text{trans} \Rightarrow \text{bool} \)
**Where**
\( \text{equiv-trans} \ t \ u \leftarrow\rightarrow \text{le-trans} \ t \ u \land \text{le-trans} \ u \ t \)

**Definition**
\( \text{equiv-utrans} :: 's \ \text{trans} \Rightarrow 's \ \text{trans} \Rightarrow \text{bool} \)
**Where**
\( \text{equiv-utrans} \ t \ u \leftarrow\rightarrow \text{le-utrans} \ t \ u \land \text{le-utrans} \ u \ t \)

**Lemma** \( \text{equiv-transI}[\text{intro}]: \)
\[
[ \forall P. \text{sound} \ P \implies t \ P = u \ P ] \implies \text{equiv-trans} \ t \ u
\]
**Unfolding** \( \text{equiv-trans-def} \) **by**(force)

**Lemma** \( \text{equiv-utransI}[\text{intro}]: \)
\[
[ \forall P. \text{sound} \ P \implies t \ P = u \ P ] \implies \text{equiv-utrans} \ t \ u
\]
**Unfolding** \( \text{equiv-utrans-def} \) **by**(force)

**Lemma** \( \text{equiv-transD}[\text{dest}]: \)
\[
[ \text{equiv-trans} \ t \ u; \text{sound} \ P ] \implies t \ P = u \ P
\]
**Unfolding** \( \text{equiv-trans-def} \) **by**(blast intro:antisym)

**Lemma** \( \text{equiv-utransD}[\text{dest}]: \)
\[
[ \text{equiv-utrans} \ t \ u; \text{unitary} \ P ] \implies t \ P = u \ P
\]
**Unfolding** \( \text{equiv-utrans-def} \) **by**(blast intro:antisym)

**Lemma** \( \text{equiv-trans-refl}[\text{iff}]: \)
\( \text{equiv-trans} \ t \ t \)
**by**(blast)

**Lemma** \( \text{equiv-utrans-refl}[\text{iff}]: \)
\( \text{equiv-utrans} \ t \ t \)
**by**(blast)

**Lemma** \( \text{le-trans-antisym}: \)
\[
[ \text{le-trans} \ x \ y; \text{le-trans} \ y \ x ] \implies \text{equiv-trans} \ x \ y
\]
**Unfolding** \( \text{equiv-trans-def} \) **by**(simp)

**Lemma** \( \text{le-utrans-antisym}: \)
\[ \text{equiv-trans-comm [ac-simps]}: \]
\[ \text{equiv-trans t u } \leftrightarrow \text{equiv-trans u t} \]
unfolding equiv-trans-def by (blast)

\[ \text{equiv-utrans-comm [ac-simps]}: \]
\[ \text{equiv-utrans t u } \leftrightarrow \text{equiv-utrans u t} \]
unfolding equiv-utrans-def by (blast)

\[ \text{equiv-imp-le [intro]}: \]
\[ \text{equiv-trans t u } \Rightarrow \text{le-trans t u} \]
unfolding equiv-trans-def by (clarify)

\[ \text{equiv-imp-le-alt}: \]
\[ \text{equiv-trans t u } \Rightarrow \text{le-trans u t} \]
by (force simp: ac-simps)

\[ \text{equiv-imp-le-alt}: \]
\[ \text{equiv-utrans t u } \Rightarrow \text{le-utrans u t} \]
by (force simp: ac-simps)

\[ \text{le-trans-equiv-rsp [simp]}: \]
\[ \text{equiv-trans t u } \Rightarrow \text{le-trans t v } \leftrightarrow \text{le-trans u v} \]
unfolding equiv-trans-def by (blast intro: le-trans-trans)

\[ \text{le-utrans-equiv-rsp [simp]}: \]
\[ \text{equiv-utrans t u } \Rightarrow \text{le-utrans t v } \leftrightarrow \text{le-utrans u v} \]
unfolding equiv-utrans-def by (blast intro: le-utrans-trans)

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

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

\[ \text{equiv-trans-le-trans [trans]}: \]
\[ \text{equiv-trans t u; le-trans v t } \leftrightarrow \text{le-trans v u} \]
unfolding equiv-trans-def by (blast intro: le-trans-trans)

\[ \text{equiv-utrans-le-utrans [trans]}: \]
\[ \text{equiv-utrans t u; le-utrans v t } \leftrightarrow \text{le-utrans v u} \]
unfolding equiv-utrans-def by (blast intro: le-trans-trans)
unfolding equiv-utrans-def by(blast intro:le-utrans-trans)

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

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

lemma equiv-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 \implies 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]:**
\[
\begin{align*}
\& b P. \, [\text{bounded-by } b P; \text{nngen } P] \implies \text{bounded-by } b (t P); \\
\& b P. \, [\text{bounded-by } b P; \text{nngen } P] \implies \text{nngen } (t P)] \implies \text{feasible } t
\end{align*}
\]
by (force simp: feasible-def)

**Lemma feasible-boundedD[dest]:**
\[
\begin{align*}
[\text{feasible } t; \text{bounded-by } b P; \text{nngen } P] & \implies \text{bounded-by } b (t P) \\
& \text{by } (simp add: feasible-def)
\end{align*}
\]

**Lemma feasible-nnegD[dest]:**
\[
\begin{align*}
[\text{feasible } t; \text{bounded-by } b P; \text{nngen } P] & \implies \text{nngen } (t P) \\
& \text{by } (simp add: feasible-def)
\end{align*}
\]

**Lemma feasible-sound[dest]:**
\[
\begin{align*}
[\text{feasible } t; \text{sound } P] & \implies \text{sound } (t P) \\
& \text{by } (rule \text{ soundI, unfold sound-def, (blast)+})
\end{align*}
\]

**Lemma feasible-pr-0[simp]:**
\[
\begin{align*}
\text{fixes } t :: (\lambda s :: \text{real} \Rightarrow \text{real}) & \Rightarrow (\lambda s :: \text{real}) \\
\text{assumes } ft & \text{ feasible } t \\
\text{shows } t (\lambda x. 0) & = (\lambda x. 0) \\
\text{proof } (rule \text{ ext, rule antisym}) \\
& \text{fix } s \\
& \text{have bounded-by } 0 (\lambda :: s. 0 :: \text{real}) \text{ by (blast)} \\
& \text{with } ft \text{ have bounded-by } 0 (t (\lambda s. 0)) \text{ by (blast)} \\
& \text{thus } t (\lambda s. 0) s \leq 0 \text{ by (blast)}
\end{align*}
\]

**Lemma feasible-fixes-top:**
\[
\begin{align*}
& \text{have nngen } (\lambda :: s. 0 :: \text{real}) \text{ by (blast)} \\
& \text{with } ft \text{ have nngen } (t (\lambda s. 0)) \text{ by (blast)} \\
& \text{thus } 0 \leq t (\lambda s. 0) s \text{ by (blast)}
\end{align*}
\]
qed

**Lemma feasible-id:**
\[
\text{feasible } (\lambda x. x)
\]
**Unfolding** feasible-def by (blast)

**Lemma feasible-bounded-by[dest]:**
\[
\begin{align*}
[\text{feasible } t; \text{sound } P; \text{bounded-by } b P] & \implies \text{bounded-by } b (t P) \\
& \text{by } (auto)
\end{align*}
\]

**Lemma feasible-fixes-bot:**
\[
\begin{align*}
\text{feasible } t \implies t (\lambda x. t) \leq (\lambda s. (t :: \text{real})) \\
& \text{by } (drule \text{ bounded-byD2 [OF feasible-bounded-by], auto})
\end{align*}
\]

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

assumes ft: feasible t
shows t (\lambda s. 0) = (\lambda s. 0)
proof(auto)

have sb: sound (\lambda s. 0) by(auto)
with ft show (\lambda s. 0) \leq t (\lambda s. 0) by(auto)

from sb have bounded-by (bound-of (\lambda s. 0::real)) (\lambda s. 0) by(auto)

with ft have bounded-by 0 (t (\lambda s. 0)) by(auto)
thus t (\lambda s. 0) \leq (\lambda s. 0) by(auto)
qed

lemma feasible-unitaryD[dest]:
assumes ft: feasible t and uP: unitary P
shows unitary (t P)
proof(auto)
from uP have sound P by(auto)
with ft show sound (t P) by(auto)
from assms show bounded-by 1 (t P) by(auto)
qed

Monotonicity

definition

mono-trans :: (('s ⇒ real) ⇒ ('s ⇒ real)) ⇒ bool
where

mono-trans t ≡ ∀ P Q. (sound P ∧ sound Q ∧ P ≤ Q) ⇒ 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 ⊢ 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 ⊢ u (t R), or that the weakest precondition of Q under u entails that of R under the composition u ∘ t. If we additionally know that P ⊢ u Q, then by transitivity we have P ⊢ u (t R). We thus derive a probabilistic form of the standard rule for sequential composition: [mono-trans t; P ⊢ u Q; Q ⊢ t R] ⇒ P ⊢ u (t R).

lemma mono-transI[intro]:
[ [ \forall P Q. [ sound P; sound Q; P \leq Q ] ⇒ t P \leq t Q ] ⇒ mono-trans t
by(simp add:mono-trans-def)

lemma mono-transD[dest]:
[ mono-trans t; sound P; sound Q; P \leq Q ] ⇒ 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

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 \implies mono-trans t
  by(simp add:healthy-def)

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

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

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

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

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

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

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

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

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

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

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

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

lemma le-trans-bot[intro,simp]:
  healthy t \implies le-trans (\lambda P. s. 0) t
  by(blast intro:le-funI)

lemma le-trans-top[intro,simp]:
  healthy t \implies le-trans t (\lambda P. s. bound-of P)
  by(blast intro!:le-trans[OF le-funI])

lemma healthy-pr-bot[simp]:
  healthy t \implies t (\lambda s. 0) = (\lambda s. 0)
  by(blast intro:feasible-pr-0)

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

lemma healthy-equivI:
  fixes t::(s \Rightarrow real) \Rightarrow s \Rightarrow real and u
  assumes equiv: equiv-trans t u
  and healthy: healthy t
  shows healthy u
proof
  have le-t-u: le-trans t u by(blast intro:equiv)
  have le-u-t: le-trans u t by(simp add:equiv-imp-le ac-simps equiv)
  from equiv have eq-u-t: equiv-trans u t by(simp add:ac-simps)
  from sP and le-u-t have \forall s. s \leq u P s by(blast)
  also from sQ and le-t-u have \forall s. u P s \leq t P s by(blast)
  finally show nneg (u P) by(blast)

show feasible u
proof
  fix b and P::'s \Rightarrow real
  assume bP: bounded-by b P and np: nneg P
  hence sP: sound P by(blast)
  with healthy have \forall s. 0 \leq t P s by(blast)
  also from sP and le-t-u have \forall s. ... s \leq u P s by(blast)
  finally show nneg (u P) by(blast)

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

show mono-trans u
proof
  fix P::'s \Rightarrow real and Q::'s \Rightarrow real
  assume sP: sound P and sQ: sound Q
  and le: P \vdash Q
  from sP and le-u-t have u P \vdash t P by(blast)
  also from sP and sQ and le and healthy have t P \vdash t Q by(blast)
  finally show 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)
morerover from sound and pos have sc-nneg: nneg (λx. c * P x)
by(blast intro:mult-nonneg-nonneg less-imp-le)
ultimately have sc-sound: sound (λx. c * P x) by(blast)

show c * u P x = u (λx. c * P x) x
proof –
from sound have c * u P x = c * t P x
by(simp add: equiv-transD[OF eq-u-t])
also have ... = t (λx. c * P x) x
using healthy and sound and pos
by(blast intro: scalingD)
also from sc-sound and equiv have ...
= u (λx. c * P x) x
by(blast intro: fun-cong)

finally show ?thesis .
qed
qed

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

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

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

\textbf{definition}
\begin{itemize}
  \item \textbf{nearly-healthy :: \(('s \Rightarrow \text{real}) \Rightarrow ('s \Rightarrow \text{real}) \Rightarrow \text{bool}\)}
  \item \textbf{where}
  \end{itemize}
\begin{itemize}
\item nearly-healthy \( t \leftarrow (\forall P. \text{unitary } P \rightarrow \text{unitary } (t \ P)) \land \)
  \( (\forall P. Q. \text{unitary } P \rightarrow \text{unitary } Q \rightarrow P \vdash Q \rightarrow t \ P \vdash t \ Q) \)
\end{itemize}
\textbf{lemma nearly-healthy[intro]:}
\begin{itemize}
\item \textbf{[ \( \forall P. \text{unitary } P \Rightarrow \text{unitary } (t \ P) \);}
\item \( P, Q. [\text{unitary } P; \text{unitary } Q; P \vdash Q] \Rightarrow t \ P \vdash t \ Q \] \Rightarrow \text{nearly-healthy } t \)}
\end{itemize}
\textbf{by (simp add:nearly-healthy-def)}
\textbf{lemma nearly-healthy-monoD[dest]:}
3.2. EXPECTATION TRANSFORMERS

\[ [\text{nearly-healthy } t; \text{ sound } P; \text{ unitary } Q ] \Rightarrow t P + t Q \]
by(simp add:nearly-healthy-def)

**Lemma** nearly-healthy-unitaryD(dest):
\[ [\text{nearly-healthy } t; \text{ unitary } P ] \Rightarrow \text{unitary } (t P) \]
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 ::
\(((\,')\Rightarrow\text{real})\Rightarrow(\,')\Rightarrow\text{bool}\)
where
\(\text{sublinear } t \leftarrow\ (\forall a b c P Q s. (\text{sound } P \land \text{sound } Q \land 0 \leq a \land 0 \leq b \land 0 \leq c) \Rightarrow a \ast t P s + b \ast t Q s \ominus c \leq t (\lambda\,s'. a \ast P s' + b \ast Q s' \ominus c) s)\)

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

**Lemma** sublinearD[dest]:
\[ [\text{sublinear } t; \text{sound } P; \text{sound } Q; 0 \leq a; 0 \leq b; 0 \leq c ] \Rightarrow a \ast t P s + b \ast t Q s \ominus c \leq t (\lambda s'. a \ast P s' + b \ast Q s' \ominus c) s \]
by(simp add:sublinear-def)

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

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

The original curve, $P$, is trivially convex—it is linear. Also, both $t$ and $u$, and the operator $\sqcap$ preserve convexity. A probabilistic choice will also preserve it. The preservation of convexity is a property of sub-additive transformers.
that respect scaling. Note the form of the definition of convexity:
\[
\forall x, y. Q(x) + Q(y) \leq Q(\frac{x + y}{2})
\]
Were we to replace \( Q \) by some sub-additive transformer \( v \), and \( x \) and \( y \) by expectations \( R \) and \( S \), the equivalent expression:
\[
\frac{vR + vS}{2} \leq v(\frac{R + S}{2})
\]
Can be rewritten, using scaling, to:
\[
\frac{1}{2}(vR + vS) \leq \frac{1}{2}v(R + S)
\]
Which holds everywhere exactly when \( v \) is sub-additive i.e.:
\[
vR + vS \leq v(R + S)
\]

**lemma sub-addI[intro]:**
\[
[ \Lambda P Q s. [ sound P; sound Q ] \implies t P s + t Q s \leq t (\lambda s'. P s' + Q s') s ] \implies sub-add t
\]
by(simp add:sub-add-def)

**lemma sub-addI2:**
\[
[\Lambda P Q. [ sound P; sound Q ] \implies \lambda s. t P s + t Q s \vdash t (\lambda s. P s + Q s) s ] \implies sub-add t
\]
by(auto)

**lemma sub-addD[dest]:**
\[
[ sub-add t; sound P; sound Q ] \implies t P s + t Q s \leq t (\lambda s'. P s' + Q s') s
\]
by(simp add:sub-add-def)

**lemma equiv-sub-add:**
\[
fixes t::('s => real) => 's => real
assumes eq: equiv-trans t u
and sa: sub-add t
shows sub-add u
\]
**proof**
\[
fix P::'s => real and Q::'s => real and s::'s
assume sP: sound P and sQ: sound Q
\]
with eq have u P s + u Q s = t P s + t Q s
by(simp add:equiv-transD)
also from sP sQ sa have t P s + t Q s \leq t (\lambda s. P s + Q s) s
by(auto)
also 
from sP sQ have sound (\lambda s. P s + Q s) by(auto)
with \( \text{eq} \) have \( t (\lambda s. P \ s + Q \ s) \ s = u (\lambda s. P \ s + Q \ s) \ s \) 
by(simp add:equiv-transD)
\}
finally show \( u P \ s + u Q \ s \leq u (\lambda s. P \ s + Q \ s) \ s \).
qed

Sublinearity and feasibility imply sub-additivity.

**lemma** sublinear-subadd:

fixes \( t :: (\'s \Rightarrow \text{real}) \Rightarrow \'s \Rightarrow \text{real} \)
assumes slt: sublinear \( t \)
and ft: feasible \( t \)
shows sub-add \( t \)

proof
fix \( P :: \'s \Rightarrow \text{real} \) and \( Q :: \'s \Rightarrow \text{real} \) and \( s :: \'s \)
assume sP: sound \( P \) and sQ: sound \( Q \)
with ft have sound \( (t P) \) sound \( (t Q) \) by(auto)
hence \( 0 \leq t P \ s \) and \( 0 \leq t Q \ s \) by(auto)
hence \( 0 \leq t P \ s + t Q \ s \) by(auto)
hence \( ... = \ ... \odot 0 \) by(simp)

also from sP sQ
have \( ... \leq t (\lambda s. P \ s + Q \ s \odot 0) \ s \)
by(rule sublinearD[OF slt, where \( a=1 \) and \( b=1 \) and \( c=0 \), simplified])

also { 
from sP sQ have \( \bigwedge s. 0 \leq P \ s \) and \( \bigwedge s. 0 \leq Q \ s \) by(auto)
hence \( \bigwedge s. 0 \leq P s + Q s \) by(auto)
hence \( t (\lambda s. P s + Q s \odot 0) \ s = t (\lambda s. P s + Q s) \ s \)
by(simp)
\}
finally show \( t P \ s + t Q \ s \leq t (\lambda s. P s + Q s) \ s \).
qed

A few properties following from sub-additivity:

**lemma** standard-negate:

assumes ht: healthy \( t \)
and sat: sub-add \( t \)
shows \( t P s + t \neg N P s \leq 1 \)

proof –
from sat have \( t P s + t \neg N P s \leq t (\lambda s. P s + \neg N P s) \ s \) by(auto)
also have \( ... = t (\lambda s. 1) \ s \) by(simp add:negate-embed)
also { 
from ht have bounded-by 1 \( t (\lambda s. 1) \) by(auto)
also { 
from \( \text{ht} \) have bounded-by 1 \( t (\lambda s. 1) \) by(auto)

finally show \( ?\text{thesis} \).
qed
3.2. EXPECTATION TRANSFORMERS

lemma sub-add-sum:
fixes t::'s trans and S::'a set
assumes sat: sub-add t
and ht: healthy t
and sP: \( \forall x. \text{sound} \ (P \ x) \)
shows \( (\lambda x. \sum y \in S. \ t \ (P \ y) \ x) \leq t (\lambda x. \sum y \in S. \ P \ y \ x) \)
proof (cases infinite S, simp-all add:ht)
assume fS: finite S
show ?thesis
proof (rule finite-induct[OF fS le-funI le-funI], simp-all)
fix s::'s
from ht have sound (t (\lambda s. 0)) by (auto)
thus \( 0 \leq t (\lambda s. 0) \ s \) by (auto)

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

lemma sub-add-guard-split:
fixes t::'s finite trans and P::'s expect and s::'s
assumes sat: sub-add t
and ht: healthy t
and sP: sound P
shows \( (\sum y \in \{s, \ G \ s\}). \ P \ y \ast t \ « \ \lambda z. \ z = y \ast s) + (\sum y \in \{s, \neg G \ s\}). \ P \ y \ast t \ « \ \lambda z. \ z = y \ast s) \leq t \ P \ s \)
proof
have \( \{s, \ G \ s\} \cap \{s, \neg G \ s\} = \{} \) by (blast)
hence \( (\sum y \in \{s, \ G \ s\}). \ P \ y \ast t \ « \ \lambda z. \ z = y \ast s) + (\sum y \in \{s, \neg G \ s\}). \ P \ y \ast t \ « \ \lambda z. \ z = y \ast s) = \)
\( (\sum y \in \{s, \ G \ s\} \cup \{s, \neg G \ s\}). \ P \ y \ast t \ « \ \lambda z. \ z = y \ast s) \)
by (auto intro: sum.union_disjoint[symmetric])
also \{
have \( \{s, \ G \ s\} \cup \{s, \neg G \ s\} = UNIV \) by (blast)
hence \( (\sum y \in \{s, \ G \ s\} \cup \{s, \neg G \ s\}). \ P \ y \ast t \ « \ \lambda z. \ z = y \ast s) = \)
\( (\lambda x. \sum y \in UNIV. \ P \ y \ast t \ (\lambda x. \ « \lambda z. \ z = y \ast x)) \ x) \ s \)
by (simp)
\}
also \{

\end{verbatim}
CHAPTER 3. SEMANTIC STRUCTURES

from $sP$ have $\bigwedge y. \ 0 \leq P \ y$ by(auto)
with healthy-scalingD[OF ht]
have $(\lambda x. \sum \ y \in \text{UNIV}. \ P \ y \ast \ t \ (\lambda x. \ « \lambda z. \ z = y «) \ x) \ s =$
$(\lambda x. \sum \ y \in \text{UNIV}. \ t \ (\lambda x. \ P \ y \ast « \lambda z. \ z = y «) \ x) \ s$
by(simp add:scalingD)

also {
from sat ht $sP$

have $(\lambda x. \sum \ y \in \text{UNIV}. \ t \ (\lambda x. \ P \ y \ast « \lambda z. \ z = y «) \ x) \leq$
$t \ (\lambda x. \sum \ y \in \text{UNIV}. \ P \ y \ast « \lambda z. \ z = y «) \ x$
by(intro sub-add-sum sound-intros, auto)

hence $(\lambda x. \sum \ y \in \text{UNIV}. \ t \ (\lambda x. \ P \ y \ast « \lambda z. \ z = y «) \ x) \ s \leq$
$t \ (\lambda x. \sum \ y \in \text{UNIV}. \ P \ y \ast « \lambda z. \ z = y «) \ x$ by(auto)
}
also {

have ru1: $(\lambda x. \sum \ y \in \text{UNIV}. \ P \ y \ast « \lambda z. \ z = y «) \ x =$
$(\lambda x. \sum \ y \in \text{UNIV}. \ if \ y = x \ then \ P \ y \ else \ 0)$
by (rule ext [OF sum.cong]) auto

also from $sP$ have ... $\vdash P$
by(cases finite (UNIV::’s set), auto simp:sum.delta)

finally have leP: $\lambda x. \sum \ y \in \text{UNIV}. \ P \ y \ast « \lambda z. \ z = y « \ x \vdash P$.
moreover have sound $(\lambda x. \sum \ y \in \text{UNIV}. \ P \ y \ast « \lambda z. \ z = y «) \ x$
proof(intro soundI2 bounded-byI nnegI sum-nonneg ballI)

fix $x$
from leP have $(\sum \ y \in \text{UNIV}. \ P \ y \ast « \lambda z. \ z = y «) \ x \leq P \ x$ by(auto)
also from $sP$ have ... $\leq$ bound-of $P$ by(auto)

finally show $(\sum \ y \in \text{UNIV}. \ P \ y \ast « \lambda z. \ z = y «) \ x \leq$ bound-of $P$.

fix $y$
from $sP$ show $0 \leq P \ y \ast « \lambda z. \ z = y « \ x$
by(auto intro:mult-nonneg-nonneg)

qed

ultimately have $t \ (\lambda x. \sum \ y \in \text{UNIV}. \ P \ y \ast « \lambda z. \ z = y «) \ x \ s \leq t \ P \ s$
using $sP$ by(auto intro:le-funD[OF mono-transD, OF healthy-monoD, OF ht])
}

finally show ?thesis .

qed

Sub-distributivity

definition sub-distrib ::
((’s ⇒ real) ⇒ (’s ⇒ real)) ⇒ bool
where
sub-distrib $t \leftarrow \ (\forall P \ s. \ sound \ P \ −− t \ P \ s \ ⋃ \ 1 \ \leq \ t \ (\lambda s'. \ P \ s' \ ⋃ \ 1) \ s)$

lemma sub-distribI[intro]:
[ $\forall P \ s. \ sound \ P \ −\ t \ P \ s \ ⋃ \ 1 \ \leq \ t \ (\lambda s'. \ P \ s' \ ⋃ \ 1) \ s$ ] ⇒ sub-distrib $t$
by(simp add:sub-distrib-def)
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]:
\[ [ \text{sub-distrib } t; \text{sound } P ] \Rightarrow t \ P \ s \ominus 1 \leq t (\lambda s'. \ P \ s' \ominus 1) \ s \]
by(simp add:sub-distrib-def)

lemma equiv-sub-distrib:
fixes t::(\'s \Rightarrow \text{real}) \Rightarrow \'s \Rightarrow \text{real}
assumes eq: equiv-trans t u
and sd: sub-distrib t
shows sub-distrib u
proof
fix P::\'s \Rightarrow real and s::\'s
assume sP: sound P
moreover have 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 eq, where a=1 and b=0 and c=1, simplified])
qed

Sublinearity implies sub-distributivity:

lemma sublinear-sub-distrib:
fixes t::(\'s \Rightarrow \text{real}) \Rightarrow \'s \Rightarrow \text{real}
assumes slt: sublinear t
shows sub-distrib t
proof
fix P::\'s \Rightarrow real and s::\'s
assume sP: sound P
moreover have 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 slt, where a=1 and b=0 and c=1, simplified])
qed

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

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

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

show a * t P s + b * t Q s ⊕ c ≤ t (λs. a * P s′ + b * Q s′ ⊕ c) s
proof(cases c = 0)
  case True note z = this
  from stabPQ have ∃s. 0 ≤ a * t P s + b * t Q s by(auto)
  moreover from sabPQ
  have ∃s. 0 ≤ a * P s + b * Q s by(auto)
  ultimately show ?thesis by(simp add:z notm)

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

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

from nz
have a * t P s + b * t Q s ⊕ c =
  (c * inverse c) * a * t P s +
  (c * inverse c) * b * t Q s ⊕ c
  by(simp)
also
have ... = c * (inverse c * a * t P s) +
  c * (inverse c * b * t Q s) ⊕ c
  by(simp add:field-simps)
also from nnc
have ... = c * (inverse c * a * t P s + inverse c * b * t Q s ⊕ 1)
  by(simp add:distrib-left tminus-left-distrib)
also { }
also from nni and notm
have inverse c * (a + t P s + b * t Q s) ≤ 
   inverse c * (t (λs. a + P s + b * Q s) s)
   by(blast intro:mult-left-mono)
also from nni ht subPQ
have (inverse c * a) * t P s + (inverse c * b) * t Q s ⊓ 1 ≤ 
   t (λs. (inverse c * a) * P s + (inverse c * b) * Q s) s ⊓ 1
   by(rule tminus-left-mono)
also { }
from sdt si
have t (λs. (inverse c * a) * P s + (inverse c * b) * Q s) s ⊓ 1 ≤ 
   t (λs. (inverse c * a) * P s + (inverse c * b) * Q s ⊓ 1) s
   by(blast)
}
finally
have c * (inverse c * a * t P s + inverse c * b * t Q s ⊓ 1) ≤ 
   c * t (λs. inverse c * a * P s + inverse c * b * Q s ⊓ 1) s
   using nnc by(blast intro:mult-left-mono)
also from nnc ht sim
have c * t (λs. inverse c * a * P s + inverse c * b * Q s ⊓ 1) s
   = t (λs. c * (inverse c * a * P s + inverse c * b * Q s ⊓ 1)) s
   by(simp add:scalingD healthy-scalingD)
also from nnc
have ... = t (λs. c * (inverse c * a * P s) + 
   c * (inverse c * b * Q s) ⊓ c) s
   by(simp add:distrib-left tminus-left-distrib)
also have ... = t (λs. (c * inverse c) * a * P s + 
   (c * inverse c) * b * Q s ⊓ c) s
   by(simp add:field-simps)
finally
show a * t P s + b * t Q s ⊓ c ≤ t (λs'. a * P s' + b * Q s' ⊓ c) s
   using nz by(simp)
qed
qed

3.2. EXPECTATION TRANSFORMERS

Sub-conjunctivity

definition
   sub-conj :: (′s ⇒ real) ⇒ ′s ⇒ real ⇒ bool
where
   sub-conj t ≡ ∀ P Q. (sound P ∧ sound Q) −→ 
       t P & & t Q ⊨ t (P & & Q)
lemma sub-conjI[intro]:
\[
\begin{aligned}
& \forall P \ Q. \ [ \ \text{sound } P; \ \text{sound } Q ] \Rightarrow \\
& \quad t P \ &\& \ t Q \ \vdash \ t ( P \ &\& \ Q )
\end{aligned}
\] \implies \ sub\-conj \ t

unfolding sub-conj-def by(simp)

lemma sub-conjD[dest]:
\[
\begin{aligned}
& [ \ sub\-conj \ t; \ \text{sound } P; \ \text{sound } Q ] \Rightarrow \\
& \quad t P \ &\& \ t Q \ \vdash \ t ( P \ &\& \ Q )
\end{aligned}
\] \implies \ sub\-conj \ t

unfolding sub-conj-def by(simp)

lemma sub-conj-wp-twice:
fixes f :: (′s ⇒ real) ⇒ (′s ⇒ real)
assumes all: \( \forall s. \ sub\-conj (f s) \)
shows sub-conj (λP s. f s P s)

proof (rule sub-conjI, rule le-funI)
fix P::′s ⇒ real and Q::′s ⇒ real and s
assume sP: sound P and sQ: sound Q
have (λs. f s P s) && (λs. f s Q s) s = (f s P && f s Q) s
  by(simp add:exp-conj-def)
also { from all have sub-conj (f s) by(blast)
  with sP and sQ have (f s P && f s Q) s ≤ f s (P && Q) s
  by(blast)
}
finally show ((λs. f s P s) && (λs. f s Q s)) s ≤ f s (P && Q) s .

qed

Sublinearity implies sub-conjunctivity:

lemma sublinear-sub-conj:
fixes t::′s ⇒ real and s::′s ⇒ real
assumes slt: sublinear t
shows sub-conj t

proof (rule sub-conjI, rule le-funI, unfold exp-conj-def pconj-def)
fix P::′s ⇒ real and Q::′s ⇒ real and s::′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:
\[
\begin{aligned}
& [ \ equiv\-trans \ t \ u; \ sublinear \ t; \ healthy \ t ] \ \Rightarrow \ sublinear \ u
\end{aligned}
\]

by (iprover 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

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

where

additive t ≡ ∀ P Q. (sound P ∧ sound Q) →
   t (λs. P s + Q s) = (λs. t P s + t Q s)

lemma additiveD:

[ additive t; sound P; sound Q ] ⇒
   t (λs. P s + Q s) = (λs. t P s + t Q s)

by (simp add:additive-def)

lemma additiveI[intro]:

[ ∧ P Q s. [ sound P; sound Q ] ⇒
   t (λs. P s + Q s) s = t P s + t Q s ] ⇒

unfolding additive-def by (blast)

Additivity is strictly stronger than sub-additivity.

lemma additive-sub-add:

additive t =⇒ sub-add t

by (simp add:sub-addI additiveD)

The additivity property extends to finite summation.

lemma additive-sum:

fixes S::'s set

assumes additive: additive t
   and healthy: healthy t
   and finite: finite S
   and sPz: ∃z. sound (P z)

shows t (∑ y∈S. P y x) = (∑ y∈S. t (P y) x)

proof (rule finite-induct, simp-all add:assms)

fix z::'s and T::'s set

assume finT: finite T
   and IH: t (∑ y∈T. P y x) = (∑ y∈T. t (P y) x)

from additive sPz

have t (λx. P z x + (∑ y∈T. P y x)) =
   (λx. t (P z) x + t (λx. ∑ y∈T. P y x) x)

by (auto intro!: sum-sound additiveD)

also from IH
have \( \ldots = (\lambda x . t \, (P \, z) \, x + (\sum y \in T . t \, (P \, y) \, x)) \)
by(simp)

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

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

lemma additive-delta-split:
fixes \( t' : ('s : \text{finite} \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real} \)
assumes additive: additive \( t \)
and \( \text{ht}: \text{healthy} \, t \)
and \( sP: \text{sound} \, P \)
shows \( t \, P \, x = (\sum y \in \text{UNIV}. \, P \, y \ast t \, (\lambda z . \, z = y \, x) \)
proof –
have \( \wedge x . ((\sum y \in \text{UNIV}. \, P \, y \ast (\lambda z . \, z = y \, x)) = (\sum y \in \text{UNIV}. \, \text{if} \, y = x \, \text{then} \, P \, y \, \text{else} \, 0) \)
by (rule sum.cong) auto
also have \( \wedge x . \ldots x = P \, x \)
by(simp add:sum.delta)
finally
have \( t \, P \, x = (\lambda x . \sum y \in \text{UNIV}. \, P \, y \ast (\lambda z . \, z = y \, x) \, x) \)
by(simp)
also { \}
from \( sP \) have \( \wedge z . \text{sound} \, (\lambda a . \, P \, z \ast (\lambda a . \, za = z \, a)) \)
by(auto intro!:mult-sound)
hence \( t \, (\lambda x . \sum y \in \text{UNIV}. \, P \, y \ast (\lambda z . \, z = y \, x) \, x) = (\sum y \in \text{UNIV}. \, t \, (\lambda x . \, P \, y \ast (\lambda z . \, z = y \, x) \, x)) \)
by(subst additive-sum, simp-all add:assms)
} \}
also from \( sP \)
have \( (\sum y \in \text{UNIV}. \, t \, (\lambda x . \, P \, y \ast (\lambda z . \, z = y \, x) \, x) = (\sum y \in \text{UNIV}. \, P \, y \ast t \, (\lambda z . \, z = y \, x)) \)
by(subst scalingD[OF healthy-scalingD, OF \text{ht}], auto)
finally show \( t \, P \, x = (\sum y \in \text{UNIV}. \, P \, y \ast t \, (\lambda z . \, z = y \, x)) \).
qed

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

lemma additive-guard-split:
fixes \( t' : ('s : \text{finite} \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real} \)
assumes additive: additive \( t \)
and \( \text{ht}: \text{healthy} \, t \)
and \( sP: \text{sound} \, P \)
shows \( t \, P \, x = (\sum y \in \{s . \, G \, s\}. \, P \, y \ast t \, (\lambda z . \, z = y \, x) \) + 
(\sum y \in \{s \setminus G \, s\}. \, P \, y \ast t \, (\lambda z . \, z = y \, x)) \)
proof –
3.2. EXPECTATION TRANSFORMERS

from assms
have \( t \ P \ x = (\sum_{y \in \text{UNIV}} P \ y \times t \ \lambda z. \ z = y \times x) \)
  by (rule additive-delta-split)
also 
  have \( \text{UNIV} = \{ \text{s. G s} \} \cup \{ \text{s. \neg G s} \} \)
  by (auto)
  hence \( (\sum_{y \in \text{UNIV}} P \ y \times t \ \lambda z. \ z = y \times x) = \)
    \( (\sum_{y \in \{ \text{s. G s} \} \cup \{ \text{s. \neg G s} \}. P \ y \times t \ \lambda z. \ z = y \times x) \)
  by (simp)

also have \( (\sum_{y \in \{ \text{s. G s} \} \cup \{ \text{s. \neg G s} \}. P \ y \times t \ \lambda z. \ z = y \times x) = \)
  \( (\sum_{y \in \{ \text{s. G s} \}. P \ y \times t \ \lambda z. \ z = y \times x) + \)
  \( (\sum_{y \in \{ \text{s. \neg G s} \}. P \ y \times t \ \lambda z. \ z = y \times x) \)
  by (auto intro: sum.union_disjoint)
finally show \(?\text{thesis}\).
qed

Maximality

definition maximal :: (('a \Rightarrow real) \Rightarrow 'a \Rightarrow real) \Rightarrow bool
where
  maximal t \equiv \forall c. \ 0 \leq c \longrightarrow t (\lambda-. \ c) = (\lambda-. \ c)

lemma maximalI[intro]:
  \([ \ \forall c. \ 0 \leq c \Longrightarrow t (\lambda-. \ c) = (\lambda-. \ c) \] \Longrightarrow \text{maximal t}
  by (simp add: maximal-def)

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

A transformer that is both additive and maximal is deterministic:

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

lemma determI[intro]:
  \([ \text{additive t}; \text{maximal t} \] \Longrightarrow \text{determ t}
  by (simp add: determ-def)

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

lemma determ-maximalD[intro]:
  determ t \Longrightarrow maximal t
  by (simp add: determ-def)
For a fully-deterministic transformer, a transformed standard expectation, and its transformed negation are complementary.

\textbf{Lemma determ-negate:}
\begin{itemize}
  \item \textbf{Assumes} determ: determ \( t \)
  \item \textbf{Shows} \( t P s + t \overline{N} P s = 1 \)
\end{itemize}
\begin{proof}
\item \textbf{Have} \( t P s + t \overline{N} P s = t (\lambda s. P s + \overline{N} P s) s \)
\item \textbf{By} \((\text{simp add: additiveD determ determ-additiveD})\)
\item \textbf{Also} \{\begin{itemize}
  \item \textbf{Have} \( \lambda s. P s + \overline{N} P s = 1 \)
  \item \textbf{By} \((\text{case-tac P s, simp-all})\)
  \item \textbf{Hence} \( t (\lambda s. P s + \overline{N} P s) = t (\lambda s. 1) \)
  \item \textbf{By} \((\text{simp})\)
\end{itemize}\}
\item \textbf{Also have} \( t (\lambda s. 1) = (\lambda s. 1) \)
\item \textbf{By} \((\text{simp add: maximalD determ determ-maximalD})\)
\item \textbf{Finally show} \( ?\text{thesis} \).
\end{proof}

\textbf{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.

\textbf{Lemma entails-combine:}
\begin{itemize}
  \item \textbf{Assumes} \( wp1: P \vdash t R \)
  \item \textbf{And} \( wp2: Q \vdash t S \)
  \item \textbf{And} \( sc: \text{sub-conj} t \)
  \item \textbf{And} \( sR: \text{sound} R \)
  \item \textbf{And} \( sS: \text{sound} S \)
  \item \textbf{Shows} \( P \land Q \vdash t (R \land S) \)
\end{itemize}
\begin{proof}
\item \textbf{From} \( wp1 \text{ and } wp2 \) \textbf{Have} \( P \land Q \vdash t R \land t S \)
\item \textbf{By} \((\text{blast intro: entails-frame})\)
\item \textbf{Also from} \( sc \text{ and } sR \text{ and } sS \text{ have} \) \( \leq t (R \land S) \)
\item \textbf{By} \((\text{rule sub-conjD})\)
\item \textbf{Finally show} \( ?\text{thesis} \).
\end{proof}

These allow mismatched results to be composed

\textbf{Lemma entails-strengthen-post:}
\begin{itemize}
  \item \([ P \vdash t Q; \text{healthy} t; R; \text{sound} Q \] \implies P \vdash t R \)
  \item \textbf{By} \((\text{blast intro: entails-trans})\)
\end{itemize}
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 t Q and h: healthy t
and sQ: sound Q and pos: 0 \leq c
shows (\lambda s. c \cdot P s) \vdash \lambda s. c \cdot Q s

proof(rule le-funI)
fix s
from pos and wp have c \cdot P s \leq c \cdot t Q s
by(auto intro: mult-left-mono)
with sQ pos 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: healthy t and wp: \forall s. P s \implies 1 \leq t «Q» s
shows «P» \vdash t «Q»

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

The other direction works too.

lemma fold-premise:
assumes ht: healthy t and wp: «P» \vdash t «Q»
shows \forall s. P s \implies 1 \leq t «Q» s

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

**Lemma conj-post:**

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

Similar to \[\text{healthy } ?t; \bigwedge s. ?P s \implies 1 \leq ?t \langle \mathit{Q} \rangle s \implies \langle \mathit{P} \rangle \vdash ?t \langle \mathit{Q} \rangle\], but more general.

**Lemma entails-pconj-assumption:**

assumes \( f \): feasible \( t \) and \( wP \): \( \forall s. P s \implies Q s \leq t R s \)

and \( uQ \): unitary \( Q \) and \( uR \): unitary \( R \)

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

unfolding \( \mathit{exp-conj-def} \)

**Proof (rule entailsI)**

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

**Proof (cases \( P s \))**

- **Case True**
  - moreover from \( uQ \) have \( 0 \leq Q s \) by (auto)
  - ultimately show \( \text{?thesis} \) by (simp add: pconj-lone \( wP \))

- **Case False**
  - moreover from \( uQ \) have \( Q s \leq 1 \) by (auto)
  - ultimately show \( \text{?thesis} \) using \( \text{assms} \) 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 \textit{HOL.Inductive}), is that we do not have a complete lattice.

Finding a lower bound is easy (it’s \( \lambda s. 0 :: \prime b \)), but as we do not insist on any global bound on expectations (and work directly in HOL’s real type, rather than extending it with a point at infinity), there is no top element. We solve the problem by defining the least (greatest) fixed point, restricted to an internally-bounded set, allowing us to substitute this bound for the top element. This works as long as the set contains at least one fixed point, which appears as an extra assumption in all the theorems.
This also works semantically, thanks to the definition of healthiness. Given a healthy transformer parameterised by some sound expectation: \( t \). Imagine that we wish to find the least fixed point of \( t \ P \). In practice, \( t \) is generally doubly healthy, that is \( \forall P. \text{sound} \ P \implies \text{healthy} \ (t \ P) \) and \( \forall Q. \text{sound} \ Q \implies \text{healthy} \ (\lambda P. t \ P \ Q) \). 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** \( \text{Inf-exp} :: 's \text{ expect set} \Rightarrow 's \text{ expect} \)

**lemma** \( \text{Inf-exp-lower}: \)

\[
[ P \in S; \forall P \in S. \text{neg} \ P ] \implies \text{Inf-exp} \ S \leq P
\]

**unfolding** \( \text{Inf-exp-def} \)

by \((\text{intro le-funI clInf-lower bdd-belowI[where m=0], auto})\)

**lemma** \( \text{Inf-exp-greatest}: \)

\[
[ S \neq \{}; \forall P \in S. Q \leq P ] \implies Q \leq \text{Inf-exp} \ S
\]

**unfolding** \( \text{Inf-exp-def} \)

by \((\text{auto intro le-funI cInf-greatest})\)

**definition** \( \text{Sup-exp} :: 's \text{ expect set} \Rightarrow 's \text{ expect} \)

**lemma** \( \text{Sup-exp-upper}: \)

\[
[ P \in S; \forall P \in S. \text{bounded-by} \ b \ P ] \implies P \leq \text{Sup-exp} \ S
\]

**unfolding** \( \text{Sup-exp-def} \)

by \((\text{cases S={}}, \text{simp-all, intro le-funI cSup-upper bdd-aboveI[where M=b], auto})\)

**lemma** \( \text{Sup-exp-least}: \)

\[
[ \forall P \in S. P \leq Q; \text{neg} \ Q ] \implies \text{Sup-exp} \ S \leq Q
\]

**unfolding** \( \text{Sup-exp-def} \)

by \((\text{cases S={}}, \text{auto intro le-funI OF cSup-least})\)

**lemma** \( \text{Sup-exp-sound}: \)

**assumes** \( sS: \bigwedge P. P \in S \implies \text{sound} \ P \)

**and** \( bS: \bigwedge P. P \in S \implies \text{bounded-by} \ b \ P \)

**shows** \( \text{sound} \ (\text{Sup-exp} \ S) \)

**proof** \((\text{cases S={}}, \text{simp add:Sup-exp-def}, \text{blast, intro soundI2 bounded-byI2 negI2})\)

**assume** \( \text{neS}: S \neq \{\} \)

**then** obtain \( P \) where \( \text{Pin: P} \in S \) by \((\text{auto})\)

**with** \( sS \ bS \) have \( \text{nP: neg} \ P \text{ bounded-by} \ b \ P \) by \((\text{auto})\)

**hence** \( \text{nb: 0} \leq b \) by \((\text{auto})\)

from \( bS \) \( \text{nb} \) show \( \text{Sup-exp} \ S \vdash \lambda s. b \)

by \((\text{auto intro:Sup-exp-least})\)

from \( nP \) have \( \lambda s. 0 \vdash P \) by \((\text{auto})\)

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

**definition** \text{lfp-exp} :: \( 's \text{ trans } \Rightarrow 's \text{ expect} \)

where \text{lfp-exp} \( t = \text{Inf-exp } \{ P. \text{ sound } P \land t \, P \leq P \} \)

**lemma** \text{lfp-exp-lowerbound}:

\[
[ t \, P \leq P; \text{ sound } P ] \Rightarrow \text{lfp-exp} \, t \leq P
\]

unfolding \text{lfp-exp-def} by (auto intro: Inf-exp-lower)

**lemma** \text{lfp-exp-greatest}:

\[
[ \forall P. [ t \, P \leq P; \text{ sound } Q ] \Rightarrow Q \leq P; t \, R \vdash R; \text{ sound } R ] \Rightarrow Q \leq \text{lfp-exp} \, t
\]

unfolding \text{lfp-exp-def} by (auto intro: Inf-exp-greatest)

**lemma** \text{feasible-lfp-exp-sound}:

\( \text{feasible } t \Rightarrow \text{ sound } (\text{lfp-exp} \, t) \)

by (intro soundI2 bounded-byI2 nnegI2, auto intro: lfp-exp-lowerbound lfp-exp-greatest)

**lemma** \text{lfp-exp-sound}:

assumes \( fR: t \, R \vdash R \) and \( sR: \text{ sound } R \)

shows \( \text{ sound } (\text{lfp-exp} \, t) \)

proof (intro soundI2)
from \( fR \, sR \) have \( \text{lfp-exp} \, t \vdash R \)
by (auto intro: lfp-exp-lowerbound)
also from \( sR \) have \( R \vdash \lambda s. \text{ bound-of } R \) by (auto)
finally show \( \text{ bounded-by } (\text{bound-of } R) \, (\text{lfp-exp} \, t) \) by (auto)
from \( fR \, sR \) show \( \text{ nneg } (\text{lfp-exp} \, t) \) by (auto intro: lfp-exp-greatest)
qed

**lemma** \text{lfp-exp-bound}:

\( (\forall P. \text{ unitary } P \Rightarrow \text{ unitary } (t \, P)) \Rightarrow \text{ bounded-by } 1 \, (\text{lfp-exp} \, t) \)

by (auto intro: lfp-exp-lowerbound)

**lemma** \text{lfp-exp-unitary}:

\( (\forall P. \text{ unitary } P \Rightarrow \text{ unitary } (t \, P)) \Rightarrow \text{ unitary } (\text{lfp-exp} \, t) \)

proof (intro unitaryI![OF lfp-exp-sound lfp-exp-bound], simp-all)
assume \( IH: (\forall P. \text{ unitary } P \Rightarrow \text{ unitary } (t \, P)) \)

have \( \text{ unitary } (\lambda s. 1) \) by (auto)
with \( IH \) have \( \text{ unitary } (t \, (\lambda s. 1)) \) by (auto)
thus \( t \, (\lambda s. 1) \vdash (\lambda s. 1) \) by (auto)
show \( \text{ sound } (\lambda s. 1) \) by (auto)
qed

**lemma** \text{lfp-exp-lemma2}:

fixes \( t: 's \text{ trans} \)
assumes \( st: (\forall P. \text{ sound } P \Rightarrow \text{ sound } (t \, P)) \)
and \( mt: \text{ mono-trans } t \)
and \( fR : t R \vdash R \) and \( sR : \text{sound } R \).

\[ \text{shows } t \left( \text{lfp-exp } t \right) \leq \text{lfp-exp } t \]

\[ \text{proof}(\text{rule lfp-exp-greatest[of } t, \text{OF } - - fR sR) \]

\[ \text{from } fR sR \text{ show } \text{sound } \left( t \left( \text{lfp-exp } t \right) \right) \text{ by(auto intro:lfp-exp-sound st) \]

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

assume \( fP : t P \vdash P \) and \( sP : \text{sound } P \)

hence \( \text{lfp-exp } t \vdash P \) by(\text{rule lfp-exp-lowerbound})

with \( fP sP \text{ have } t \left( \text{lfp-exp } t \right) \vdash t P \text{ by(auto intro:mono-transD[OF mt]} \text{lfp-exp-sound} \]

also note \( fP \)

finally show \( t \left( \text{lfp-exp } t \right) \vdash P \).

\[ \text{qed} \]

\[ \text{lemma lfp-exp-lemma3:} \]

\[ \text{assumes } st : \bigwedge P. \text{sound } P \Rightarrow \text{sound } \left( t P \right) \]

\[ \text{and mt: mono-trans } t \]

\[ \text{and } fR : t R \vdash R \text{ and } sR : \text{sound } R \]

\[ \text{shows } \text{lfp-exp } t \leq t \left( \text{lfp-exp } t \right) \]

\[ \text{by}(\text{iprover intro:}\text{lfp-exp-lowerbound } \text{lfp-exp-sound } \text{lfp-exp-lemma2 assms} \]

\[ \text{mono-transD[OF mt]} \text{lfp-exp-sound} \]

\[ \text{lemma lfp-exp-unfold:} \]

\[ \text{assumes } nt : \bigwedge P. \text{sound } P \Rightarrow \text{sound } \left( t P \right) \]

\[ \text{and mt: mono-trans } t \]

\[ \text{and } fR : t R \vdash R \text{ and } sR : \text{sound } R \]

\[ \text{shows } \text{lfp-exp } t = t \left( \text{lfp-exp } t \right) \]

\[ \text{by}(\text{iprover intro:antisym } \text{lfp-exp-lemma2 } \text{lfp-exp-lemma3 assms}) \]

\[ \text{definition gfp-exp :: 's trans \Rightarrow 's expect} \]

\[ \text{where } gfp-exp t = \text{Sup-exp } \{ P : \text{unitary } P \wedge P \leq t P \} \]

\[ \text{lemma gfp-exp-upperbound:} \]

\[ \big[ P \leq t P ; \text{unitary } P \big] \Rightarrow P \leq gfp-exp t \]

\[ \text{by(auto simp:}\text{gfp-exp-def intro:Sup-exp-upper}) \]

\[ \text{lemma gfp-exp-least:} \]

\[ \big[ \bigwedge P. \big[ P \leq t P ; \text{unitary } P \big] \Rightarrow P \leq Q ; \text{unitary } Q \big] \Rightarrow gfp-exp t \leq Q \]

\[ \text{unfolding gfp-exp-def by(auto intro:Sup-exp-least)} \]

\[ \text{lemma gfp-exp-bound:} \]

\[ \big( \bigwedge P. \text{unitary } P \Rightarrow \text{unitary } \left( t P \right) \big) \Rightarrow \text{bounded-by } 1 \left( gfp-exp t \right) \]

\[ \text{unfolding gfp-exp-def by(\text{rule bounded-byI2[OF Sup-exp-least]}, auto)} \]

\[ \text{lemma gfp-exp-nneg[iff]:} \]

\[ \text{nneg } (gfp-exp t) \]

\[ \text{proof(\text{intro nnegI2, simp add:gfp-exp-def, cases})} \]

\[ \text{assume empty: } \{ P. \text{unitary } P \wedge P \vdash t P \} = \{} \]

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

lemma gfp-exp-unitary:
(\forall P. unitary P \implies unitary (t P)) \implies unitary (gfp-exp t)
by(prover intro:gfp-exp-nneg gfp-exp-bound unitaryI2)

lemma gfp-exp-lemma2:
assumes ft: \(\forall P. unitary P \implies unitary (t P)\)
and mt: \(\forall P Q. [ unitary P; unitary Q; P \vdash Q ] \implies t P \vdash t Q\)
shows gfp-exp t \leq t (gfp-exp t)
proof(rule gfp-exp-least)
say unitary (t (gfp-exp t)) by(auto intro:gfp-exp-unitary ft)
fix P
assume fp: P \leq t P and up: 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: \(\forall P. unitary P \implies unitary (t P)\)
and mt: \(\forall P Q. [ unitary P; unitary Q; P \vdash Q ] \implies t P \vdash t Q\)
shows t (gfp-exp t) \leq gfp-exp t by(auto intro:gfp-exp-upperbound unitary-sound
gfp-exp-unitary gfp-exp-lemma2 assms)

lemma gfp-exp-unfold:
(\forall P. unitary P \implies unitary (t P)) \implies (\forall P Q. [ unitary P; unitary Q; P \vdash Q
\implies t P \vdash t Q\]) \implies gfp-exp t = t (gfp-exp t)
by(prover 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 \(\lambda \cdot . \emptyset::'c\), 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 \( S = (\lambda P. \text{Inf-exp} \{ t P \mid t. t \in S \}) \)

lemma Inf-trans-lower:
\[
[ t \in S; \forall u \in S. \forall P. \text{sound } P \rightarrow \text{sound } (u P) ] \Rightarrow \text{le-trans } (\text{Inf-trans } S) t
\]

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

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

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

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

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

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

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

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

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

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

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

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

from $\text{neS}$ obtain $t$ where $\text{in}: t \in S$ by(auto)

with $\text{fS}$ have $t$: feasible $t$ by(auto)

with $\text{bP nP}$ have $\lambda s. \; 0 \vdash t \; P$ by(auto)

also {
  from $\text{bP nP}$ have $\text{sound} \; P$ by(auto)

  with $\text{in fS}$ have $t \; P \vdash \text{Sup-trans} \; S \; P$
    by(auto intro!:Sup-trans-upper2)
}

finally show $\text{nneq} \; (\text{Sup-trans} \; S \; P)$ by(auto)

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

qed

definition $\text{lfp-trans} :: \; (\prime \prime \text{trans} \Rightarrow \; \prime \prime \text{trans}) \Rightarrow \; \prime \prime \text{trans}$

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

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 $\text{lfp-trans-greatest}$:

\[
[ \; \bigwedge t. \; [ \; \text{le-trans} \; (T \; t) \; t; \; \bigwedge P. \; \text{sound} \; P \Rightarrow \; \text{sound} \; (t \; P) \; ] \Rightarrow \; \text{le-trans} \; u \; t; \;
\bigwedge P. \; \text{sound} \; P \Rightarrow \; \text{sound} \; (v \; P); \; \text{le-trans} \; (T \; v) \; v \; ] \Rightarrow \; \text{le-trans} \; u \; (\text{lfp-trans} \; T)
\]

unfolding $\text{lfp-trans-def}$

by(rule Inf-trans-greatest, auto)

lemma $\text{lfp-trans-sound}$:

fixes $P \; Q$: $\prime \prime \text{expect}$

assumes $sP$: $\text{sound} \; P$

and $sf$: $\text{le-trans} \; (T \; v) \; v$

and $sv$: $\bigwedge P. \; \text{sound} \; P \Rightarrow \; \text{sound} \; (v \; P)$

shows $\text{sound} \; (\text{lfp-trans} \; T \; P)$

proof(intro soundI2 bounded-byI2 nneqI2)

from $fv \; sv$ have $\text{le-trans} \; (\text{lfp-trans} \; T) \; v$

by(iprover 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(iprover)

  hence $v \; P \vdash \; \lambda s. \; \text{bound-of} \; (v \; P)$ by(auto)
}

finally show $\text{lfp-trans} \; T \; P \vdash \lambda s. \; \text{bound-of} \; (v \; P)$

have $\text{le-trans} \; (\lambda P \; s. \; 0) \; (\text{lfp-trans} \; T)$

proof(intro lfp-trans-greatest)
3.3. INDUCTION

fix t::'s trans
assume \( \forall P. \text{sound } P \implies \text{sound } (t P) \)
hence \( \forall P. \text{sound } P \implies \lambda s. \bot + t P \) by(auto)
thus le-trans \( (\lambda P s. \bot) t \) by(auto)
next
fix P::'s expect
assume sound P thus sound \( (v P) \) by(rule sv)
next
show le-trans \( (T v) v \) by(rule f2)
qed
with sP show \( \lambda s. \bot \vdash \lambda f. \text{lfp-trans } T P \) by(auto)
qed

lemma lfp-trans-unitary:
fixes P Q::'s expect
assumes uP: unitary P
and fP: le-trans \( (T v) v \)
and sP: \( \forall P. \text{sound } P \implies \text{sound } (v P) \)
and fT: le-trans \( (T (\lambda P s. \text{bound-of } P)) (\lambda P s. \text{bound-of } P) \)
shows unitary \( (\lambda f. \text{lfp-trans } T P) \)
proof(rule unitaryI)
from unitary-sound[OF uP] fP sP show sound \( (\lambda f. \text{lfp-trans } T P) \) by(rule lfp-trans-sound)
show bounded-by \( I \) \( (\lambda f. \text{lfp-trans } T P) \)
proof(rule bounded-byI12)
from fT have le-trans \( (\lambda f. \text{lfp-trans } T) (\lambda P s. \text{bound-of } P) \)
by(auto intro: lfp-trans-lowerbound)
with uP have \( \lambda f. \text{lfp-trans } T P \vdash \lambda s. \text{bound-of } P \) by(auto)
also from uP have \( \vdash \lambda s. 1 \) by(auto)
finally show \( \lambda f. \text{lfp-trans } T P \vdash \lambda s. 1 \).
qed
qed

lemma lfp-trans-lemma2:
fixes v::'s trans
assumes mono: \( \forall t u. \begin{array}{l}
\forall P. \text{sound } P \implies \text{sound } (t u) \\
\lambda P. \text{sound } P \implies \text{sound } (u P) \\
\end{array} \implies \text{le-trans } (T t) (T u) \)
and nT: \( \forall t P. \begin{array}{l}
\forall Q. \text{sound } Q \implies \text{sound } (t Q) \\
\text{sound } P \end{array} \implies \text{sound } (T t) (T u) \)

and fP: le-trans \( (T v) v \)
and sP: \( \forall P. \text{sound } P \implies \text{sound } (v P) \)
shows le-trans \( (T (\lambda f. \text{lfp-trans } T)) (\lambda f. \text{lfp-trans } T) \)
proof(rule lfp-trans-greatest[where \( T=T \) and \( v=v \), simp-all add:assms])
fix t::'s trans and P::'s expect
assume ft: le-trans \( (T t) t \) and st: \( \forall P. \text{sound } P \implies \text{sound } (t P) \)
hence le-trans \( (\lambda f. \text{lfp-trans } T) t \) by(auto intro: lfp-trans-lowerbound)
with ft st have le-trans \( (T (\lambda f. \text{lfp-trans } T)) (T t) \)
by(intro mono intro: lfp-trans-sound f2 s2)
also note \( ft \)
finally show \( le-trans \ (T \ (lfp-trans \ T)) \ t \).
\[ \text{qed} \]

**Lemma lfp-trans-lemma3:**

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

**Lemma lfp-trans-unfold:**

fixes \( P : ' s \ expect \)
assumes mono: \( \forall t u. \ [ \ le-trans \ t u; \ \forall P. \ sound \ P \implies \ sound \ (t \ P); \]
\( \forall P. \ sound \ P \implies \ sound \ (u \ P) \ ] \implies \ le-trans \ (T \ t) \ (T \ u) \)
and \( fu: \ le-trans \ (T \ v) \ v \)
and \( su: \ \forall P. \ sound \ P \implies \ sound \ (v \ P) \)
shows \( equiv-trans \ (lfp-trans \ T) \ (T \ (lfp-trans \ T)) \)
by (rule le-trans-antisym,
rule lfp-trans-lemma2[OF mono], (iprover intro:assms)+,
rule lfp-trans-lemma2[OF mono], (iprover intro:assms)+)
\[ \text{definition gfp-trans :: } (' s \ trans \implies ' s \ trans) \implies ' s \ trans \]
where \( gfp-trans \ T = \sup-trans \ { t. \ \forall P. \ unitary \ P \implies unitary \ (t \ P) } \land le-utrans \ t \ (T \ t) \)

**Lemma gfp-trans-upperbound:**

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

lemma gfp-trans-least:
\[
\begin{aligned}
&\forall t. \le-utrans (T t); \\
&P, \text{unitary } P \Rightarrow \text{unitary } (t P) \Longrightarrow \le-utrans t w; \\
&P, \text{unitary } P \Rightarrow \text{unitary } (u P) \Longrightarrow \\
&\le-utrans (gfp-trans T) u
\end{aligned}
\]

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

lemma gfp-trans-unitary:
fixes P :: 's expect
assumes uP : unitary P
shows unitary (gfp-trans T P)
proof
(intro unitaryI2 nnegI2 bounded-byI2)
show \(\lambda s. 1\) by(auto)
qed

let \(?S\) = \{ \(t P\) | \(t\). \(t\) \in \{ \(t P\). \(\forall Q\). \text{unitary } Q \Rightarrow \text{unitary } (t Q) \land \le-utrans t (T t)\}\} 

show \(\lambda s. 0\) \not\preceq gfp-trans T P 
unfolding gfp-trans-def Sup-trans-def
proof(cases)
assume empty: \(?S\) = {}
show \(\lambda s. 0\) \not\preceq \(\lambda s. 1\) by(auto)
qed

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

finally show \(\lambda s. 0\) \preceq Sup-exp \(?S\) .
qed

qed

lemma gfp-trans-lemma2:
assumes mono: \(\forall t u. \le-utrans t w; \forall P, \text{unitary } P \Rightarrow \text{unitary } (t P); \\
&P, \text{unitary } P \Rightarrow \text{unitary } (u P) \Longrightarrow \le-utrans (T t) (T u) \\
\)

and hT: \(\forall P. \forall Q. \text{unitary } Q \Rightarrow \text{unitary } (t Q); \text{unitary } P \Rightarrow \text{unitary } (T t P) \)

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

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

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

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

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

lemma gfp-pulldown:
fixes P::’s expect
assumes tailcall: \( \forall u P. \text{unitary } P \Rightarrow T u P = t P (u P) \)
and ft: \( \forall t P. \ [ \forall Q. \text{unitary } Q \Rightarrow \text{unitary } (t Q); \ \text{unitary } P \ ] \Rightarrow \text{unitary } (T t P) \)
and ft: \( \forall P Q. \text{unitary } P \Rightarrow \text{unitary } Q \Rightarrow \text{unitary } (t P Q) \)
and mt: \( \forall P Q R. \ [ \text{unitary } P; \ \text{unitary } Q; \ \text{unitary } R; \ Q \vdash R \ ] \Rightarrow t P Q \vdash t P R \)
and uP: \( \forall u P. \text{unitary } P \)
and monoT: \( \forall t u. \ [ \text{le-utrans } t u; \ \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P); \ \forall P. \text{unitary } P \Rightarrow \text{unitary } (u P) ] \Rightarrow \text{le-utrans } (T t) (T u) \)
shows gfp-trans T P = gfp-exp (t P) (is ?X P = ?Y P)
3.3. INDUCTION

proof (rule antisym)
show ?x P ≤ ?y P
proof (rule gfp-exp-upperbound)
  from mono T T uP have (gfp-trans T) P ≤ (T (gfp-trans T)) P
  by(auto intro!: le-utrans[OF gfp-trans-lemma2])
also from uP have (T (gfp-trans T)) P = t P (gfp-trans T P) by(rule tailcall)
finally show gfp-trans T P ⊢ t P (gfp-trans T P).
from aP gfp-trans-unitary show unitary (gfp-trans T P) by(auto)
qed
show ?y P ≤ ?x P
proof (rule le-utransD[OF gfp-trans-upperbound], simp-all add:assms)
  show le-utrans (λa. gfp-exp (t a)) (T (λa. gfp-exp (t a)))
  proof (rule le-utransI)
    fix Q::'s expect assume uQ: unitary Q
    with ft have \ \ ∧R. unitary R ⇒ unitary (t Q R) by(auto)
    with mt[OF uQ] have gfp-exp (t Q) = t Q (gfp-exp (t Q)) by(blast intro: le-utransD)
  finally show gfp-exp (t Q) ≤ T (λa. gfp-exp (t a)) Q by(simp)
  qed
fix Q::'s expect assume unitary Q
with ft have \ \ ∧R. unitary R ⇒ unitary (t Q R) by(auto)
thus unitary (gfp-exp (t Q)) by(rule gfp-exp-unitary)
qed

lemma lfp-pulldown:
fixes P::'s expect and t::'s expect ⇒ 's trans
  and T::'s trans ⇒ 's trans
assumes tailcall: \ \ ∧u. sound P ⇒ T u P = t P (u P)
  and st: \ \ ∧P. sound P ⇒ sound Q ⇒ sound (t P Q)
  and mt: \ \ ∧P. sound P ⇒ mono-trans (t P)
  and monoT: \ \ ∧u. \ \ le-trans t u; \ \ ∧P. sound P ⇒ sound (t P);
             \ \ ∧P. sound P ⇒ sound (u P) \ ⇒ le-trans (T t) (T u)
  and aT: \ \ ∧t P. \ \ ∧Q. sound Q ⇒ sound (t Q); sound P \ ⇒ sound (T t P)
  and ft: le-trans (T v) v
  and sv: \ \ ∧P. sound P ⇒ sound (v P)
  and sp: sound P
shows lfp-trans T P = lfp-exp (t P) (is ?x P = ?y P)
proof (rule antisym)
show ?y P ≤ ?x P
proof (rule lfp-exp-lowerbound)
  from sP have t P (lfp-trans T P) = (T (lfp-trans T)) P by(rule tailcall[antisym])
  also have (T (lfp-trans T)) P ≤ (lfp-trans T P)
  by(rule le-utransD[OF lfp-trans-lemma2[OF mono T]], (iprover intro:assms)+)
  finally show t P (lfp-trans T P) ≤ lfp-trans T P.
  from sP show sound (lfp-trans T P)
by (iprover intro: lfp-trans-sound assms)
qed

have \( \forall P. \text{sound } P \Rightarrow t P (v P) = T v P \) by (simp add: tailcall)
also have \( \forall P. \text{sound } P \Rightarrow \vdash v P \) by (auto intro: le-transD[OF fv])
finally have \( \text{fvP} \): \( \forall P. \text{sound } P \Rightarrow t P (v P) \vdash \vdash v P \).

have \( \text{svP} \): \( \forall P. \text{sound } P \Rightarrow \text{sound } (v P) \) by (rule sv)

show \( ?X P \leq ?Y P \)
proof (rule le-transD[OF lfp-trans-lowerbound, OF - sP])
 show le-trans (T (\( \lambda a. \text{lfp-exp} (t a) \)) (\( \lambda a. \text{lfp-exp} (t a) \)))
proof (rule le-transI)
fix \( P \): ’s expect
assume \( sP \)
show \( T (\lambda a. \text{lfp-exp} (t a)) P \vdash \vdash \text{lfp-exp} (t P) \) by (simp)
qed
fix \( P \): ’s expect
assume \( \text{sound } P \)
with \( \text{fvP svP} \) show \( \text{sound } (\text{lfp-exp} (t P)) \)
by (blast intro: lfp-exp-sound)
qed

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

lemma Inf-utrans-lower:
\[
\begin{array}{c}
\text{\( \mid t \in S; \forall t \in S. \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) \mid \Rightarrow \text{le-utrans } (\text{Inf-utrans } S) t \)}
\end{array}
\]
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:
\[
\begin{array}{c}
\text{\( \mid \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P); \forall u \in S. \text{ le-utrans } t u \mid \Rightarrow \text{le-utrans } t \)}
\end{array}
\]
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, \( \lambda P \ s. 0 : ':c \), or always succeeds, \( \lambda P \ s. 1 : ':c \).

definition Abort :: 's prog
where Abort ≡ λ ab 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. P (f s)

Seq is sequential composition.

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

\( PC \) is probabilistic choice between programs.

definition PC :: 's prog ⇒ ('s ⇒ real) ⇒ 's prog ⇒ 's prog
(- \oplus - [58,57,57] 57)
where PC a P b ≡ λab Q s. P s * a ab Q s + (1 - P s) * b ab Q s
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 the least or greatest fixed point.

**definition** Mu :: ('s prog ⇒ 's prog) ⇒ 's prog (binder µ 50)
**where** Mu(T) ≡ (λa. if a then lfp-trans (λt. T (Embed t) ab) else gfp-trans (λt. T (Embed t) ab))

repeat expresses finite repetition

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

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

**definition** SetDC :: ('a ⇒ 's prog) ⇒ ('s ⇒ 'a set) ⇒ 's prog
**where** SetDC f S ≡ λab P s. Inf ((λa. f a ab P s) · S s)

**syntax** - SetDC :: pttrn => ('s => 'a set) => 's prog => 's prog

**translations** ∏ x∈S. p == CONST SetDC (%x. p) S

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

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

**definition** SetPC :: ('a ⇒ 's prog) ⇒ ('s ⇒ 'a ⇒ real) ⇒ 's prog
**where** SetPC p p ≡ λab P s. ∑ a∈supp (p s). p s a * f a ab P s

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

**definition**
- Bind :: ('s ⇒ 'a) ⇒ ('a ⇒ 's prog) ⇒ 's prog
where

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

This gives us something like let syntax

```ml
let -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)
```

**definition flip :: ('a => 'b) => 'b => 'a => 'c**

```ml
where [simp]: flip f = (\ b a. f a b)
```

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

```ml
translations bind x at p in a => CONST SetPC (%x. a) (CONST flip (%x. p))
translations bind x from S in a => CONST SetDC (%x. a) S
```

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

```ml
fun assign-tr - [Const (name, -), arg] = 
    Const (Embedding.Apply, dummyT) $ Abs (s, dummyT,
        Syntax.const (suffix Record.updateN name) $ Abs (Name.uu-, dummyT, arg $ Bound 1) $ Bound 0)
    | assign-tr - ts = raise TERM (assign-tr, ts)
)
```

```ml
fun set-pc-tr - [Const (f, -), P] = 
    Const (SetPC, dummyT) $ Abs (v, dummyT,
        (Const (Embedding.Apply, dummyT) $ Abs (s, dummyT,
            Syntax.const (suffix Record.updateN f) $ Abs (Name.uu-, dummyT, Bound 2) $ Bound 0))) $ P
    | set-pc-tr - ts = raise TERM (set-pc-tr, ts)
```
These definitions instantiate the embedding as either weakest precondition (True) or weakest liberal precondition (False).

**syntax**

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

**translations**

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

**definition**

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

**definition**

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

If-Then-Else as a degenerate probabilistic choice.

**abbreviation**(input)

```
if-then-else :: ['s => bool, 's prog, 's prog] => 's prog
    (If - Then - Else - 58)
where
    If P Then a Else b == a _P§ b
```

Syntax for loops

**abbreviation**

```
do-while :: ['s => bool, 's prog] => 's prog
    (do - —→// (4 -) //od)
where
    do-while P a ≡ µ x. If P Then a ;; x Else Skip
```
4.1. A SHALLOW EMBEDDING OF PGCL IN HOL

4.1.2 Unfolding rules for non-recursive primitives

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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:
\(\text{healthy} (wp \text{Abort})\)
proof (rule healthy-parts)
fix \(b\) and \(P:::a \Rightarrow \text{real}\)
assume \(nP: \text{nneg} P\) and \(bP: \text{bounded-by} b P\)
thus \(\text{bounded-by} b (wp \text{Abort} P)\)
unfolding wp-eval by (blast)
show \(\text{nneg} (wp \text{Abort} P)\)
unfolding wp-eval by (blast)
next
fix \(P\) and \(Q:::a \text{ expect}\)
show \(wp \text{Abort} P \vdash wp \text{Abort} Q\)
unfolding wp-eval by (blast)
next
fix \(P\) and \(c\) and \(s:::a\)
show \(c * wp \text{Abort} P s = wp \text{Abort} (\lambda x. c * P s) s\)
unfolding wp-eval by (auto)
qed
CHAPTER 4. THE PGCL LANGUAGE

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

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

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

lemma healthy-wp-Seq:
fixes t :: 's prog and u
assumes ht: healthy (wp t) and hu: healthy (wp u)
shows healthy (wp (t;; u))
proof (rule healthy-parts, simp-all add: wp-eval)
fix b and P::'s ⇒ real
assume bounded-by b P and nneg P
with hu have bounded-by b (wp u P) and nneg (wp u P) by (auto)
with ht show bounded-by b (wp t (wp u P))
  and nneg (wp t (wp u P)) by (auto)
next
fix P::'s ⇒ real and c::real and s
assume pos: 0 ≤ c and sP: sound P
with ht and hu have c * wp t (wp u P) s = wp t (λs. c * wp u P s) s
  by (auto intro!: scalingD)
also with hu and pos and sP have ... = wp t (wp u (λs. c * P s)) s
  by (simp add: scalingD[OF healthy-scalingD])
finally show c * wp t (wp u P) s = wp t (wp u (λs. c * P s)) s .
qed

lemma nearly-healthy-wlp-Seq:
fixes t::'s prog and u
4.2. HEALTHINESS

assumes \( \text{ht: nearly-healthy (wlp t)} \) and \( \text{hu: nearly-healthy (wlp u)} \)
shows nearly-healthy (wlp (t ;; u))
proof(rule nearly-healthyI, simp-all add:wp-eval)
  fix \( b \) and \( P: s \Rightarrow \text{real} \)
  assume unitary \( P \)
  with \( \text{hu} \) have unitary (wlp u \( P \)) \( \text{by(auto)} \)
  with \( \text{ht} \) show unitary (wlp t (wlp u \( P \))) \( \text{by(auto)} \)
next
  fix \( P \) Q: \( s \Rightarrow \text{real} \) and \( s: \cdot \cdot \cdot \)
  assume \( \text{nQ: nneg Q and bQ: bounded-by b Q} \)
Non-negative:

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

Bounded:

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

next

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

next

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

by\((\text{auto intro:add-nonneg-nonneg mult-nonneg-nonneg})\)

from \(utQ\) have \(\text{wlp} f Q s \leq 1 \text{wlp} g Q s \leq 1\) by(\text{auto})
with \(nnP\) have \(P s \ast \text{wlp} f Q s + (1 - P s) \ast \text{wlp} g Q s \leq P s \ast 1 + (1 - P s)\) * \(1\)
by(\text{blast intro:add-mono mult-left-mono})
thus \(P s \ast \text{wlp} f Q s + (1 - P s) \ast \text{wlp} g Q s \leq 1\) by(\text{simp})

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

qed

lemma \text{healthy-wp-DC}:
fixes \(f\):’s prog
assumes \(hf\): healthy \((wp f)\) and \(hg\): healthy \((wp g)\)
shows healthy \((wp (f \sqcap g))\)
proof
(intro healthy-parts bounded-byI \text{nnegI le-funI}, \text{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 \(hf\) have bounded-by \(b\) \((wp f)\) 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 \(\text{assms have} mf\): mono-trans \((wp f)\) and \(mg\): mono-trans \((wp g)\) by(\text{auto})
assume \(sP\): sound \(P\) and \(sQ\): sound \(Q\) and \(le\): \(P \triangleright Q\)
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}[\text{OF mono-transD}[\text{OF mf}]], \text{le-funD}[\text{OF mono-transD}[\text{OF mg}]]\)
thus \(\text{min} (wp f P s) (wp g P s) \leq \text{min} (wp f Q s) (wp g Q s)\) by(\text{auto})

next
fix \(P\):’s \(\Rightarrow\) \text{real} and \(c\): real and \(s\):’s
assume \(sP\): sound \(P\) and \(pos\): \(0 \leq c\)
from assms have \(sf\): scaling \((wp f)\) and \(sg\): scaling \((wp g)\) by\((auto)\)

from pos have \(c \times \min (wp f P s) (wp g P s) = \min (c \times wp f P s) (c \times wp g P s)\)

by\((simp add:min-distrib)\)

also from \(sP\) and pos have ...

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

qed

lemma nearly-healthy-wlp-DC:

fixes \(f\)::\('s prog\)

assumes \(h:\): nearly-healthy \((wp f)\)

and \(h:\): nearly-healthy \((wp g)\)

shows 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 \(nP\): \(nneg P\) and \(nP\): \(bounded-by b P\)

with \(h:\) \(h:\) have \(nP\): \(unitary P\)

thus \(0 \leq wp f P s \leq wp g P s\) by\((auto)\)

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

also from \(nP\) have ...

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

fix \(Q::\:'s\Rightarrow real\)

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

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

also from \(nP\) \(nP\) le have ...

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

qed

lemma healthy-wp-AC:

fixes \(f::\:'s prog\)

assumes \(h:\): healthy \((wp f)\) and \(h:\): healthy \((wp g)\)

shows healthy \((wp f \sqcap g)\)

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

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

assume \(nP\): \(nneg P\) and \(nP\): \(bounded-by b P\)

with \(h:\) have \(bounded-by b (wp f P)\) by\((auto)\)
4.2. HEALTHINESS

\[ \text{hence } wp \ f \ P \ s \leq b \ \text{by}(\text{blast}) \]

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

ultimately show \( \max \ (wp \ f \ P \ s) \ (wp \ g \ P \ s) \leq b \) \ by\((\text{auto})\)

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

next
fix \( P::'s \Rightarrow \text{real} \) and \( Q \) and \( s::'s \)
from \( \text{assms} \) have \( mf: \text{mono-trans} \ (wp \ f) \) and \( mg: \text{mono-trans} \ (wp \ g) \) \ by\((\text{auto})\)
assume \( sp: \text{sound} \ P \) and \( sq: \text{sound} \ Q \) and \( le: P \vdash Q \)

hence \( wp \ f \ P \ s \leq wp \ f \ Q \ s \) and \( wp \ g \ P \ s \leq wp \ g \ Q \ s \)
\ by\((\text{auto intro:le-funD[OF mono-transD, OF mf]} \le-funD[OF mono-transD, OF mg])\)
thus \( \max \ (wp \ f \ P \ s) \ (wp \ g \ P \ s) \leq \max \ (wp \ f \ Q \ s) \ (wp \ g \ Q \ s) \) \ by\((\text{auto})\)

next
fix \( P::'s \Rightarrow \text{real} \) and \( c::\text{real} \) 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\((\text{auto})\)
from \( \text{pos} \) have \( c \ast \max \ (wp \ f \ P \ s) \ (wp \ g \ P \ s) = \max \ (c \ast wp \ f \ P \ s) \ (c \ast wp \ g \ P \ s) \)
\ by\((\text{simp add:max-distrib})\)
also from \( sf\) and \( pos\)
have \( = \max \ (wp \ f \ (\lambda s. c \ast P \ s) \ s) \ (wp \ g \ (\lambda s. c \ast P \ s) \ s) \)
\ by\((\text{simp add:scalingD[OF sf]} \ scalingD[OF sg])\)
finally show \( c \ast \max \ (wp \ f \ P \ s) \ (wp \ g \ P \ s) = \max \ (wp \ f \ (\lambda s. c \ast P \ s) \ s) \ (wp \ g \ (\lambda s. c \ast P \ s) \ s) \).

qed

lemma nearly-healthy-wlp-AC:

fixes \( f::'s \Rightarrow \text{prog} \)
assumes \( hf: \text{nearly-healthy} \ (wp \ f) \)
and \( hg: \text{nearly-healthy} \ (wp \ g) \)
shows \( \text{nearly-healthy} \ (wp \ (f \uplus \ g)) \)
proof\((\text{intro nearly-healthyI bounded-byI nnegI unitaryI2 le-funI, simp-all only:wp-eval})\)
fix \( b \) and \( P::'s \Rightarrow \text{real} \) and \( s::'s \)
assume \( up: \text{unitary} \ P \)
with \( hf \) have \( wp \ f \ P \ s \leq 1 \) \ by\((\text{auto})\)
moreover from \( up \ hg \) have \( \text{unitary} \ (wp \ g \ P) \) \ by\((\text{auto})\)
hence \( wp \ g \ P \ s \leq 1 \) \ by\((\text{auto})\)
ultimately show \( \max \ (wp \ f \ P \ s) \ (wp \ g \ P \ s) \leq 1 \) \ by\((\text{auto})\)

from \( up \ hf \) have \( \text{unitary} \ (wp \ f \ P) \) \ by\((\text{auto})\)
hence \( \theta \leq wp \ f \ P \ s \) \ by\((\text{auto})\)
thus \( 0 \leq \max \ (wp \ f \ P \ s) \ (wp \ g \ P \ s) \) \ by\((\text{auto})\)
next
fix $P : \text{real} \Rightarrow \text{real}$ and $Q$ and $s : \text{unitary}$
assume $uP : \text{unitary} P$ and $uQ : \text{unitary} Q$ and $le : P \vdash Q$
hence $\text{wlp} f P s \leq \text{wlp} f Q s$ and $\text{wlp} g P s \leq \text{wlp} g Q s$
by (auto intro: le-funD[of nearly-healthy-monoD, OF hf]
\hspace{1em} le-funD[of nearly-healthy-monoD, OF hg])
thus $\max (\text{wlp} f P s) (\text{wlp} g P s) \leq \max (\text{wlp} f Q s) (\text{wlp} g Q s)$ by (auto)
qed

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

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

lemma healthy-wp-repeat:
assumes $h-a : \text{healthy} (\text{wp} a)$
shows $\text{healthy} (\text{wp} (\text{repeat} n a))$ (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 $h-a : \text{nearly-healthy} (\text{wlp} a)$
shows $\text{nearly-healthy} (\text{wlp} (\text{repeat} n a))$ (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 $\text{prog} : \Rightarrow \text{real}$ and $S : \Rightarrow \text{set}$
assumes healthy: $\forall x . \forall s . x \in S s \Rightarrow \text{healthy} (\text{wp} (\text{prog} x))$
and nonempty: $\exists x . \forall s . x \in S s$
shows healthy $(\text{wp} (\text{SetDC} \text{prog} S))$ (is healthy ?T)
proof (intro healthy-parts bounded-byI nnegI le-funI, simp-all only: wp-eval)
fix $b$ and $P : \Rightarrow \text{real}$ and $s : \Rightarrow$
assume $bP : \text{bounded-by} b P$ and $nP : \text{nneg} P$
hence $sP : \text{sound} P$ by (auto)

from nonempty obtain $x$ where $x : (\lambda a . \text{wp} (\text{prog} a) P s) \cdot S s$ by (blast)
moreover from $sP$ and healthy
have $\forall x : (\lambda a . \text{wp} (\text{prog} a) P s) \cdot S s . \theta \leq x$ by (auto)
4.2. HEALTHINESS

ultimately have \( \inf ((\lambda a. \wp (\text{prog} a) P s) \cdot S s) \leq x \)
   by(intro cInf-lower bdd-belowI, auto)
also from \( zin \) and \( \text{healthy and sP and bP} \) have \( x \leq b \) by(blast)
finally show \( \inf ((\lambda a. \wp (\text{prog} a) P s) \cdot S s) \leq b \).

from \( zin \) and \( sP \) and \( \text{healthy} \)
show \( 0 \leq \inf ((\lambda a. \wp (\text{prog} a) P s) \cdot S s) \) by(blast intro:cInf-greatest)
next
fix \( P' : a \Rightarrow \text{real and Q and s':a} \)
assume \( sP : \text{sound P and sQ: sound Q and le: P} \vdash Q \)

from \( \text{nonempty} \) obtain \( x \) where \( zin: x \in (\lambda a. \wp (\text{prog} a) P s) \cdot S s \) by(blast)
moreover from \( sP \) and \( \text{healthy} \)
have \( \forall x \in (\lambda a. \wp (\text{prog} a) P s) \cdot S s. 0 \leq x \) by(auto)
moreover
have \( \forall x \in (\lambda a. \wp (\text{prog} a) Q s) \cdot S s. \exists y \in (\lambda a. \wp (\text{prog} a) P s) \cdot S s. y \leq x \)
proof(rule ballI, clarify, rule bexI)
fix \( x \) and \( a \) assume \( a \in S s \)
with \( \text{healthy and sP and sQ and le} \) show \( \wp (\text{prog} a) P s \leq \wp (\text{prog} a) Q s \)
   by(auto dest:mono-transD[OF healthy-monoD])
from \( a \in \text{show} \) \( \wp (\text{prog} a) P s \in (\lambda a. \wp (\text{prog} a) P s) \cdot S s \) by(simp)
qed
ultimately
show \( \inf ((\lambda a. \wp (\text{prog} a) P s) \cdot S s) \leq \inf ((\lambda a. \wp (\text{prog} a) Q s) \cdot S s) \)
   by(intro cInf-mono, blast+)
next
fix \( P' : a \Rightarrow \text{real and c::real and s':a} \)
assume \( sP : \text{sound P and pos: 0} \leq c \)
from \( \text{nonempty} \) obtain \( x \) where \( zin: x \in (\lambda a. \wp (\text{prog} a) P s) \cdot S s \) by(blast)
have \( c \in \inf ((\lambda a. \wp (\text{prog} a) P s) \cdot S s) = \)
   \( \inf ((\ast) c \cdot ((\lambda a. \wp (\text{prog} a) P s) \cdot S s)) (\text{is } ?U = ?V) \)
proof(rule antisym)
show \( ?U \leq ?V \)
proof(rule cInf-greatest)
from \( \text{nonempty} \) show \( (\ast) c \cdot ((\lambda a. \wp (\text{prog} a) P s) \cdot S s) \neq \{\} \) by(auto)
fix \( x \) assume \( x \in (\ast) c \cdot ((\lambda a. \wp (\text{prog} a) P s) \cdot S s) \)
then obtain \( y \) where \( yin: y \in (\lambda a. \wp (\text{prog} a) P s) \cdot S s \) and \( \text{rux: x = c} \)
* \( y \) by(auto)
   have \( \inf ((\lambda a. \wp (\text{prog} a) P s) \cdot S s) \leq y \)
   proof(intro cInf-lower[OF yin] bdd-belowI)
      fix \( z \) assume \( zin: z \in (\lambda a. \wp (\text{prog} a) P s) \cdot S s \)
      then obtain \( a \) where \( a \in S s \) and \( z = \wp (\text{prog} a) P s \) by(auto)
      with \( sP \) show \( 0 \leq z \) by(auto dest:healthy)
   qed
   with pos \( \text{rux} \) show \( c \in \inf ((\lambda a. \wp (\text{prog} a) P s) \cdot S s) \leq x \) by(auto intro:mult-left-mono)
   qed
   show \( ?V \leq ?U \)
   proof(cases)
assume \(cz::c = 0\)
moreover {
from nonempty obtain \(c\) where \(c \in S\) \(s\) by(auto)
  hence \(\exists x. \exists xa \in S. x = \text{wp} (\text{prog} xa)\) \(P s\) by(auto)
}
ultimately show \(?\text{thesis}\) by(simp add:image-def)
next
assume \(c \neq 0\)
from nonempty have \(S s \neq \{\}\) by blast
then have inverse \(c \cdot (\text{INF} x : S. c \cdot \text{wp} (\text{prog} x) P s) \leq (\text{INF} a : S. w p (\text{prog} a) P s)\)
proof (rule cINF-greatest)
  fix \(x\)
  assume \(x \in S\)
  have bdd-below \((\lambda x. c \cdot \text{wp} (\text{prog} x) P s)\) \(S\)
proof (rule bdd-belowI [of - 0])
    fix \(z\)
    assume \(z \in (\lambda x. c \cdot \text{wp} (\text{prog} x) P s)\) \(S\)
    then obtain \(b\) where \(b \in S\) and \(\text{ruw}: z = c \cdot \text{wp} (\text{prog} b) P s\) by auto
    with \(sP\) have \(\text{wp} (\text{prog} b) P s\) by (auto dest: healthy)
    with pos show \(\text{wp} (\text{prog} b) P s\) by (auto simp: ruw intro: mult-nonneg-nonneg)
  qed
then have \((\text{INF} x : S. c \cdot \text{wp} (\text{prog} x) P s) \leq c \cdot \text{wp} (\text{prog} x) P s\)
using \(x \in S\) by (rule cINF-lower)
with \(c \neq 0\) show inverse \(c \cdot (\text{INF} x : S. c \cdot \text{wp} (\text{prog} x) P s) \leq \text{wp} (\text{prog} x) P s\)
  by (simp add: mult-div-mono-left pos)
qed
with \(c \neq 0\) have inverse \(c \cdot ?V \leq inverse c \cdot ?U\)
  by (simp add: mult.assoc [symmetric])
with pos have inverse \(c \cdot ?V \leq inverse c \cdot ?U\)
  by (auto intro:mult-left-mono)
with \(c \neq 0\) show \(?\text{thesis}\) by (simp add:mult.assoc [symmetric])
  qed
qed
also have ... = \(\text{INF} ((\lambda a. c \cdot \text{wp} (\text{prog} a) P s)\) \(S\)
by(simp add:image-comp[symmetric] o-def)
also from \(sP\) and pos have ... = \(\text{INF} ((\lambda a. \text{wp} (\text{prog} a) (\lambda s. c \cdot P s) s)\) \(S\)
by(simp add:scalingD[OF healthy-scalingD, OF healthy] cong:image-cong)
finally show \(c \cdot \text{INF} ((\lambda a. \text{wp} (\text{prog} a) (\lambda s. c \cdot P s) s)\) \(S\) = \(\text{INF} ((\lambda a. \text{wp} (\text{prog} a) (\lambda s. c \cdot P s) s)\) \(S\)\).
  qed

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

\[ \text{fix } b \text{ and } P :: \forall a \Rightarrow \text{real and } s :: a \]
\[ \text{assume } uP : \text{unitary } P \]

\[ \text{from nonempty obtain } x \text{ where } x \in (\lambda a. \text{wp} (\text{prog} a) P) \Leftrightarrow S s \text{ by (blast)} \]
\[ \text{moreover } \}
\[ \text{from } uP \text{ healthy } \]
\[ \text{have } \forall x \in (\lambda a. \text{wp} (\text{prog} a) P) \Leftrightarrow S s \text{. unitary } x \text{ by (auto)} \]
\[ \text{hence } \forall x \in (\lambda a. \text{wp} (\text{prog} a) P) \Leftrightarrow S s. \theta \leq x \text{ by (auto)} \]
\[ \text{hence } \forall y \in (\lambda a. \text{wp} (\text{prog} a) P) \Leftrightarrow S s. \theta \leq y \text{ by (auto)} \]
\[ \text{ultimately have } \text{Inf } ((\lambda a. \text{wp} (\text{prog} a) P) \Leftrightarrow S s) \leq x \text{ by (intro cInf-lower bdd-below1, auto)} \]
\[ \text{also from } x \in \text{healthy } uP \text{ have } x \leq 1 \text{ by (blast)} \]
\[ \text{finally show } \text{Inf } ((\lambda a. \text{wp} (\text{prog} a) P) \Leftrightarrow S s) \leq 1 \]

\[ \text{from } x \in \text{healthy } uP \text{ healthy } \]
\[ \text{show } 0 \leq \text{Inf } ((\lambda a. \text{wp} (\text{prog} a) P) \Leftrightarrow S s) \]
\[ \text{by (blast dest!: unitary-sound \{(OF nearly-healthy-unitaryD[OF - uP]\}) intro: cInf-greatest)} \]

\[ \text{next } \]
\[ \text{fix } P :: \forall a \Rightarrow \text{real and } Q \text{ and } s :: a \]
\[ \text{assume } uP : \text{unitary } P \text{ and } uQ : \text{unitary } Q \text{ and } le : P \Rightarrow Q \]

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

\[ \text{lemma healthy-wp-SetPC:} \]
\[ \text{fixes } p :: s \Rightarrow 'a \Rightarrow \text{real} \]
\[ \text{and } f :: 'a \Rightarrow 's \text{ prog} \]
\[ \text{assumes healthy: } \lambda a. s. a \in \text{supp} (p s) \Rightarrow \text{healthy } (\text{wp} (f a)) \]
\[ \text{and sound: } \lambda s. \text{sound} (p s) \]
\[ \text{and sub-dist: } \lambda s. (\sum a \in \text{supp} (p s). p s a) \leq 1 \]
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 ⇒ 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 \( \forall a. a \in \text{supp}(p \ s) \Rightarrow \text{wp}(f \ a) P \ s \leq b \)
  by(blast dest:healthy-bounded-byD)
  with sound have \( (\sum a \in \text{supp}(p \ s). p \ s \ a \ a \ \ast \ \text{wp}(f \ a) P \ s) \leq (\sum a \in \text{supp}(p \ s). p \ s \ a \ a \ \ast \ b) \)
  by(blast intro:sum-mono mult-left-mono)
  also have \( \ldots = (\sum a \in \text{supp}(p \ s). p \ s \ a \ a \ \ast \ b) \)
  by(simp add:sum-distrib-right)
  also { 
    from bP and nP have 0 ≤ b by(blast)
    with sub-dist have \( (\sum a \in \text{supp}(p \ s). p \ s \ a \ a \ \ast \ b) \leq 1 \ast b \)
    by(rule mult-right-mono)
  }
  also have 1 * b = b by(simp)
  finally show \( (\sum a \in \text{supp}(p \ s). p \ s \ a \ a \ \ast \ \text{wp}(f \ a) P \ s) \leq b \).

  show 0 ≤ (\sum a \in \text{supp}(p \ s). p \ s \ a \ a \ \ast \ \text{wp}(f \ a) P \ s)
  proof (rule sum-nonneg [OF mult-nonneg-nonneg])
    fix x
    from sound show 0 ≤ p s x by(blast)
    assume x ∈ \text{supp}(p \ s) with sP and healthy
    show 0 ≤ wp(f x) P s by(blast)
  qed
next
  fix P::'s ⇒ real and Q::'s ⇒ real and s
  assume s-P: sound P and s-Q: sound Q and ent: P ⊢ Q
  with healthy have \( \forall a. a \in \text{supp}(p \ s) \Rightarrow \text{wp}(f \ a) P \ s \leq \text{wp}(f \ a) Q \ s \)
  by(blast)
  with sound show \( (\sum a \in \text{supp}(p \ s). p \ s \ a \ a \ \ast \ \text{wp}(f \ a) P \ s) \leq (\sum a \in \text{supp}(p \ s). p \ s \ a \ a \ \ast \ \text{wp}(f \ a) Q \ s) \)
  by(blast intro:sum-mono mult-left-mono)
next
  fix P::'s ⇒ real and c::real and s::'s
  assume sound: sound P and pos: 0 ≤ c
  have c • (\sum a \in \text{supp}(p \ s). p \ s \ a \ a \ \ast \ \text{wp}(f \ a) P \ s) =
  (\sum a \in \text{supp}(p \ s). p \ s \ a \ a \ (c • \text{wp}(f \ a) P \ s))
  (is \ ?A = ?B)
  by(simp add:sum-distrib-left ac-simps)
  also from sound and pos and healthy
  have \( \ldots = (\sum a \in \text{supp}(p \ s). p \ s \ a \ a \ \ast \ \text{wp}(f \ a) (\lambda s. c \ a P s) \)
  by(auto simp:scalingD[OF healthy-scalingD])
  finally show ?A = \ldots .
qed
lemma nearly-healthy-wlp-SetPC:
  fixes v::'s ⇒ 'a ⇒ real
  and f::'a ⇒ 's prog
  assumes healthy: ∀ a s. a ∈ supp (p s) ⇒ nearly-healthy (wp (f a))
  and sound: ∀ s. sound (p s)
  and sub-dist: ∀ s. (∑ a∈supp (p s). p s a) ≤ 1
  shows nearly-healthy (wp (SetPC f p)) (is nearly-healthy ?X)
proof
  intro nearly-healthy1 unitaryI2 bounded-byI nnegI le-funI, simp-all:wp-eval
  finally show "s expect real and s::'s"
  assume uP: unitary P
  from uP healthy have ∀ a. a ∈ supp (p s) ⇒ unitary (wp (f a) P) by(auto)
  hence ∀ a. a ∈ supp (p s) ⇒ wp (f a) P s ≤ 1 by(auto)
  with sound have (∑ a∈supp (p s). p s a * wp (f a) P s) ≤ (∑ a∈supp (p s). p s a * 1)
    by(blast intro:sum-mono mult-left mono)
  also have "= (∑ a∈supp (p s). p s a)
    by(simp add:sum-distrib-right)
  also note sub-dist
finally show (∑ a∈supp (p s). p s a * wp (f a) P s) ≤ 1 .
show 0 ≤ (∑ a∈supp (p s). p s a * wp (f a) P s)
proof
  rule sum-nonneg [OF mult-nonneg nonneg]
  fix x
  from sound show 0 ≤ p s x by(blast)
  assume x ∈ supp (p s) with uP healthy
  show 0 ≤ wp (f x) P s by(blast)
qed
next
fix P::'s expect and Q::'s expect and s
assume uP: unitary P and uQ: unitary Q and lec: P ⊢ Q
hence ∀ a. a ∈ supp (p s) ⇒ wp (f a) P s ≤ wp (f a) Q s
  by(blast intro:le-funD[OF nearly-healthy monoD, OF healthy])
with sound show (∑ a∈supp (p s). p s a * wp (f a) P s) ≤
  (∑ a∈supp (p s). p s a * wp (f a) Q s)
    by(blast intro: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-healthy1 unitaryI2 nnegI bounded-byI, auto simp:o-def wp-eval)

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

lemma nearly-healthy-wlp-Bind:
fixes f::'s ⇒ 'a
assumes hsub: ∀s. nearly-healthy (wp (p (f s)))
shows nearly-healthy (wp (Bind f p))
proof(intro nearly-healthyI unitaryI2 nnegI bounded-byI le-funI, simp-all only:wp-eval)
fix P::'s expect and s::'s
assume unitary P
with hsub have unitary (wp (p (f s)) P) by(auto)
thus 0 ≤ wp (p (f s)) P s wp (p (f s)) Q s ≤ 1 by(auto)
fix Q::'s expect
assume unitary Q P ⊢ Q
with uP show wp (p (f s)) P s ≤ wp (p (f s)) Q s
  by(blast intro:le-funD[OF nearly-healthy-monoD, OF hsub])
qed

4.2.2 Healthiness for Loops

lemma wp-loop-step-mono:
fixes t u::'s trans
assumes hh: healthy (wp body)
  and le: le-trans t u
  and ht: ∀P. sound P ⇒ sound (t P)
  and hu: ∀P. sound P ⇒ sound (u P)
shows le-trans (wp (body :: Embed t « G » ⊕ Skip))
  (wp (body :: Embed u « G » ⊕ Skip))
proof(intro le-transI le-funI, simp add:wp-eval)
fix P::'s expect and s::'s
assume sP: sound P
with le have t P ⊢ u P by(auto)
moreover from sP ht hu have sound (t P) sound (u P) by(auto)
ultimately have \( \text{wp body} (t \ P) \ s \leq \text{wp body} (u \ P) \ s \)
by(auto intro:le-funD[OF mono-transD, OF healthy-monoD, OF hh])
thus \( «G» \ s * \text{wp body} (t \ P) \ s \leq «G» \ s * \text{wp body} (u \ P) \ s \)
by(auto intro:mult-left-mono)
qed

lemma \( \text{lfp-loop-step-mono} \):
fixes \( t \ u ::'s \ trans \)
assumes \( \text{mb} : \text{nearly-healthy} (\text{wp body}) \)
and \( \text{ht} : \bigwedge \!P. \ \text{unitary} \ (t \ P) \implies \text{unitary} \ (t \ P) \)
and \( \text{hu} : \bigwedge \!P. \ \text{unitary} \ (u \ P) \implies \text{unitary} \ (u \ P) \)
shows \( \text{le-utrans} (\text{wp body} ;; \text{Embed} \ t \ «G» \oplus \text{Skip}) \)
(\text{wp body} ;; \text{Embed} \ u \ «G» \oplus \text{Skip})
proof(intro le-utransI le-funI, simp add:wp-eval)
fix \( P ::'s \ expect \) and \( s ::'s \)
assume \( uP : \text{unitary} \ P \)
with \( \text{le} \) have \( t \ P \vdash u \ P \) by(auto)
moreover from \( uP \ \text{ht} \ \text{hu} \) have \( \text{unitary} \ (t \ P) \implies \text{unitary} \ (u \ P) \)
ultimately have \( \text{wp body} (t \ P) \ s \leq \text{wp body} (u \ P) \ s \)
by(rule le-funD[OF nearly-healthy-monoD[OF mb]])
thus \( «G» \ s * \text{wp body} (t \ P) \ s \leq «G» \ s * \text{wp body} (u \ P) \ s \)
by(auto intro:mult-left-mono)
qed

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

lemma \( \text{lfp-loop-fp} \):
assumes \( \text{hb} : \text{healthy} (\text{wp body}) \)
and \( \text{sP} : \text{sound} \ P \)
shows \( \lambda \ s. \ «G» \ s * \text{wp body} (\lambda \ s. \text{bound-of} \ P) \ s + \langle N \ G \rangle \ s * P \ s \vdash \lambda \ s. \text{bound-of} \ P \)
proof(rule le-funI)
fix \( s \)
from \( sP \) have \( \text{sound} \ (\lambda \ s. \text{bound-of} \ P) \) by(auto)
moreover hence bounded-by (bound-of P) (\lambda \ s. bound-of P) by(auto)
ultimately have bounded-by (bound-of P) (\text{wp body} (\lambda \ s. bound-of P))
using \( \text{hb} \) by(auto)

hence \( \text{wp body} (\lambda \ s. \text{bound-of} \ P) \ s \leq \text{bound-of} \ P \) by(auto)

moreover from \( sP \) have \( P \ s \leq \text{bound-of} \ P \) by(auto)
ultimately have \( «G» \ s * \text{wp body} (\lambda a. \text{bound-of} \ P) \ s + (1 - «G» \ s) * P \ s \leq 
«G» \ s * \text{bound-of} \ P + (1 - «G» \ s) * \text{bound-of} \ P \)
by(blast intro:add-mono mult-left-mono)
thus \( «G» \ s * \text{wp body} (\lambda a. \text{bound-of} \ P) \ s + \langle N \ G \rangle \ s * P \ s \leq \text{bound-of} \ P \)
by(simp add:algebra-simps negate-embed)
qed

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

**lemma** \text{lfp-loop-sound}:

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

**lemma** \text{wlp-loop-step-unitary}:

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

**proof**(intro unitarityI2 nnegI bounded-byI, simp-all add:wp-eval)
fix $s$::'

from $ht uP$ have $utP$: unitary $(t P)$ by(auto)
with $hb$ have unitary ($\text{wlp body} (t P)$) by(auto)

hence $0 \leq \text{wlp body} (t P) s$ by(auto)

with $uP$ show $0 \leq \langle G \rangle s * \text{wlp body} (t P) s + (1 - \langle G \rangle s) * P s$
by(auto intro!:\text{add-nonneg-nonneg mult-nonneg-nonneg})

from $ht uP$ have bounded-by 1 $(t P)$ by(auto)
with $utP hb$ have bounded-by 1 ($\text{wlp body} (t P)$) by(auto)

hence $\text{wlp body} (t P) s \leq 1$ by(auto)

with $uP$ have $\langle G \rangle s * \text{wlp body} (t P) s + (1 - \langle G \rangle s) * P s \leq \langle G \rangle s * 1 + (1 - \langle G \rangle s) * 1$
by(blast intro!:\text{add-mono mult-left-mono})
also have $... = 1$ by(simp)
finally show $\langle G \rangle s * \text{wlp body} (t P) s + (1 - \langle G \rangle s) * P s \leq 1$ .

qed

**lemma** \text{wp-loop-step-sound}:

fixes $t$: u::'s trans
assumes $hb$: healthy ($\text{wp body}$)
and $ht$: $\forall P. \text{sound} P \implies \text{sound} (t P)$
and $sP$: sound $P$

shows sound ($\text{wp body} (t P)$; Embed $t \langle G \rangle s \oplus \text{Skip} P$)

**proof**(intro soundI2 nnegI bounded-byI, simp-all add:wp-eval)
fix $s$::'

from $ht sP$ have $stP$: sound $(t P)$ by(auto)
with $hb$ have $0 \leq \text{wp body} (t P) s$ by(auto)

with $sP$ show $0 \leq \langle G \rangle s * \text{wp body} (t P) s + (1 - \langle G \rangle s) * P s$
by(auto intro!: add-nonneg-nonneg mult-nonneg-nonneg)

from htsP 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 bb by(auto)
hence wp body (t P) s ≤ max (bound-of P) (bound-of (t P)) by(auto)
moreover {
  from sP have P s ≤ bound-of P by(auto)
hence P s ≤ max (bound-of P) (bound-of (t P)) by(auto)
}
ultimately have «G» s * wp body (t P) s + (1 - «G» s) * P s ≤
  «G» s * max (bound-of P) (bound-of (t P)) +
  (1 - «G» s) * max (bound-of P) (bound-of (t P))
  by(blast intro:add-mono mult-left-mono)
also have ... = max (bound-of P) (bound-of (t P)) by(simp add: algebra-simps)
finally show «G» s * wp body (t P) s + (1 - «G» s) * P s ≤
  max (bound-of P) (bound-of (t P))
qed

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

lemma wlp-Loop1:
  fixes body :: 's prog
  assumes unitary: unitary P
  and healthy: nearly-healthy (wlp body)
  shows wlp (do G -> body od) P =
    gfp-exp (λQ s. «G» s * wlp body Q s + «N G» s * P s)
  (is ?X = gfp-exp (?Y P))
proof —
  let ?Z u = (body ;; Embed u « G ⊕ Skip)
  show ?thesis
proof(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 (wlp body Q) by(auto)
hence 0 ≤ wlp body Q s by(auto)
with uP show \(0 \leq \langle G \rangle s \ast \text{wlp body } Q s + \langle N \rangle G s \ast P s\)
by(auto intro: add-nonneg-nonneg mult-nonneg-nonneg)

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

next
fix P Q R :: s expect and s :: s
assume uP: unitary P and uQ: unitary Q and uR: unitary R
and le: \(Q \vdash R\)

hence wlp body Q s \leq wlp body R s
by(blast intro: le-funD[OF nearly-healthy-monoD, OF healthy])

thus \(\langle G \rangle s \ast \text{wlp body } Q s + \langle N \rangle G s \ast P s \leq
\langle G \rangle s \ast \text{wlp body } R s + \langle N \rangle G s \ast P s\)
by(auto intro: mult-left-mono)

next
fix t u :: s trans
assume le-utrans t u
\[
\forall P. \text{unitary } P \implies \text{unitary } (t P)
\]
\[
\forall P. \text{unitary } P \implies \text{unitary } (u P)
\]

thus le-utrans (wlp (?Z t)) (wlp (?Z u))
by(blast intro!: wlp-loop-step-mono[OF healthy])

qed
definition wlp-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 ;; Skip)) ?v
by(intro le-transI, simp add: wp-eval lfp-loop-fp[unfolded negate-embed] hb)

show \(\forall P. \text{sound } P \implies \text{sound } (?v P)\)
by(auto)

qed

Likewise, we can rewrite strict loops.

lemma wp-Loop1:
fixes body :: s prog
assumes sP: sound P
and healthy: healthy (wp body)
shows wp (do G \rightarrow body od) P =
lfp-exp (\(\lambda Q s. \langle G \rangle s \ast \text{wlp body } Q s + \langle N \rangle G s \ast P s\))
(is ?X = lfp-exp (?Y P))
4.2. HEALTHINESS

proof

let \(?Z u = (\text{body} ; Embed u \uplus G \uplus \text{Skip})\)

show \(?\)thesis

proof (simp only: wp-eval, intro lfp-pulldown assms le-funI sP mono-trans)

fix \(u P\)

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

next

fix \(t::'s\ trans\ and\ P::'s\ expect\)

assume at: \(\forall Q.\ \text{sound} \ Q \implies \text{sound} \ (t\ Q)\)\ and \(uP: \text{sound} \ P\)

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

next

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

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

show \(\text{sound} \ (\lambda a. \«G» a \ast wp \text{body} \ Q a + \«N G» a \ast P a)\)

proof (intro soundI2 nnegI bounded-byI)

fix \(s::'s\)

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

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

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

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

moreover {
  from sP have \(\text{P} \ s \leq \text{bound-of} \ P\)\ \(\text{by}(auto)\)
  hence \(\text{P} \ s \leq \text{max} \ (\text{bound-of} \ P)\ (\text{bound-of} \ Q)\ \text{by}(auto)\)
}

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

\(\«N G» \ s \ast \text{max} \ (\text{bound-of} \ P)\ (\text{bound-of} \ Q)\)

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

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

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

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

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

ultimately show \(0 \leq \«G» \ s \ast \text{wp} \text{body} \ Q \ s + \«N G» \ s \ast P \ s\)

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

qed

next

fix \(P Q R::'s\ expect\)\ and \(s::'s\)

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

and le: \(Q \vdash R\)

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

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

thus \(\«G» \ s \ast \text{wp} \text{body} \ Q \ s + \«N G» \ s \ast P \ s \leq\)

\(\«G» \ s \ast \text{wp} \text{body} \ R \ s + \«N G» \ s \ast P \ s\)

by (auto intro: mult-left-mono)

next

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

next
from $\text{healthy}$ show le-trans $(wp\ (\lambda Z\ (\lambda P\ s.\ \text{bound-of } P)))\ (\lambda P\ s.\ \text{bound-of } P)$
by (intro le-transI, simp add: $\text{wp-eval}\ \text{lfp-loop-fp}[\text{unfolded}\ \text{negate-embed}]$)

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

lemma nearly-healthy-wlp-loop:
fixes body::$\varepsilon\ s$ prog
assumes hb: nearly-healthy $(wlp\ body)$
shows nearly-healthy $(wlp\ (do\ G\ \rightarrow\ body\ od))$
proof (intro nearly-healthyI unitaryI2 nnegI2 bounded-byI2,
  simp-all add: $\text{wlp-Loop1}$)

fix $P::\varepsilon\ s$ expect
assume $uP$: unitary $P$
let $?X\ R = \lambda Q\ s.\ \langle G\rangle\ s * wlp\ body\ Q\ s + \langle N\ G\rangle\ s * R\ s$

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

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

fix $Q::\varepsilon\ s$ expect
assume $uQ$: unitary $Q$ and $lc$: $P \vdash Q$

show $\text{gfp-exp}\ (?X\ P) \vdash \text{gfp-exp}\ (?X\ Q)$
proof (rule gfp-exp-least)
  fix $R::\varepsilon\ s$ expect assume $uR$: unitary $R$
  assume $fp$: $R \vdash ?X\ P\ R$
  also from $lc$ have $\vdash \langle \text{le}\_\text{funI}\rangle$
    by (blast intro!: add-mono mult-left-mono le-funI)
4.2. HEALTHINESS

finally show $R \vdash \text{gfp-exp} \ (\forall X \ Q)$
using $uR$ by(auto intro:gfp-exp-upperbound)

next
show unitary $(\text{gfp-exp} \ (\forall X \ Q))$
proof(rule gfp-exp-unitary, intro unitaryI2 nnegI bounded-byI)
fix $R$'s expect and $s$'s assume $uR$: unitary $R$
with $hb$ have $ubP$: unitary $(\text{wlp body} \ R)$ by(auto)
with $uQ$ show $0 \leq \langle \ G \rangle \ s * \text{wlp body} \ R \ s + \langle \ N \ G \rangle \ s * Q \ s$
by(blast intro:add-nonneg-nonneg mult-nonneg-nonneg)

from $ubP$ $uQ$ have $\text{wlp body} \ R \ s \leq 1 \ Q \ s \leq 1$ by(auto)

hence $\langle \ G \rangle \ s * \text{wlp body} \ R \ s + \langle \ N \ G \rangle \ s * Q \ s \leq \langle \ G \rangle \ s * 1 + \langle \ N \ G \rangle \ s$

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

thus $\langle \ G \rangle \ s * \text{wlp body} \ R \ s + \langle \ N \ G \rangle \ s * Q \ s \leq 1$
by(simp add:negate-embed)

qed

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

lemma healthy-wp-loop:
fixes body::'s prog
assumes $hb$: healthy $(\text{wp body})$
shows healthy $(\text{wp (do } G \longrightarrow \text{ body od)})$

proof –
let $?X \ P = (\lambda Q \ s. \langle \ G \rangle \ s * \text{wp body} \ Q \ s + \langle \ N \ G \rangle \ s * P \ s)$
show $?\text{thesis}$
proof(intro healthy-parts bounded-byI2 nnegI2, simp-all add:wp-Loop1 $hb$ soundI2 sound-intros)
fix $P$'s expect and $c::\text{real}$ and $s$'s
assume $sp$: sound $P$ and $nnc$: $0 \leq c$
show $c * (\text{lfp-exp} \ (\forall X \ P)) \ s = \text{lfp-exp} \ (\forall X \ (\lambda s. \ c * P \ s)) \ s$

proof(cases)
assume $c = 0$ thus $?\text{thesis}$
proof(simp, intro antisym)
from $hb$ have $fp$: $\lambda s. \langle \ G \rangle \ s * \text{wp body} \ (\lambda -. \ 0) \ s \vdash \lambda s. \ 0$ by(simp)
hence $\text{lfp-exp} \ (\lambda P \ s. \langle \ G \rangle \ s * \text{wp body} \ P \ s) \vdash \lambda s. \ 0$

by(auto intro:lfp-exp-lowerbound)
thus $\text{lfp-exp} \ (\lambda P \ s. \langle \ G \rangle \ s * \text{wp body} \ P \ s) \ s \leq 0$ by(auto)
have $\lambda s. \ 0 \vdash \text{lfp-exp} \ (\lambda P \ s. \langle \ G \rangle \ s * \text{wp body} \ P \ s)$

by(auto intro:lfp-exp-greatest fp)
thus $0 \leq \text{lfp-exp} \ (\lambda P \ s. \langle \ G \rangle \ s * \text{wp body} \ P \ s) \ s$ by(auto)

qed

next
have onesided: $\bigwedge P \ c. \ c \neq 0 \implies 0 \leq c \implies \text{sound} \ P \implies$
\begin{align*}
& \lambda a. \ c * \text{lfp-exp} \ (\lambda a \ b. \langle \ G \rangle \ b * \text{wp body} \ a \ b + \langle \ N \ G \rangle \ b * P \ b) \ a \\
& \quad \vdash \text{lfp-exp} \ (\lambda a \ b. \langle \ G \rangle \ b * \text{wp body} \ a \ b + \langle \ N \ G \rangle \ b * (c * P \ b))
\end{align*}
proof
  -
  fix P::'s expect and c::real
  assume cnz: c ≠ 0 and nnc: 0 ≤ c and sP: sound P
  with nnc have cnps: 0 < c by(auto)
  hence nnic: 0 ≤ inverse c by(auto)
  show λa. c * lfp-exp (λa b. «G» b * wp body a b + «N G» b * P b) a ⊢
     lfp-exp (λa b. «G» b * wp body a b + «N G» b * (c * P b))
  proof (rule lfp-exp-greatest)
    fix Q::'s expect
    assume sQ: sound Q
    and fp: λb. «G» b * wp body Q b + «N G» b * (c * P b) ⊢ Q
    hence ∃s. «G» s * wp body Q s + «N G» s * (c * P s) ≤ Q s by(auto)
    with nnic
    have ∃s. inverse c * («G» s * wp body Q s + «N G» s * (c * P s)) ≤
      inverse c * Q s
      by(auto intro:mult-left-mono)
    hence ∃s. «G» s * (inverse c * wp body Q s) + (inverse c * c) * «N G»
      s * P s ≤
      inverse c * Q s
      by(simp add: algebra-simps)
    hence ∃s. «G» s * wp body (λs. inverse c * Q s) s + «N G» s * P s ≤
      inverse c * Q s
      by(simp add:cnz scalingD[OF healthy-scalingD, OF hb sQ nnic])
    hence λs. «G» s * wp body (λs. inverse c * Q s) s + «N G» s * P s ⊢
      λs. inverse c * Q s by(rule le-funI)
    moreover from nnic sQ have sound (λs. inverse c * Q s)
      by(auto intro:sound-intros)
    ultimately have lfp-exp (λa b. «G» b * wp body a b + «N G» b * P b) ⊢
      λs. inverse c * Q s
      by(rule lfp-exp-lowerbound)
    hence ∃s. lfp-exp (λa b. «G» b * wp body a b + «N G» b * P b) s ≤
      inverse c * Q s
      by(rule le-funD)
    with nnc
    have ∃s. c * lfp-exp (λa b. «G» b * wp body a b + «N G» b * P b) s ≤
      c * (inverse c * Q s)
      by(auto intro:mult-left-mono)
    also from cnz have ∃s. ... s = Q s by(simp)
    finally show λa. c * lfp-exp (λa b. «G» b * wp body a b + «N G» b * P
      b) a ⊢ Q
      by(rule le-funI)
    next
    from sP have sound (λs. bound-of P) by(auto)
    with hb sP have sound (lfp-exp (?X P))
      by(blast intro:lfp-exp-sound lfp-loop-fp)
    with nnc show sound (λs. c * lfp-exp (?X P) s)
      by(auto intro!:sound-intros)
    from hb sP nnc
\textbf{4.2. HEALTHINESS}

\begin{verbatim}
show \lambda s\cdot \langle G \rangle s * wp\ body\ (\lambda s\ \text{bound-of}\ \langle \lambda s\ .\ c\ *\ P\ s \rangle)\ s + 
\langle N \rangle G\ s * (c\ *\ P\ s) \vdash \lambda s\ \text{bound-of}\ \langle \lambda s\ .\ c\ *\ P\ s \rangle 
\text{by}(\text{iprover intro:lfp-loop-fp sound-intros})

from sP nnc show sound\ (\lambda s\ .\ \text{bound-of}\ \langle \lambda s\ .\ c\ *\ P\ s \rangle) 
\text{by}(\text{auto intro!:sound-intros})
qed

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

from nzc have nzc: inverse c \neq 0 \text{ by(auto)}
moreover with nnc have nnic: \ 0 \leq\ inverse c \text{ by(auto)}

moreover from nnc sP have scP: sound\ (\lambda s\ .\ c\ *\ P\ s) \text{ by(auto intro!:sound-intros})
ultimately have \ ?X (\lambda s\ .\ c\ *\ P\ s) (inverse c) \vdash \ ?Y (\lambda s\ .\ c\ *\ P\ s) (inverse c)

\text{by}(\text{rule onesided})
with nnc have \lambda s\ .\ c\ *\ ?X (\lambda s\ .\ c\ *\ P\ s) (inverse c) s \vdash 
\lambda s\ .\ c\ *\ ?Y (\lambda s\ .\ c\ *\ P\ s) (inverse c) s
\text{by}(\text{blast intro:mult-left-mono})

with nzc show \ ?Y P c \vdash \ ?X P c \text{ by(simp add:mult.assoc[symmetric])}
qed

next
fix P::'s expect and b::real
assume bp: bounded-by b P and np: nneg P
show lp-exp (\lambda Q s\ .\ \langle G \rangle s * wp\ body\ Q\ s + \langle N \rangle G\ s * P\ s) \vdash \lambda s\ b
proof(intro lfp-exp-lowerbound le-funI)
fix s::'s
from bp np hb have bounded-by b (wp\ body\ (\lambda s\ .\ b)) \text{ by(auto)}

hence wp\ body\ (\lambda s\ .\ b) s \leq b \text{ by(auto)}
moreover from bp have P s \leq b \text{ by(auto)}
ultimately have \langle G \rangle s * wp\ body\ (\lambda s\ .\ b) s + \langle N \rangle G\ s * P\ s \leq \langle G \rangle s * b

+ \langle N \rangle G\ s * b

\text{by(auto intro!:add-mono mult-left-mono)}
also have ... = b \text{ by(simp add: negate-embed field-simps)}
finally show \langle G \rangle s * wp\ body\ (\lambda s\ .\ b) s + \langle N \rangle G\ s * P\ s \leq b .
from bp np have 0 \leq b \text{ by(auto)}
\text{thus sound (\lambda s\ .\ b) by(auto)}
qed
from bb bp np show \lambda s\ .\ 0 \vdash lfp-exp (\lambda Q s\ .\ \langle G \rangle s * wp\ body\ Q\ s + \langle N \rangle G\ s * P\ s)
\text{ by(auto dest!:sound-nneg intro!:lfp-loop-greatest})

next
fix P Q::'s expect
\end{verbatim}
assume \( sP \): sound \( P \) and \( sQ \): sound \( Q \) and \( \text{le}: P \vdash Q \)
show \( \text{lfp-exp} \ (\exists X \ P) \vdash \text{lfp-exp} \ (\exists X \ Q) \)
proof (rule \text{lfp-exp-greatest})
fix \( R \)’s expect
assume \( sR \): sound \( R \)
and \( \text{fp}: \lambda s. \ «G» s \ast \text{wp body} R s + «\mathcal{N} G» s \ast Q s \vdash R \)
from \( \text{le} \) have \( \lambda s. «G» s \ast \text{wp body} R s + «\mathcal{N} G» s \ast P s \vdash \)
\( \lambda s. «G» s \ast \text{wp body} R s + «\mathcal{N} G» s \ast Q s \)
by (auto intro:le-funI add-left-mono mult-left-mono)
also note \( \text{fp} \)
finally show \( \text{lfp-exp} \ (\lambda R \ s. «G» s \ast \text{wp body} R s + «\mathcal{N} G» s \ast P s) \vdash R \)
using \( sR \) by (auto intro:lfp-exp-lowerbound)
next
from \( \text{hb} \) \( sP \) show sound (\( \text{lfp-exp} \ (\lambda R s . «G» s \ast \text{wp body} R s + «\mathcal{N} G» s \ast P s) \))
by (rule \text{lfp-loop-sound})
from \( \text{hb} \) \( sQ \) show \( \lambda s. «G» s \ast \text{wp body} (\lambda s. \text{bound-of} Q) s + «\mathcal{N} G» s \ast Q \)
\( s \vdash \lambda s. \text{bound-of} Q \)
by (rule \text{lfp-loop-fp})
from \( sQ \) show 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-contitions for primitive programs automatically.

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

end

4.3 Continuity

theory Continuity imports Healthiness begin

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

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

**Definition** bd-cts :: 's trans ⇒ bool

**Where**

\[ bd-cts \, t = (\forall M. (\forall i. (M \, i \supset M \, (Suc \, i)) \land sound \, (M \, i)) \implies \\
    (\exists b. \forall i. bounded-by \, b \, (M \, i)) \implies \\
    t \, (Sup-exp \, (range \, M)) = Sup-exp \, (range \, (t \, o \, M))) \]

**Lemma** bd-ctsD:

\[
[ bd-cts \, t; (\forall i. (M \, i \supset M \, (Suc \, i)); \, \land \, sound \, (M \, i); \, \land \, bounded-by \, b \, (M \, i)] \implies \\
 t \, (Sup-exp \, (range \, M)) = Sup-exp \, (range \, (t \, o \, M))
\]

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

**Lemma** bd-ctsI:

\[
(\exists b. \forall i. (M \, i \supset (\forall \, (M \, i \supset (\forall \, bounded-by \, b \, (M \, i)))) \implies \\
 t \, (Sup-exp \, (range \, M)) = Sup-exp \, (range \, (t \, o \, M))) = \\
 bd-cts \, t
\]

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

A generalised property for transformers of transformers.

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

**Where**

\[ bd-cts-tr \, T = (\forall M. (\forall i. le-trans \, (M \, i) \, (M \, (Suc \, i)) \land feasible \, (M \, i)) \implies \\
 equiv-trans \, (T \, (Sup-trans \, (M \cdot UNIV))) \, (Sup-trans \, ((T \, o \, M) \cdot UNIV))) \]

**Lemma** bd-cts-trD:

\[
[ bd-cts-tr \, T; (\forall i. le-trans \, (M \, i) \, (M \, (Suc \, i)); \, \land \, feasible \, (M \, i)] \implies \\
 equiv-trans \, (T \, (Sup-trans \, (M \cdot UNIV))) \, (Sup-trans \, ((T \, o \, M) \cdot UNIV))
\]

**By** (simp add: bd-cts-tr-def)

**Lemma** bd-cts-trI:

\[
(\forall M. (\forall i. le-trans \, (M \, i) \, (M \, (Suc \, i))) \implies (\forall i. feasible \, (M \, i)) \implies \\
 equiv-trans \, (T \, (Sup-trans \, (M \cdot UNIV))) \, (Sup-trans \, ((T \, o \, M) \cdot UNIV)))
\]

**By** (simp add: bd-cts-tr-def)

4.3.1 Continuity of Primitives

**Lemma** cts-wp-Abort:

\[ bd-cts \, (wp \, (Abort :: 's prog)) \]

**Proof**

- **Have** X: range (\lambda i::nat \cdot (s::'s). 0) = \{\lambda s. 0\} **by** (auto)
- **Show** ?thesis **by** (intro bd-ctsI, simp add: wp-eval o-def Sup-exp-def X)

**Qed**

**Lemma** cts-wp-Skip:

\[ bd-cts \, (wp \, Skip) \]

**By** (rule bd-ctsI, simp add: wp-def Skip-def o-def)

**Lemma** cts-wp-Apply:

\[ bd-cts \, (wp \, (Apply \, f)) \]

**Proof**
have \( X : \forall M \ s \ . \ \{ P \ (f \ s) \ | \ P. \ P \in \ range \ M \} = \{ P \ s \ | \ P. \ P \in \ range \ (\lambda i \ s \ . \ M \ i \ (f \ s)) \} \) by(auto)

show \(?thesis by(intro bd-ctsI ext, simp add:wp-eval o-def Sup-exp-def X)\)
qed

**lemma cts-wp-Bind:**

fixes \(a::'a \Rightarrow 's \ prog\)

assumes \(ca: \forall s. \ bd-cts \ (wp \ (a \ (f \ s)))\)

shows \(bd-cts \ (wp \ (Bind \ f \ a)))\)

**proof(rule bd-ctsI)**

fix \(M::nat \Rightarrow 's \ expect\ and \ c::real\)

assume \(chain: \forall i. \ M \ i \vdash M \ (Suc \ i) \ and\ sM: \forall i. \ sound \ (M \ i)\)

and \(bM: \forall i. \ bounded-by \ c \ (M \ i)\)

with \(bd-ctsD[OF ca]\)

have \(\forall s. \ wp \ (a \ (f \ s)) \ (Sup-exp \ (range \ M)) =\)

Sup-exp \ (range \ (wp \ (a \ (f \ s)) \ o \ M))\)

by(auto)

moreover have \(\forall s. \ \{fa \ s \ | \ fa. \ fa \in \ range \ (\lambda x. \ wp \ (a \ (f \ s)) \ (M \ x))\} =\)

\(\{fa \ s \ | \ fa. \ fa \in \ range \ (\lambda x. \ wp \ (a \ (f \ s)) \ (M \ x) \ s)\}\)

by(auto)

ultimately show \(wp \ (Bind \ f \ a) \ (Sup-exp \ (range \ M)) =\)

Sup-exp \ (range \ (wp \ (Bind \ f \ a) \ o \ M))\)

by(simp add:wp-eval o-def Sup-exp-def)
qed

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

**lemma cts-wp-DC:**

fixes \(a \ b::'s \ prog\)

assumes \(ca: \ bd-cts \ (wp \ a)\)

and \(cb: \ bd-cts \ (wp \ b)\)

and \(ha: \ healthy \ (wp \ a)\)

and \(hb: \ healthy \ (wp \ b)\)

shows \(bd-cts \ (wp \ (a \ ∪ \ b))\)

**proof(rule bd-ctsI, rule antisym)**

fix \(M::nat \Rightarrow 's \ expect\ and \ c::real\)

assume \(chain: \forall i. \ M \ i \vdash M \ (Suc \ i) \ and\ sM: \forall i. \ sound \ (M \ i)\)

and \(bM: \forall i. \ bounded-by \ c \ (M \ i)\)

from \(ha \ hb\) have \(hab: \ healthy \ (wp \ (a \ ∪ \ b))\) by(rule healthy-intros)

from \(bM\) have \(leSup: \forall i. \ M \ i \vdash Sup-exp \ (range \ M)\) by(auto intro:Sup-exp-upper)

from \(sM \ bM\) have \(sSup: \ sound \ (Sup-exp \ (range \ M))\) by(auto intro:Sup-exp-sound)

show \(Sup-exp \ (range \ (wp \ (a \ ∪ \ b) \ o \ M)) \vdash wp \ (a \ ∪ \ b) \ (Sup-exp \ (range \ M))\)

**proof(rule Sup-exp-least, clarsimp, rule le-funI)**

fix \(i \ s\)

from \(\ mono-transD[OF healthy-monoD[OF hab]]\) have \(leSup \ sM \ sSup\)

have \(wp \ (a \ ∪ \ b) \ (M \ i) \vdash wp \ (a \ ∪ \ b) \ (Sup-exp \ (range \ M))\) by(auto)
4.3. CONTINUITY

Thus \( \text{wp} \ (a \cap b) \ (M \ i) \ s \leq \text{wp} \ (a \cap b) \ (\text{Sup-exp} \ (\text{range} \ M)) \ s \ \text{by(auto)} \)

From \( \cup b \ \text{have sound} \ (\text{wp} \ (a \cap b) \ (\text{Sup-exp} \ (\text{range} \ M))) \ \text{by(auto)} \)

Thus \( \text{neg} \ (\text{wp} \ (a \cap b) \ (\text{Sup-exp} \ (\text{range} \ M))) \ \text{by(auto)} \)

Qed

From \( sM \ bM \ ha \ \text{have} \ \text{\( \land \)} i. \ \text{bounded-by c} \ (\text{wp} \ a \ (M \ i)) \ \text{by(auto)} \)

Hence \( baM: \ \text{\( \land \)} i. s. \ \text{wp} \ a \ (M \ i) \ s \leq c \ \text{by(auto)} \)

From \( sM \ bM \ hb \ \text{have} \ \text{\( \land \)} i. \ \text{bounded-by c} \ (\text{wp} \ b \ (M \ i)) \ \text{by(auto)} \)

Hence \( bbM: \ \text{\( \land \)} i. s. \ \text{wp} \ b \ (M \ i) s \leq c \ \text{by(auto)} \)

Show \( \text{wp} \ (a \cap b) \ (\text{Sup-exp} \ (\text{range} \ M)) \vdash \text{Sup-exp} \ (\text{range} \ (\text{wp} \ (a \cap b) \circ M)) \)

Proof: \( \text{simp add:wp-eval o-def, rule le-funI} \)

Fix \( s:\ 's \)

From \( \text{bd-ctsD} \{ \text{OF ca, of M, OF chain sM bM} \} \ \text{bd-ctsD} \{ \text{OF cb, of M, OF chain sM bM} \} \)

Have \( \text{min} \ (\text{wp} \ a \ (\text{Sup-exp} \ (\text{range} \ M)) \ s) \ (\text{wp} \ b \ (\text{Sup-exp} \ (\text{range} \ M)) \ s) = \text{min} \ (\text{Sup-exp} \ (\text{range} \ (\text{wp} \ a o M)) \ s) \ (\text{Sup-exp} \ (\text{range} \ (\text{wp} \ b o M)) \ s) \)

by(simp)

Also \{\)

Have \( \{ f \ s | f, f \in \text{range} \ (\lambda x. \ \text{wp} \ a \ (M \ x)) \} = \text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s) \)

\( \{ f \ s | f, f \in \text{range} \ (\lambda x. \ \text{wp} \ b \ (M \ x)) \} = \text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s) \)

by(auto)

Hence \( \text{min} \ (\text{Sup-exp} \ (\text{range} \ (\text{wp} \ a o M)) \ s) \ (\text{Sup-exp} \ (\text{range} \ (\text{wp} \ b o M)) \ s) = \text{min} \ (\text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s))) \ (\text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s))) \)

by(simp add:Sup-exp-def o-def)

\}

Also \{\)

Have \( (\lambda i. \ \text{wp} \ a \ (M \ i) \ s) \longrightarrow \text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s)) \)

Proof: \( \text{rule increasing-LIMSEQ} \)

Fix \( n \)

From \( \text{mono-transD} \{ \text{OF healthy-monoD, OF ha} \} \ \text{sM chain} \)

Show \( \text{wp} \ a \ (M \ n) \ s \leq \text{wp} \ a \ (M \ \text{Suc} \ n) \ s \ \text{by(auto intro:le-funD)} \)

From \( baM \ \text{show} \ \text{wp} \ a \ (M \ n) \ s \leq \text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s)) \)

by(intro cSup-upper bdd-aboveI, auto)

Fix \( c: \text{real assume pe: 0 < e} \)

From \( baM \ \text{have cSup: Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s)) \in \text{closure} \ (\text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s)) \)

by(blast intro:closure-contains-Sup)

With \( pe \ \text{obtain y where yin: y} \in \ (\text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s)) \)

And \( dy: \text{dist} \ y \ (\text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s))) < e \)

by(blast dest:iffD1 \{ \text{OF closure-approachable} \})

From \( yin \ \text{obtain i where y = wp} \ a \ (M \ i) \ s \ \text{by(auto)} \)

With \( dy \ \text{have dist} \ (\text{wp} \ a \ (M \ i) \ s) \ (\text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s))) < e \)

by(simp)

Moreover from \( baM \ \text{have wp} \ a \ (M \ i) \ s \leq \text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s)) \)

by(intro cSup-upper bdd-aboveI, auto)
ultimately have $\sup (\text{range } (\lambda i. \text{wp } a (M i) s)) \leq \text{wp } a (M i) s + e$

by (simp add: dist-real-def)

thus $\exists i. \sup (\text{range } (\lambda i. \text{wp } a (M i) s)) \leq \text{wp } a (M i) s + e$ by (auto)

qed

moreover

have $(\lambda i. \text{wp } b (M i) s) \longrightarrow \sup (\text{range } (\lambda i. \text{wp } b (M i) s))$

proof (rule increasing-LIMSEQ)

fix $n$

from mono-transD[OF healthy-monoD, OF $hb$] $sm$ chain

show $wp b (M n) s \leq wp b (M (Suc n)) s$ by (auto intro: le-funD)

from $bbM$ show $wp b (M n) s \leq \sup (\text{range } (\lambda i. \text{wp } b (M i) s))$

by (intro cSup-upper bdd-aboveI, auto)

fix $c$: real assume $pc: 0 < e$

from $bbM$ have $c\sup: \sup (\text{range } (\lambda i. \text{wp } b (M i) s)) \in \text{closure } (\text{range } (\lambda i. \text{wp } b (M i) s))$

by (blast intro: closure-contains-Sup)

with $pe$ obtain $y$ where $yin: y \in (\text{range } (\lambda i. \text{wp } b (M i) s))$

and $dy: \text{dist } y (\sup (\text{range } (\lambda i. \text{wp } b (M i) s))) < e$

by (blast dest: iffD1[OF closure-approachable])

from $yin$ obtain $i$ where $y = wp b (M i) s$ by (auto)

with $dy$ have $\text{dist } (wp b (M i) s) (\sup (\text{range } (\lambda i. \text{wp } b (M i) s))) < e$

by (simp)

moreover from $bbM$ have $wp b (M i) s \leq \sup (\text{range } (\lambda i. \text{wp } b (M i) s))$

by (intro cSup-upper bdd-aboveI, auto)

ultimately have $\sup (\text{range } (\lambda i. \text{wp } b (M i) s)) \leq wp b (M i) s + e$

by (simp add: dist-real-def)

thus $\exists i. \sup (\text{range } (\lambda i. \text{wp } b (M i) s)) \leq wp b (M i) s + e$ by (auto)

qed

ultimately have $(\lambda i. \text{min } (wp a (M i) s) (wp b (M i) s)) \longrightarrow \\
\min (\sup (\text{range } (\lambda i. \text{wp } a (M i) s))) (\sup (\text{range } (\lambda i. \text{wp } b (M i) s)))$

by (rule tendsto-min)

moreover have $\text{bdd-above } (\text{range } (\lambda i. \text{min } (wp a (M i) s) (wp b (M i) s)))$

proof (intro bdd-aboveI, clarsimp)

fix $i$

have $\text{min } (wp a (M i) s) (wp b (M i) s) \leq wp a (M i) s$ by (auto)

also {

from $ha$ $sm$ $bm$ have bounded-by $c$ $(wp a (M i))$ by (auto)

hence $wp a (M i) s \leq c$ by (auto)

}

finally show $\text{min } (wp a (M i) s) (wp b (M i) s) \leq c$.

qed

ultimately have $\text{min } (\sup (\text{range } (\lambda i. \text{wp } a (M i) s))) (\sup (\text{range } (\lambda i. \text{wp } b (M i) s)))$

by (blast intro: LIMSEQ-le-const2 cSup-upper min mono[OF $baM$ $bbM$])
4.3. CONTINUITY

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))) s 
  by (simp add: Sup-exp-def cong del: strong-SUP-cong) 
} 

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 ⊢ ⊢ (M (Suc i)) 
    and sM: ∀i. sound (M i) 
    and bM: ∀i. bounded-by c (M i) 
  hence wp a (wp b (Sup-exp (range M))) = wp a (Sup-exp (range (wp b o M))) 
    by (subst bd-ctsD[OF cb], auto) 
  also { 
    from sM hb have ∀i. sound ((wp b o M) i) by(auto) 
    moreover from chain sM 
    have ∀i. (wp b o M) i ⊢ (wp b o M) (Suc i) 
      by (auto intro:mono-transD[OF healthy-monoD, OF hb]) 
    moreover from sM bM hb have ∀i. bounded-by c ((wp b o M) i) by(auto) 
    ultimately have wp a (Sup-exp (range (wp b o M))) = 
      Sup-exp (range (wp a o (wp b o M))) 
    by (subst bd-ctsD[OF ca], auto) 
  } 
  also have Sup-exp (range (wp a o (wp b o M))) = 
    Sup-exp (range (λi. wp a (wp b (M i)))) 
  by (simp add:o-def) 
  finally show wp a (wp b (Sup-exp (range M))) = 
    Sup-exp (range (λi. wp a (wp b (M i)))) . 
qed 

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

from \( sM \) have \( \forall i. \iunitary \ (M \ i) \) by(auto)
with \( bM \) have \( nc: \theta \leq c \) by(auto)

from chain \( sM \) \( bM \) have \( wp \ a \ (Sup-exp \ (range \ M)) = Sup-exp \ (range \ (wp \ a \ o \ M)) \)
by(rule bd-ctsD[OF ca])

hence \( wp \ a \ (Sup-exp \ (range \ M)) \ s = Sup-exp \ (range \ (wp \ a \ o \ M)) \ s \)
by(simp)
also \{ 
  have \( \{ f | s \ f \in range \ ((\lambda x. \ wp \ a \ (M \ x))) \} = range \ ((\lambda i. \ wp \ a \ (M \ i)) \ s) \)
  by(auto)
  hence \( Sup-exp \ (range \ (wp \ a \ o \ M)) \ s = Sup \ (range \ ((\lambda i. \ wp \ a \ (M \ i)) \ s)) \)
  by(simp add:Sup-exp-def o-def)
\}

finally have \( p \ s \ast \ wp \ a \ (Sup-exp \ (range \ M)) \ s = \)
\( p \ s \ast Sup \ (range \ ((\lambda i. \ wp \ a \ (M \ i)) \ s)) \) by(simp)
also have \( ... = Sup \ \{ p \ s \ast x \ | x \in range \ ((\lambda i. \ wp \ a \ (M \ i)) \ s) \} \)
proof (rule cSup-mult, blast, clarsimp)
from \( wp \) show \( \theta \leq p \ s \) by(auto)
fix \( i \)
from \( sM \) \( bM \) ha have \( \bounded-by \ c \ (wp \ a \ (M \ i)) \) by(auto)
thus \( wp \ a \ (M \ i) \ s \leq c \) by(auto)
qed
also \{ 
  have \( \{ p \ s \ast x \ | x \in range \ ((\lambda i. \ wp \ a \ (M \ i)) \ s) \} = range \ ((\lambda i. \ p \ s \ast wp \ a \ (M \ i)) \ s) \)
  by(auto)
  hence \( Sup \ \{ p \ s \ast x \ | x \in range \ ((\lambda i. \ wp \ a \ (M \ i)) \ s) \} = \)
  \( Sup \ (range \ ((\lambda i. \ p \ s \ast wp \ a \ (M \ i)) \ s)) \) by(simp)
\}

finally have \( p \ s \ast wp \ a \ (Sup-exp \ (range \ M)) \ s = Sup \ (range \ ((\lambda i. \ p \ s \ast wp \ a \ (M \ i)) \ s)) \).

moreover \{ 
  from chain \( sM \) \( bM \) have \( wp \ b \ (Sup-exp \ (range \ M)) = Sup-exp \ (range \ (wp \ b \ o \ M)) \)
by(rule bd-ctsD[OF cb])
  hence \( wp \ b \ (Sup-exp \ (range \ M)) \ s = Sup-exp \ (range \ (wp \ b \ o \ M)) \ s \)
by(simp)
also \{ 
  have \( \{ f | s \ f \in range \ ((\lambda x. \ wp \ b \ (M \ x))) \} = range \ ((\lambda i. \ wp \ b \ (M \ i)) \ s) \)
  by(auto)
  hence \( Sup-exp \ (range \ (wp \ b \ o \ M)) \ s = Sup \ (range \ ((\lambda i. \ wp \ b \ (M \ i)) \ s)) \)
\}
4.3. CONTINUITY

\begin{align*}
\text{by}(\text{simp add: Sup-exp-def o-def}) \\
\text{finally have } (1 - p s) \ast wp b (\text{Sup-exp (range } M)) s = \\
(1 - p s) \ast \text{Sup (range } (\lambda i. wp b (M i) s)) \text{ by(simp)} \\
\text{also have } \ldots = \text{Sup } \{(1 - p s) \ast x \mid x \in \text{range } (\lambda i. wp b (M i) s)\} \\
\text{qed}
\end{align*}

\begin{align*}
\text{also } \{ \\
\text{have } \{(1 - p s) \ast x \mid x \in \text{range } (\lambda i. wp b (M i) s)\} = \\
\text{range } (\lambda i. (1 - p s) \ast wp b (M i) s) \text{ by(auto)} \\
\text{hence } \text{Sup } \{(1 - p s) \ast x \mid x \in \text{range } (\lambda i. wp b (M i) s)\} = \\
\text{Sup } \text{range } (\lambda i. (1 - p s) \ast wp b (M i) s) \text{ by(simp)} \\
\text{finally have } (1 - p s) \ast wp b (\text{Sup-exp (range } M)) s = \\
\text{Sup } \text{range } (\lambda i. (1 - p s) \ast wp b (M i) s)) \text{.}
\end{align*}

\begin{align*}
\text{ultimately have } p s \ast wp a (\text{Sup-exp (range } M)) s + (1 - p s) \ast wp b (\text{Sup-exp (range } M)) s = \\
\text{Sup } \text{range } (\lambda i. p s \ast wp a (M i) s)) + \text{Sup } \text{range } (\lambda i. (1 - p s) \ast wp b (M i) s)) \\
\text{by(simp)} \\
\text{also } \{ \\
\text{from } bM sM ha \text{ have } \bigwedge i. \text{bounded-by c (wp a (M i)) by(auto)} \\
\text{hence } \bigwedge i. wp a (M i) s \leq c \text{ by(auto)} \\
\text{moreover from } wp \text{ have } 0 \leq p s \text{ by(auto)} \\
\text{ultimately have } \bigwedge i. p s \ast wp a (M i) s \leq p s \ast c \text{ by(auto intro:mult-left-mono)} \\
\text{also from } wp nc \text{ have } p s \ast c \leq 1 \ast c \text{ by(blast intro:mult-right-mono)} \\
\text{also have } \ldots = c \text{ by(simp)} \\
\text{finally have } baM: \bigwedge i. p s \ast wp a (M i) s \leq c \text{.}
\end{align*}

\begin{align*}
\text{have lima: (}\lambda i. p s \ast wp a (M i) s) \longrightarrow \text{Sup } \text{range } (\lambda i. p s \ast wp a (M i) s)) \\
\text{proof (rule increasing-LIMSEQ)} \\
\text{fix } n \\
\text{from } sM \text{ chain healthy-monoD[OF ha] have } wp a (M n) \vdash wp a (M (Suc n)) \text{ by(auto)} \\
\text{with } wp \text{ show } p s \ast wp a (M n) s \leq p s \ast wp a (M (Suc n)) s \text{ by(blast intro:mult-left-mono)} \\
\text{from } baM \text{ show } p s \ast wp a (M n) s \leq \text{Sup } \text{range } (\lambda i. p s \ast wp a (M i) s) \text{ by(intro cSup-upper bdd-aboveI, auto)} \\
\text{next } \\
\text{fix } e::\text{real} \\
\text{assume } pe: 0 < e
\end{align*}
from \text{baM} have \( \text{Sup} (\text{range} (\lambda i. \text{p s} * \text{wp a} (M i) s)) \in \text{closure} (\text{range} (\lambda i. \text{p s} * \text{wp a} (M i) s)) \)
  by (blast intro: closure-contains-Sup)

\textbf{thm} closure-approachable

with \text{pe} obtain \( y \) where \( yin: y \in \text{range} (\lambda i. \text{p s} * \text{wp a} (M i) s) \)
  and \( dy: \text{dist} y (\text{Sup} (\text{range} (\lambda i. \text{p s} * \text{wp a} (M i) s))) < e \)
  by (blast dest: iffD1 [OF closure-approachable])

from \( yin \) obtain \( i \) where \( y = \text{p s} * \text{wp a} (M i) s \) by (auto)

with \( dy \) have \( \text{dist} (\text{p s} * \text{wp a} (M i) s) (\text{Sup} (\text{range} (\lambda i. \text{p s} * \text{wp a} (M i) s))) < e \)
  by (simp)

moreover from \text{baM} have \( \text{p s} * \text{wp a} (M i) s \leq \text{Sup} (\text{range} (\lambda i. \text{p s} * \text{wp a} (M i) s)) \)
  by (intro cSup-upper bdd-aboveI, auto)

ultimately have \( \text{Sup} (\text{range} (\lambda i. \text{p s} * \text{wp a} (M i) s)) \leq \text{p s} * \text{wp a} (M i) s + e \)
  by (simp add: dist-real-def)

thus \( \exists i. \text{Sup} (\text{range} (\lambda i. \text{p s} * \text{wp a} (M i) s)) \leq \text{p s} * \text{wp a} (M i) s + e \)
  by (auto)

qed

from \text{bm sM hb} have \( \bigwedge i. \text{bounded-by c (wp b (M i))} \) by (auto)

hence \( \bigwedge i. \text{wp b (M i) s} \leq c \) by (auto)

moreover from \text{wp have} \( \theta \leq (1 - \text{p s}) \) by (auto simp: sign-simps)

ultimately have \( \bigwedge i. (1 - \text{p s}) * \text{wp b (M i) s} \leq (1 - \text{p s}) * c \) by (auto intro: mult-left-mono)

also {
  from \text{wp have} \( 1 - \text{p s} \leq 1 \) by (auto)

  with \( \text{nc} \) have \( (1 - \text{p s}) * c \leq 1 * c \) by (blast intro: mult-right-mono)
}

also have \( 1 * c = c \) by (simp)

finally have \( \text{bbM} \): \( \bigwedge i. (1 - \text{p s}) * \text{wp b (M i) s} \leq c \).

have \( \text{limb}: (\lambda i. (1 - \text{p s}) * \text{wp b (M i) s}) \longrightarrow \text{Sup} (\text{range} (\lambda i. (1 - \text{p s}) * \text{wp b (M i) s})) \)

\textbf{proof} (\text{rule increasing-LIMSEQ})

fix \( n \)

from \( \text{sM chain healthy-monoD[OF hb]} \) have \( \text{wp b (M n)} \not\leq \text{wp b (M (Suc n))} \)
  by (auto)

moreover from \text{wp have} \( \theta \leq 1 - \text{p s} \) by (auto simp: sign-simps)

ultimately show \( (1 - \text{p s}) * \text{wp b (M n) s} \leq (1 - \text{p s}) * \text{wp b (M (Suc n))} \)

s

by (blast intro: mult-left-mono)

from \( \text{bbM show} (1 - \text{p s}) * \text{wp b (M n) s} \leq \text{Sup} (\text{range} (\lambda i. (1 - \text{p s}) * \text{wp b (M i) s})) \)
  by (intro cSup-upper bdd-aboveI, auto)

next

fix \( e::\text{real} \)

assume \( \text{pe}: 0 < e \)
from $bbM$ have $\sup (\text{range} (\lambda_i. (1 - p) s) \ast wp (M i) s) \leq$
\text{closure} (\text{range} (\lambda_i. (1 - p) s) \ast wp (M i) s))
\text{by(blast intro: closure-contains-sup)}
with $pe$ obtain $y$ where $y \in \text{range} (\lambda_i. (1 - p) s) \ast wp (M i) s$
\text{and dy: dist} $y \{ \sup (\text{range} (\lambda_i. (1 - p) s) \ast wp (M i) s)) < e$
\text{by(blast dest: iffd1 [OF closure-approachable]})
from $yin$ obtain $i$ where $y = (1 - p) s \ast wp (M i) s$ \text{by(auto)}
with $dy$ have $\text{dist} ((1 - p) s) \ast wp (M i) s$
$\{ \sup (\text{range} (\lambda_i. (1 - p) s) \ast wp (M i) s)) < e$
\text{by(simp)}
\text{moreover from $bbM$}
have $(1 - p) s \ast wp (M i) s \leq \sup (\text{range} (\lambda_i. (1 - p) s) \ast wp (M i) s))$
\text{by(intro cSup-upper bdd-aboveI, auto)}
ultimately have $\sup (\text{range} (\lambda_i. (1 - p) s) \ast wp (M i) s)) \leq (1 - p) s \ast wp (M i) s + e$
\text{by(simp add: dist-real-def)}
\text{thus } \exists i. \sup (\text{range} (\lambda_i. (1 - p) s) \ast wp (M i) s)) \leq (1 - p) s \ast wp (M i) s + e \text{ by(auto)}
\text{qed}

from $lima \ limb$ have $(\lambda_i. p s \ast wp a (M i) s) + (1 - p) s \ast wp b (M i) s)\sup (\text{range} (\lambda_i. (1 - p) s) \ast wp (M i) s))$
\text{by(rule tendsto-add)}
\text{moreover from add-mono[OF baM $bbM$]}
have $\lambda i: p s \ast wp a (M i) s + (1 - p) s \ast wp b (M i) s \leq$
$\sup (\text{range} (\lambda_i. p s \ast wp a (M i) s) + (1 - p) s \ast wp b (M i) s))$
\text{by(intro cSup-upper bdd-aboveI, auto)}
ultimately have $\sup (\text{range} (\lambda_i. p s \ast wp a (M i) s)) +$
$\sup (\text{range} (\lambda_i. (1 - p) s) \ast wp b (M i) s)) \leq$
$\sup (\text{range} (\lambda_i. p s \ast wp a (M i) s) + (1 - p) s \ast wp b (M i) s))$
\text{by(blast intro: LIMSEQ-le-const2)}$
\text{also}\}
\text{have range} (\lambda_i. p s \ast wp a (M i) s + (1 - p) s) \ast wp b (M i) s) =
\{ f s | f \in \text{range} (\lambda x. p s \ast wp a (M x) s + (1 - p) s) \ast wp b (M x) s)\}$
\text{by(auto)}
\text{hence}$\sup (\text{range} (\lambda_i. p s \ast wp a (M i) s) + (1 - p) s) \ast wp b (M i) s)) =$
$\sup (\text{range} (\lambda x. p s \ast wp a (M x) s + (1 - p) s) \ast wp b (M x) s))$
\text{by (simp add: Sup-exp-def cong del: strong-SUP-cong)}$
\text{finaly}\}
\text{have} p s \ast wp a (\text{Sup-exp (range M)}) s + (1 - p) s) \ast wp b (\text{Sup-exp (range M)}) s \leq$
$\sup (\text{range} (\lambda_i s. p s \ast wp a (M i) s) + (1 - p) s) \ast wp b (M i) s)))$
\text{moreover}\}
\text{have} \sup (\text{range} (\lambda_i s. p s \ast wp a (M i) s) + (1 - p) s) \ast wp b (M i) s)) s \leq$
$p s \ast wp a (\text{Sup-exp (range M)}) s + (1 - p) s) \ast wp b (\text{Sup-exp (range M)}) s$
proof (rule le-funD (OF Sup-exp-least), clarsimp, rule le-funI)
  fix i::nat and s::'s
  from bM have leSup: M i ⊢ Sup-exp (range M)
    by (blast intro: Sup-exp-upper)
  moreover from sM bM have sSup: sound (Sup-exp (range M))
    by (auto intro: Sup-exp-sound)
  moreover note healthy-monoD[OF ha]
  ultimately have wp a (M i) ⊢ wp a (Sup-exp (range M))
    by (auto intro: Sup-exp-upper)
  hence wp a (M i) s ≤ wp a (Sup-exp (range M)) s by (auto)
  moreover { from leSup sSup healthy-monoD[OF hb]
    have wp b (M i) ⊢ wp b (Sup-exp (range M))
      by (auto intro: Sup-exp-upper)
    hence wp b (M i) s ≤ wp b (Sup-exp (range M)) s by (auto)
  }
  moreover from ap have 0 ≤ p s 0 ≤ 1 − p s by (auto simp: sign-simps)
  ultimately show p s * wp a (M i) s + (1 − p s) * wp b (M i) s ≤
    p s * wp a (Sup-exp (range M)) s + (1 − p s) * wp b (Sup-exp (range M))
    s
    by (blast intro: add-mono mult-left-mono)
  from sSup ha hb have sound (wp a (Sup-exp (range M)))
    sound (wp b (Sup-exp (range M)))
    by (auto)
  hence ∀s. 0 ≤ wp a (Sup-exp (range M)) s ∧ s. 0 ≤ wp b (Sup-exp (range M)) s
    by (auto)
  moreover from ap have ∀s. 0 ≤ p s ∧ s. 0 ≤ 1 − p s by (auto simp: sign-simps)
  ultimately show nneg (λc. p c * wp a (Sup-exp (range M)) c +
    (1 − p c) * wp b (Sup-exp (range M)) c)
    by (blast intro: add-nonneg-nonneg mult-nonneg-nonneg)
  qed
  ultimately show p s * wp a (Sup-exp (range M)) s + (1 − p s) * wp b (Sup-exp (range M)) s =
    Sup-exp (range (λx s. p s * wp a (M x) s + (1 − p s) * wp b (M x) s))
    s
    by (auto)
  qed

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

lemma SetPC-Bind:
  SetPC a p = Bind p (λp. SetPC a (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
4.3. CONTINUITY

and \( \text{fsupp: finite (supp p)} \) shows \( \text{SetPC a (}\lambda \cdot \text{p}) = PC (a \cdot x) (\lambda \cdot \text{p x}) \) \( \text{(SetPC a (}\lambda \cdot \text{dist-remove p x)} \) \)

proof(intro ext, simp add:SetPC-def PC-def)

fix \( ab \) \( P \) \( s \)

from \( \text{nz} \) have \( x \in \text{supp p} \) by (simp add: supp-def)

hence \( \text{supp p} = \text{insert x (supp p - \{x\})} \) by (auto)

hence \( \sum_{x \in \text{supp p}} p x \cdot a x \cdot ab \cdot P \cdot s = \text{insert x (supp p - \{x\})}. p x \cdot a x \cdot ab \cdot P \cdot s \)

by (simp)

also from \( \text{fsupp} \) have \( \ldots = \text{p x} \cdot \text{a x} \cdot ab \cdot P \cdot s + \sum_{x \in \text{supp p}} (1 - p x) \cdot ((\sum_{x \in \text{supp p}} p x - \{x\}). p x \cdot a x \cdot ab \cdot P \cdot s) / (1 - p x) \)

by (simp add:field-simps)

also have \( \ldots = \text{p x} \cdot \text{a x} \cdot ab \cdot P \cdot s + (1 - p x) \cdot ((\sum_{y \in \text{supp p}} p y - \{x\}). p y / (1 - p x) \cdot a y \cdot ab \cdot P \cdot s)) \)

by (simp add:sum-divide-distrib)

also have \( \ldots = \text{p x} \cdot \text{a x} \cdot ab \cdot P \cdot s + (1 - p x) \cdot ((\sum_{y \in \text{supp p}} p y - \{x\}). \text{dist-remove p x y} \cdot a y \cdot ab \cdot P \cdot s)) \)

by (simp add:dist-remove-def)

also from \( \text{nz n1} \) have \( \ldots = \text{p x} \cdot \text{a x} \cdot ab \cdot P \cdot s + (1 - p x) \cdot ((\sum_{y \in \text{supp (dist-remove p x)}} \text{dist-remove p x y} \cdot a y \cdot ab \cdot P \cdot s)) \)

by (simp add:supp-dist-remove)

finally show \( \sum_{x \in \text{supp p}} p x \cdot a x \cdot ab \cdot P \cdot s = p x \cdot a x \cdot ab \cdot P \cdot s + (1 - p x) \cdot ((\sum_{y \in \text{supp (dist-remove p x)}} \text{dist-remove p x y} \cdot a y \cdot ab \cdot P \cdot s)) \).

qed

lemma cts-bot:

\( bd-cts (\lambda(P::'s expect) (s::'s). 0::real) \)

proof -

have \( X: \text{\{P::'s expect\} s | P. P \in \text{range (}\lambda P s. 0\)} = \{0\} \) by (auto)

show \( ?\text{thesis} \) by (intro bd-ctsI, simp add:Sup-exp-def o-def X)

qed

lemma wp-SetPC-nil:

\( wp (\text{SetPC a (}\lambda s a. 0\)) = (\lambda P s. 0) \)

by (intro ext, simp add:wp-eval)

lemma SetPC-sgl:

\( \text{supp p} = \{x\} \implies \text{SetPC a (}\lambda \cdot \text{p}) = (\lambda ab \cdot P s. p x \cdot a x \cdot ab \cdot P \cdot s) \)

by (simp add: SetPC-def)

lemma bd-cts-scale:
fixes a::'s trans
assumes ca: bd-cts a
  and ha: healthy a
  and nnc: 0 ≤ c
shows bd-cts (λP s. c * a P s)
proof (intro bd-ctsI ext, simp add:o-def)
  fix M::nat ⇒ 's expect and d::real and s::'
  assume chain: \( \forall i. M i \vdash (Succ i) \) and sM: \( \forall i. sound (M i) \)
  and bM: \( \forall i. bounded-by d (M i) \)
  
  from sM have \( \forall i. nneg (M i) \) by (auto)
  with bM have nnd: \( 0 \leq d \) by (auto)

  from sM bM have sSup: sound (Sup-exp (range M)) by (auto intro:Sup-exp-sound)
  with healthy-scalingD[OF ha] nnc
  have c * a (Sup-exp (range M)) s = a (λs. c * Sup-exp (range M) s) s
    by (auto intro:scalingD)
  also { 
    have \( \forall s. \{ f s | f \in range M \} = range (λi. M i s) \) by (auto)
    hence a (λs. c * Sup-exp (range M) s) s =
      a (λs. c * Sup (range (λi. M i s))) s
    by (simp add:Sup-exp-def)
  }
  also { 
    from bM have \( \forall x s. x \in range (λi. M i s) \Longrightarrow x \leq d \) by (auto)
    with nnc have a (λs. c * Sup (range (λi. M i s))) s =
      a (λs. Sup (\{ c*x | x. x \in range (λi. M i s) \}) s)
    by (substs cSup-mult, blast+)
  }
  also { 
    have X: \( \forall s. \{ c * x | x. x \in range (λi. M i s) \} = range (λi. c * M i s) \) by (auto)
    have a (λs. Sup (\{ c * x | x. x \in range (λi. M i s) \}) s =
      a (λs. Sup (range (λi. c * M i s))) s by (simp add:X)
  }
  also { 
    have \( \forall s. range (λi. c * M i s) = \{ f s | f \in range (λi s. c * M i s) \} \)
      by (auto)
    hence (λs. Sup (range (λi. c * M i s))) = Sup-exp (range (λi s. c * M i s))
      by (simp add: Sup-exp-def cong del: strong-SUP-cong)
    hence a (λs. Sup (range (λi. c * M i s))) s =
      a (Sup-exp (range (λi s. c * M i s))) s by (simp)
  }
  also { 
    from le-funD[OF chain] nnc
    have \( \forall i. (λs. c * M i s) \vdash (λs. c * M (Suc i) s) \)
      by (auto intro:le-funI[OF mult-left-mono])
  moreover from sM nnc
    have \( \forall i. sound (λs. c * M i s) \)
      by (auto intro:sound-intros)
4.3. CONTINUITY

moreover from \( bM \) nnc have \( \bigwedge i. \text{bounded-by} (c \ast d) (\lambda s. c \ast M i s) \)
by(auto intro: mult-left-mono)
ultimately have \( a \ (\text{Sup-exp} \ (\text{range} \ (\lambda i s. c \ast M i s))) = \)
\( \text{Sup-exp} \ (\text{range} \ (a \circ (\lambda i s. c \ast M i s))) \)
by(rule bd-ctsD[OF ca])
hence \( a \ (\text{Sup-exp} \ (\text{range} \ (\lambda i s. c \ast M i s))) s = \)
\( \text{Sup-exp} \ (\text{range} \ (a \circ (\lambda i s. c \ast M i s))) s \)
by(auto)

also have \( \text{Sup-exp} \ (\text{range} \ (a \circ (\lambda i s. c \ast M i s))) s = \)
\( \text{Sup-exp} \ (\text{range} \ (\lambda x. a \ (\lambda s. c \ast M x s))) s \)
by(simp add: a-o-def)
also { from nnc sM have \( \bigwedge x. a \ (\lambda s. c \ast M x s) = (\lambda s. c \ast a \ (M x) s) \)
by(auto intro: scalingD[OF healthy-scalingD, OF ha, symmetric])
hence \( \text{Sup-exp} \ (\text{range} \ (\lambda x. a \ (\lambda s. c \ast M x s))) s = \)
\( \text{Sup-exp} \ (\text{range} \ (\lambda x s. c \ast a \ (M x) s)) s \)
by(simp)
}

finally show \( c \ast a \ (\text{Sup-exp} \ (\text{range} \ M)) s = \text{Sup-exp} \ (\text{range} \ (\lambda x s. c \ast a \ (M x) s)) s \).

qed

lemma cts-wp-SetPC-const:
  fixes \( a :: 'a \Rightarrow 's \text{ prog} \)
  assumes ca: \( \bigwedge x. x \in (\text{supp} \ p) \Rightarrow \text{bd-cts} \ (wp \ (a x)) \)
  and ha: \( \bigwedge x. x \in (\text{supp} \ p) \Rightarrow \text{healthy} \ (wp \ (a x)) \)
  and wp: \text{unitary} \ p
  and sump: \text{sum} \ p \ (\text{supp} \ p) \leq 1
  and fsupp: \text{finite} \ (\text{supp} \ p)
  shows \( \text{bd-cts} \ (wp \ (\text{SetPC} \ a \ (\lambda -. p))) \)
proof(cases \text{supp} \ p = \{\}, simp add: \text{supp-empty} \ \text{SetPC-def} \ wp-def \ \text{cts-bot})
assume nesupp: \( \text{supp} \ p \neq \{\} \)
from fsupp have unitary \( p \rightarrow \text{sum} \ p \ (\text{supp} \ p) \leq 1 \rightarrow \)
\( (\forall x \in \text{supp} \ p. \ \text{bd-cts} \ (wp \ (a x))) \rightarrow \)
\( (\forall x \in \text{supp} \ p. \ \text{healthy} \ (wp \ (a x))) \rightarrow \)
\( \text{bd-cts} \ (wp \ (\text{SetPC} \ a \ (\lambda -. p))) \)
proof(induct \text{supp} \ p \ \text{arbitrary}: p, simp add: \text{supp-empty} \ wp-\text{SetPC-nil} \ \text{cts-bot}, clarify)
fix \( x :: 'a \ \text{set} \ \text{and} \ p :: 'a \Rightarrow \text{real} \)
assume fF: \text{finite} \ F
assume \text{insert} \ x \ F = \text{supp} \ p
hence pstep: \text{supp} \ p = \text{insert} \ x \ F \ by(simp)
hence \text{xin}: \( x \in \text{supp} \ p \) by(auto)
assume \text{wp}: \text{unitary} \ p \ \text{and} \ ca: \( \forall x \in \text{supp} \ p. \ \text{bd-cts} \ (wp \ (a x)) \)
\text{and} ha: \( \forall x \in \text{supp} \ p. \ \text{healthy} \ (wp \ (a x)) \)
and \( \sum p \leq 1 \)

and \( x \notin F \)

assume IH: \( \forall p. F = supp p \rightarrow \)

\( unitary p \rightarrow \sum p (supp p) \leq 1 \rightarrow \)

\( (\forall x \in supp p. \text{bd-cts} (\wp (a x))) \rightarrow \)

\( (\forall x \in supp p. \text{healthy} (\wp (a x))) \rightarrow \)

\( \text{bd-cts} (\wp (\text{SetPC} a (\lambda-. p))) \)

from \( fF \) pstep have fsupp: finite \( (supp p) \) by (auto)

from \( x \in \) have nzp: \( p x \neq 0 \) by (simp add: supp-def)

have xy-le-sum: \( \forall y. y \in supp p \rightarrow y \neq x \rightarrow p x + p y \leq \sum p (supp p) \)

proof -

fix \( y \) assume yin: \( y \in supp p \) and yne: \( y \neq x \)

from \( up \) have \( 0 \leq \sum p (supp p - \{x, y\}) \)

by (auto intro: sum-nonneg)

hence \( p x + p y \leq p x + p y + \sum p (supp p - \{x, y\}) \)

by (auto)

also {

from yin yne fsupp

have \( p x + \sum p (supp p - \{x, y\}) = \sum p (supp p - \{x\}) \)

by (subst sum.insert[symmetric], (blast intro!: sum.cong)+)

moreover

from \( x \in \) suppf

have \( p x + \sum p (supp p - \{x\}) = \sum p (supp p) \)

by (subst sum.insert[symmetric], (blast intro!: sum.cong)+)

ultimately

have \( p x + p y + \sum p (supp p - \{x, y\}) = \sum p (supp p) \) by (simp)

}

finally show \( p x + p y \leq \sum p (supp p) \).

qed

have n1p: \( \forall y. y \in supp p \rightarrow y \neq x \rightarrow p x \neq 1 \)

proof (rule ccontr, simp)

assume px1: \( p x = 1 \)

fix \( y \) assume yin: \( y \in supp p \) and yne: \( y \neq x \)

from \( up \) have \( 0 \leq p y \) by (auto)

with \( yin \) have \( 0 < p y \) by (auto simp: supp-def)

hence \( 0 + p x < p y + p x \) by (rule add-strict-right-mono)

with px1 have \( 1 < p x + p y \) by (simp)

also from yin yne have \( p x + p y \leq \sum p (supp p) \)

by (rule xy-le-sum)

finally show False using sump by (simp)

qed

show \( \text{bd-cts} (\wp (\text{SetPC} a (\lambda-. p))) \)

proof (cases \( F = \{\} \))
4.3. CONTINUITY

\[ \text{case } \text{True with } \text{pstep have } \text{supp } p = \{ x \} \text{ by(simp)} \]

\[ \text{hence } \wp (\text{SetPC } a (\lambda \cdot, p)) = (\lambda P s. p x \ast \wp (a x) P s) \]

\[ \text{by(simp add:SetPC-sgl wp-def)} \]

\[ \text{moreover } \]

\[ \text{from up ca ha xin have bd-cts (wp (a x)) healthy (wp (a x)) } 0 \leq p x \]

\[ \text{by(auto)} \]

\[ \text{hence bd-cts (}\lambda P s. p x \ast \wp (a x) P s) \]

\[ \text{by(rule bd-cts-scale)} \]

\[ \text{ultimately show ?thesis by(simp)} \]

\[ \text{next} \]

\[ \text{assume neF: } F \neq \{\} \]

\[ \text{then obtain y where yinF: } y \in F \text{ by(auto)} \]

\[ \text{with xni have yne: } y \neq x \text{ by(auto)} \]

\[ \text{from yinF pstep have yin: } y \in \text{ supp } p \text{ by(auto)} \]

\[ \text{from supp-dist-remove[of p x, OF nzp n1p, OF yin yne] have supp-sub: supp (dist-remove p x) } \subseteq \text{ supp } p \text{ by(auto)} \]

\[ \text{from xin ca have cax: bd-cts (wp (a x)) by(auto)} \]

\[ \text{from xin ha have hax: healthy (wp (a x)) by(auto)} \]

\[ \text{from supp-sub ha have hra: } \forall x \in \text{ supp (dist-remove p x). healthy (wp (a x))} \]

\[ \text{by(auto)} \]

\[ \text{from supp-dist-remove[of p x, OF nzp n1p, OF yin yne] have Fsupp: } F = \text{ supp (dist-remove p x)} \]

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

\[ \text{have udp: unitary (dist-remove p x)} \]

\[ \text{proof(intro unitaryI2 nnegI bounded-byI)} \]

\[ \text{fix } y \]

\[ \text{show } 0 \leq \text{ dist-remove p x y} \]

\[ \text{proof(cases } y=x, \text{ simp-all add:dist-remove-def)} \]

\[ \text{from up have } 0 \leq p y 0 \leq 1 - p x \text{ by(auto simp:sign-simps)} \]

\[ \text{thus } 0 \leq p y \slash (1 - p x) \]

\[ \text{by(rule divide-nonneg-nonneg)} \]

\[ \text{qed} \]

\[ \text{show dist-remove p x y } \leq 1 \]

\[ \text{proof(cases } y=x, \text{ simp-all add:dist-remove-def, cases } y \in \text{ supp } p, \text{ simp-all add:nsupp-zero)} \]

\[ \text{assume yne: } y \neq x \text{ and yin: } y \in \text{ supp } p \]

\[ \text{hence } p x + p y \leq \text{ sum p (supp p)} \]

\[ \text{by(auto intro:xy-le-sum)} \]

\[ \text{also note sump} \]

\[ \text{finally have } p y \leq 1 - p x \text{ by(auto)} \]

\[ \text{moreover from up have } p x \leq 1 \text{ by(auto)} \]
moreover from \( \text{gin yne} \) have \( p x \neq 1 \) by (rule n1p)
ultimately show \( p y / (1 - p x) \leq 1 \) by (auto)
qed
qed

from \( x n \) have \( \text{pxn0: } p x \neq 0 \) by (auto simp: supp-def)
from \( \text{yin yne} \) have \( \text{pxn1: } p x \neq 1 \) by (rule n1p)

from \( \text{pxn0 pxn1} \) have \( \text{sum (dist-remove } p x) (\text{supp (dist-remove } p x)) = \text{sum (dist-remove } p x) (\text{supp } p - \{x\}) \) by (simp add: supp-dist-remove)
also have \( \ldots = (\sum y \in \text{supp } p - \{x\}. \text{p y} / (1 - p x)) \) by (simp add: dist-remove-def)
also have \( \ldots = (\sum y \in \text{supp } p - \{x\}. \text{p y} / (1 - p x) \) by (simp add: sum-divide-distrib)
also \{
from \( \text{xin} \) have \( \text{insert x (supp p) = supp p by (auto)} \) with \( \text{fsupp} \) have \( \text{pxn1 have } p x \leq 1 \) by (auto)
with \( \text{pxn1 have } p x < 1 \) 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 \( \text{sdp: sum (dist-remove } p x) (\text{supp (dist-remove } p x)) \leq 1 \).

from \( \text{Fsupp udp sdp hra cra IH} \) have \( \text{cts-dr: bd-cts (wp (SetPC a (}\lambda.-. \text{dist-remove } p x))) \) by (auto)

from \( \text{up have upx: unitary (}\lambda.-. p x) \) by (auto)

from \( \text{pxn0 pxn1 fsupp hra show } \text{thesis} \) by (simp add: SetPC-remove,
blast intro:cts-wp-PC cax cts-dr hax healthy-intros
unitary-sound[OF udp] sdp upx)
qed
qed

with \( \text{assms show } \text{thesis by (auto)} \)
qed

lemma \( \text{cts-wp-SetPC:} \) fixes \( a::'a \Rightarrow 's \text{ prog} \) assumes \( ca: \forall x. x \in (\text{supp } p s) \implies \text{bd-cts (wp } a x) \)
4.3. CONTINUITY

and ha: \( \forall x. x \in (\text{supp} \, p \, s) \implies \text{healthy} \, (wp \, (a \, x)) \)
and wp: \( \forall s. \text{unitary} \, (p \, s) \)
and sump: \( \forall s. \text{sum} \, (p \, s) \, (\text{supp} \, (p \, s)) \leq 1 \)
and fsupp: \( \forall s. \text{finite} \, (\text{supp} \, (p \, s)) \)
shows bd-cts \((wp \, (\text{SetPC} \, a \, p))\)

proof –
from assms have bd-cts \((wp \, (\text{Bind} \, p \, (\lambda \, \cdot. \text{SetPC} \, a \, (\lambda \cdot. \, p))))\)
by (iprover intro:cts-wp-Bind cts-wp-SetPC-const)
thus ?thesis by (simp add:SetPC-Bind [symmetric])
qed

lemma wp-SetDC-Bind:
\( \text{SetDC} \, a \, s = \text{Bind} \, s \, (\lambda \cdot. \text{SetDC} \, a \, (\lambda \cdot. \, s)) \)
by (intro ext, simp add: SetDC-def Bind-def)

lemma SetDC-finite-insert:
assumes fS: \( \text{finite} \, S \)
and neS: \( S \neq \{\} \)
sows SetDC \( a \, (\lambda \cdot. \, \text{insert} \, x \, S) = a \, x \prod \, \text{SetDC} \, a \, (\lambda \cdot. \, S) \)
proof (intro ext, simp add: SetDC-def DC-def)
fix ab P s
from fS have A: \( \text{finite} \, (\text{insert} \, (a \, x \, ab \, P \, s) \, (\lambda \cdot. \, a \, x \, ab \, P \, s) \, \cdot \, S)) \)
and B: \( \text{finite} \, ((\lambda \cdot. \, a \, x \, ab \, P \, s) \, \cdot \, S) \) by (auto)
from neS have C: \( \text{insert} \, (a \, x \, ab \, P \, s) \, (\lambda \cdot. \, a \, x \, ab \, P \, s) \, \cdot \, S \neq \{\} \) by (auto)
and D: \( (\lambda \cdot. \, a \, x \, ab \, P \, s) \, \cdot \, S \neq \{\} \) by (auto)
from A C have Inf \((\text{insert} \, (a \, x \, ab \, P \, s) \, ((\lambda \cdot. \, a \, x \, ab \, P \, s) \, \cdot \, S)) =\)
\( \text{Min} \, (\text{insert} \, (a \, x \, ab \, P \, s) \, ((\lambda \cdot. \, a \, x \, ab \, P \, s) \, \cdot \, S)) \)
by (auto intro:cInf-eq-Min)
also from B D have \( .. = \text{min} \, (a \, x \, ab \, P \, s) \, (\text{Min} \, ((\lambda \cdot. \, a \, x \, ab \, P \, s) \, \cdot \, S)) \)
by (auto intro:Min-insert)
also from B D have \( .. = \text{min} \, (a \, x \, ab \, P \, s) \, (\text{Inf} \, ((\lambda \cdot. \, a \, x \, ab \, P \, s) \, \cdot \, S)) \)
by (simp add:cInf-eq-Min)
finally show \( (\text{INF} \, x:\text{insert} \, x \, S. \, a \, x \, ab \, P \, s) =\)
\( \text{min} \, (a \, x \, ab \, P \, s) \, (\text{INF} \, x:S. \, a \, x \, ab \, P \, s) \)
by (simp cong del: strong-INF-cong)
qed

lemma SetDC-singleton:
\( \text{SetDC} \, a \, (\lambda \cdot. \, \{x\}) = a \, x \)
by (simp add: SetDC-def cong del: strong-INF-cong)

lemma cts-wp-SetDC-const:
fixes a::\'a \Rightarrow \'s \text{ prog}
assumes ca: \( \forall x. \, x \in S \implies \text{bd-cts} \, (wp \, (a \, x)) \)
and ha: \( \forall x. \, x \in S \implies \text{healthy} \, (wp \, (a \, x)) \)
and fS: \( \text{finite} \, S \)
and neS: \( S \neq \{\} \)
sows bd-cts \((wp \, (\text{SetDC} \, a \, (\lambda \cdot. \, S)))\)
proof –
have \( \text{finite } S \implies S \neq \{\} \implies \)
\((\forall x \in S. \text{bd-cts} \ (wp \ (a \ x))) \implies \)
\((\forall x \in S. \text{healthy} \ (wp \ (a \ x))) \implies \)
\text{bd-cts} \ (wp \ (SetDC \ a \ (\lambda x. \ S)))

\text{proof} (\text{induct } S \ \text{rule: finite-induct, simp, clarsimp})

\text{fix } x :: 'a \text{ and } F :: 'a \text{ set}

\text{assume } \exists F: \text{finite } F

\text{and } \exists IH: F \neq \{\} \implies \text{bd-cts} \ (wp \ (SetDC \ a \ (\lambda x. \ F)))

\text{and } \exists cax: \text{bd-cts} \ (wp \ (a \ x))

\text{and } \exists hax: \text{healthy} \ (wp \ (a \ x))

\text{and } \exists haF: \forall x \in F. \text{healthy} \ (wp \ (a \ x))

\text{show } \text{bd-cts} \ (wp \ (SetDC \ a \ (\lambda x. \ \text{insert } x \ F)))

\text{proof} (\text{cases } F = \{\}, \text{ simp add: SetDC-singleton cax})

\text{assume } F \neq \{\}

\text{with } \exists fF \ cax \ hax \ haF \ IH \ \text{show } \text{bd-cts} \ (wp \ (SetDC \ a \ (\lambda x. \ \text{insert } x \ F)))

\text{by} (\text{auto intro! cts-wp-DC healthy-intros simp: SetDC-finite-insert})

\text{qed}

\text{with } \exists \text{assms } \text{show } ?\text{thesis } \text{by} (\text{auto})

\text{qed}

\text{lemma cts-wp-SetDC:}

\text{fixes } a :: 'a \Rightarrow 's \text{ prog}

\text{assumes } ca: \bigwedge x s. x \in S s \implies \text{bd-cts} \ (wp \ (a \ x))

\text{and } ha: \bigwedge x s. x \in S s \implies \text{healthy} \ (wp \ (a \ x))

\text{and } \exists S: \bigwedge s. \text{finite} \ (S s)

\text{and } \exists neS: \bigwedge s. S s \neq \{\}

\text{shows } \text{bd-cts} \ (wp \ (\text{SetDC} \ a \ S))

\text{proof –}

\text{from } \exists \text{assms } \text{have } \text{bd-cts} \ (wp \ (\text{Bind} \ S \ (\lambda S. \ \text{SetDC} \ a \ (\lambda x. \ S))))

\text{by (iprover intro!: cts-wp-Bind cts-wp-SetDC-const)}

\text{thus } ?\text{thesis } \text{by} (\text{simp add: wp-SetDC-Bind[symmetric]})

\text{qed}

\text{lemma cts-wp-repeat:}

\text{bd-cts} \ (wp \ a) \implies \text{healthy} \ (wp \ a) \implies \text{bd-cts} \ (wp \ (\text{repeat } n \ a))

\text{by (induct } n, \text{ auto intro!: cts-wp-Skip cts-wp-Seq healthy-intros)}

\text{lemma cts-wp-Embed:}

\text{bd-cts} \ t \implies \text{bd-cts} \ (wp \ (\text{Embed} \ t))

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

4.3.2 Continuity of a Single Loop Step

A single loop iteration is continuous, in the more general sense defined above
for transformer transformers.

\text{lemma cts-wp-loopstep:}

\text{fixes } body :: 's \text{ prog}

\text{assumes } \exists bb: \text{healthy} \ (wp \ body)
and cb: bd-cts (wp body)
shows bd-cts-tr (λx. wp (body ;; Embed x ∈ G ⊕ Skip)) (is bd-cts-tr ?F)
proof (rule bd-cts-trI, rule le-trans-antisym)
fix M :: nat ⇒ s trans and b :: real
assume chain: ∃i. le-trans (M i) (M (Suc i))
and fM: ∃i. feasible (M i)
show fu: le-trans (Sup-trans (range (?F o M))) (?F (Sup-trans (range M)))
proof (rule le-transI [OF Sup-trans-least2], clarsimp)
fix P :: s expect and t
assume sP: sound P
assume sQ: nneg Q and bP: bounded-by (bound-of P) Q
hence sQ: sound Q by (auto)

from fM have fSup: feasible (Sup-trans (range M))
  by (auto intro: feasible-Sup-trans)

from sQ fM have M t Q ⊢ Sup-trans (range M) Q
  by (auto intro: Sup-trans-upper2)
moreover from sQ fM fSup
have sMtP: sound (M t Q) sound (Sup-trans (range M) Q) by (auto)
ultimately have wp body (M t Q) ⊢ wp body (Sup-trans (range M) Q)
  using healthy-monoD [OF hb] by (auto)
    hence ∃s. wp body (M t Q) s ≤ wp body (Sup-trans (range M) Q) s
  by (rule le-funD)
thus ?F (M t) Q ⊢ ?F (Sup-trans (range M)) Q
  by (intro le-funI, simp add: wp-eval mult-left-mono)

show nneg (wp (body ;; Embed (Sup-trans (range M)) ⊕ Skip) Q)
proof (rule nnegI, simp add: wp-eval)
fix s :: s
  from fSup sQ have sound (Sup-trans (range M) Q) by (auto)
  with hb have sound (wp body (Sup-trans (range M) Q)) by (auto)
  hence 0 ≤ wp body (Sup-trans (range M) Q) s by (auto)
moreover from sQ have 0 ≤ Q s by (auto)
ultimately show 0 ≤ «G» s * wp body (Sup-trans (range M) Q) s + (I − «G» s) * Q s
  by (auto intro: add-nonneg-nonneg mult-nonneg-nonneg)
qed

next
fix P :: s expect assume sP: sound P
thus nneg P bounded-by (bound-of P) P by (auto)
show ∀ u ∈ range ((λx. wp (body ;; Embed x ∈ G ⊕ Skip)) o M).
  ∀ R. nneg R ∧ bounded-by (bound-of P) R R →
    nneg (u R) ∧ bounded-by (bound-of P) (u R)
proof (clarsimp, intro conjI nnegI bounded-byI, simp add: wp-eval)
fix w :: nat and R :: s expect and s :: s
assume nR: nneg R and bR: bounded-by (bound-of P) R
hence sR: sound R by (auto)
with fM have sM u R: sound (M u R) by (auto)
with \( \text{hb have sound \((wp \text{ body} (M u R))\)} \) \( \text{by(auto)} \)

hence \( 0 \leq wp \text{ body} (M u R) \) \( s \) \( \text{by(auto)} \)

moreover from \( nR \) have \( 0 \leq R \) \( s \) \( \text{by(auto)} \)

ultimately show \( 0 \leq \langle G \rangle s * wp \text{ body} (M u R) s + (1 - \langle G \rangle s) * R s \)
by(\( \text{auto intro:add-nonneg-nonneg mult-nonneg-nonneg} \))

from \( sR \) \( bR \) \( fM \) have bounded-by (bound-of \( P \)) \( M \) \( u \) \( R \) \( \text{by(auto)} \)

with \( sM u R \) \( \text{hb have bounded-by (bound-of \( P \)} \) \( (wp \text{ body} (M u R))\) \( \text{by(auto)} \)

hence \( wp \text{ body} (M u R) \) \( s \) \( \leq \) bound-of \( P \) \( \text{by(auto)} \)

moreover from \( bR \) \( \text{have R s} \) \( \leq \) bound-of \( P \) \( \text{by(auto)} \)

ultimately have \( \langle G \rangle s * wp \text{ body} (M u R) s + (1 - \langle G \rangle s) * R s \leq \)

by(\( \text{auto intro:} \text{add-mono mult-left-mono} \))

also have ... = \( \text{bound-of P by(simp add:algebra-simps)} \)

finally show \( \langle G \rangle s * wp \text{ body} (M u R) s + (1 - \langle G \rangle s) * R s \leq \) bound-of 

\( P \).

qed

qed

show le-trans (\( \langle F \rangle \) (Sup-trans (\( \text{range} M \))) (Sup-trans (\( \text{range} (\langle F o M \rangle) \)))

proof (rule le-transI, rule le-funI, simp add:wp-eval)
fix \( P:\)'s expect and \( s:\)'s
assume \( sP\); sound \( P\)

have \( \{ t \in \text{range} M \} = \text{range} (\lambda i. M i P) \)
by(blast)

hence \( wp \text{ body} (\text{Sup-trans (range} M P) s = wp \text{ body} (\text{Sup-exp (range} (\lambda i. M i P)) s \)
by(simp add:Sup-trans-def)

also \{
from \( sP \) \( fM \) have \( \lambda i. \text{sound} (M i P) \) \( \text{by(auto)} \)
moreover from \( sP \) \( \text{chain have} \( \lambda i. M i P + M \) (\( \text{Suc i} \) P) \( \text{by(auto)} \)
moreover \{
from \( sP \) have bounded-by (bound-of \( P \) \( M i P \) \( \text{by(auto)} \)
with \( sP \) \( fM \) have \( \lambda i. \text{bounded-by (bound-of \( P \)} \) \( M i P \) \( \text{by(auto)} \)
\}
ultimately have \( wp \text{ body} (\text{Sup-exp (range} (\lambda i. M i P)) s = \)
\( \text{Sup-exp (range} (\lambda i. wp \text{ body} (M i P)) s \)
by(subst bd-ctsD[OF cb], auto simp:o-def)
\}
also have \( \text{Sup-exp (range} (\lambda i. wp \text{ body} (M i P)) s = \)
\( \text{Sup} \{ f s \mid f. f \in \text{range} (\lambda i. wp \text{ body} (M i P)) \}
by(simp add:Sup-exp-def)
finally have \( \langle G \rangle s * wp \text{ body} (\text{Sup-trans (range} M P) s + (1 - \langle G \rangle s) * P \)
s = \( \langle G \rangle s * \text{Sup} \{ f s \mid f. f \in \text{range} (\lambda i. wp \text{ body} (M i P)) \} + (1 - \langle G \rangle 

s) * P s \)
by(simp)
also \{
from \( sP \) \( fM \) have \( \lambda i. \text{sound} (M i P) \) \( \text{by(auto)} \)
moreover from \( sP \land M \) have \( \forall i. \text{bounded-by} \ (\text{bound-of} P) \ (M \ i \ P) \) by(auto)
ultimately have \( \forall i. \text{bounded-by} \ (\text{bound-of} P) \ (wp \ \text{body} \ (M \ i \ P)) \) using \( hb \)
byp(auto)
hence \( \text{bound}: \forall i. \text{wp body} \ (M \ i \ P) \ s \leq \text{bound-of} \ P \) by(auto)
moreover have \( \{ \ « G » \ s * x \ | x. \ x \in \{ f s \ \mid f. \ f \in \text{range} \ (\lambda i. \text{wp body} \ (M \ i \ P))\}\} = \\
\{ « G » \ s * f s \ | f. \ f \in \text{range} \ (\lambda i. \text{wp body} \ (M \ i \ P))\}\}
by(blast)
ultimately have \( \{ « G » \ s * \text{Sup} \ \{ f s \ | f. \ f \in \text{range} \ (\lambda i. \text{wp body} \ (M \ i \ P))\}\} = \\
\text{Sup} \ \{ « G » \ s * f s \ | f. \ f \in \text{range} \ (\lambda i. \text{wp body} \ (M \ i \ P))\}\}
by(subst cSup-mult, auto)
moreover have \( \{ x + (1 - « G » \ s) * P \ s \ | x. \ x \in \{ « G » \ s * f s \ | f. \ f \in \text{range} \ (\lambda i. \text{wp body} \ (M \ i \ P))\}\} = \\
\{ « G » \ s * f s + (1 - « G » \ s) * P \ s \ | f. \ f \in \text{range} \ (\lambda i. \text{wp body} \ (M \ i \ P))\}\}
by(blast)
moreover from \( \text{bound} \ sP \) have \( \forall i. « G » \ s * \text{wp body} \ (M \ i \ P) \ s \leq \text{bound-of} \ P \)
byp(auto)
ultimately have \( \{ « G » \ s * \text{Sup} \ \{ f s \ | f. \ f \in \text{range} \ (\lambda i. \text{wp body} \ (M \ i \ P))\} + (1 - « G » \ s) * P \ s \} = \\
\text{Sup} \ \{ « G » \ s * f s + (1 - « G » \ s) * P \ s \ | f. \ f \in \text{range} \ (\lambda i. \text{wp body} \ (M \ i \ P))\}\}
by(subst cSup-add, auto)
}
ultimately have \( \{ « G » \ s * \text{Sup} \ \{ f s \ | f. \ f \in \text{range} \ (\lambda i. \text{wp body} \ (M \ i \ P))\} + (1 - « G » \ s) * P \ s \} = \\
\text{Sup} \ \{ « G » \ s * f s + (1 - « G » \ s) * P \ s \ | f. \ f \in \text{range} \ (\lambda i. \text{wp body} \ (M \ i \ P))\}\}
by(simp)
}
also \{ \n\text{have} \ \{ i. \ « G » \ s * \text{wp body} \ (M \ i \ P) \ s + (1 - « G » \ s) * P \ s = \\
(\lambda x . wp (\text{body} ;; \text{Embed} \ x \ a \ G \oplus \text{Skip}) \circ M) \ i \ P \ s \}
by(simp add:wp-eval)
\text{also have} \ \{ i. \ s \leq \text{Sup} \ \{ f s \ | f. \ f \in \{ t P \ | t. \ t \in \text{range} \ ((\lambda x . wp (\text{body} ;; \text{Embed} \ x \ a \ G \oplus \text{Skip}) \circ M))\}\}\}
proof(intro cSup-upper bdd-aboveI, blast, clarsimp simp:wp-eval)
fix \( i \)
from \( sP \) have \( bp: \text{bounded-by} \ (\text{bound-of} P) \ P \) by(auto)
with \( sP \land M \) have \( \text{sound} \ (M \ i \ P) \text{ bounded-by} \ (\text{bound-of} P) \ (M \ i \ P) \) by(auto)
with \( hb \) have \( \text{bounded-by} \ (\text{bound-of} P) \ (wp \ \text{body} \ (M \ i \ P)) \) by(auto)
with \( bP \) have \( \text{wp body} \ (M \ i \ P) \ s \leq \text{bound-of} \ P \) by(auto)
hence \( « G » \ s * \text{wp body} \ (M \ i \ P) \ s + (1 - « G » \ s) * P \ s \leq \\
« G » \ s * (\text{bound-of} P) + (1 - « G » \ s) * (\text{bound-of} P) \)
CHAPTER 4. THE PGCL LANGUAGE

by (auto intro: add-mono mult-left-mono)
also have ... = bound-of P by (simp add: algebra-simps)
finally show \( «G» s * \text{wp body} (M \ i \ P) \ s + (1-«G» s) \ s \leq \text{bound-of} \ P \).

by (auto intro: add-mono mult-left-mono)
also have \( \text{Sup} \ \{ «G» s * f s + (I-«G» s) \ s \ | \ f. \ f \in \text{range} \ ((\lambda x. \ \text{wp body} \ (M \ i \ P)) \}\) \leq 
\( \text{Sup} \ \{ f s \ | f. \ f \in \{ t P \ | t. \ t \in \text{range} \ ((\lambda x. \ \text{wp body} ; ; \text{Embed} \ x \ «G» \oplus \text{Skip})) \circ M)\}\)
by (blast intro: cSup-least)

finally have \( \text{Sup} \ \{ f s \ | f. \ f \in \{ t P \ | t. \ t \in \text{range} \ ((\lambda x. \ \text{wp body} ; ; \text{Embed} \ x \ «G» \oplus \text{Skip})) \circ M)\} = 
\text{Sup}-\text{trans} \ (\text{range} \ ((\lambda x. \ \text{wp body} ; ; \text{Embed} \ x \ «G» \oplus \text{Skip})) \circ M) \ P \ s 
by (simp add: Sup-trans-def Sup-exp-def)
finally show \( «G» s * \text{wp body} (\text{Sup}-\text{trans} \ (\text{range} \ M) \ P) \ s + (1-«G» s) \ s \leq 
\text{Sup}-\text{trans} \ (\text{range} \ ((\lambda x. \ \text{wp body} ; ; \text{Embed} \ x \ «G» \oplus \text{Skip})) \circ M) \ P \ s \).

qed

end

4.4 Continuity and Induction for Loops

theory LoopInduction imports Healthiness Continuity begin

Showing continuity for loops requires a stronger induction principle than we have used so far, which in turn relies on the continuity of loops (inductively). Thus, the proofs are intertwined, and broken off from the main set of continuity proofs. This result is also essential in showing the sublinearity of loops.

A loop step is monotonic.

lemma wp-loop-step-mono-trans:
fixes body::'s prog
assumes sP: sound P
and hb: healthy (wp body)
shows mono-trans (\( \lambda Q. \ «G» s * \text{wp body} Q \ s + «N \ G» s * P \ s \))
proof (intro mono-transI le-funI, simp)
fix Q R::'s expect and s::'s
assume sQ: sound Q and sR: sound R and le: Q \leq R
hence wp body Q \vdash wp body R
by (rule mono-transD[OF healthy-monoD, OF hb])
thus \( «G» s * \text{wp body} Q \ s \leq «G» s * \text{wp body} R \ s 
by (auto dest: le-funD intro: mult-left-mono)
We can therefore apply the standard fixed-point lemmas to unfold it:

**lemma lfp-wp-loop-unfold:**

**fixes** body:‘s prog

**assumes** hb: healthy (wp body)

**and** sP: sound P

**shows** lfp-exp (\(\lambda Q s. \langle G \rangle s * wp body Q s + \langle N G \rangle s * P s\)) =

(\(\lambda s. \langle G \rangle s * wp body (lfp-exp (\lambda Q s. \langle G \rangle s * wp body Q s + \langle N G \rangle s * P s)) s + \langle N G \rangle s * P s\))

**proof** (rule lfp-exp-unfold)

from assms show mono-trans (\(\lambda Q s. \langle G \rangle s * wp body Q s + \langle N G \rangle s * P s\))

by (blast intro:wp-loop-step-mono-trans)

from sP show sound (\(\lambda s. \langle G \rangle s * wp body (\lambda s. bound-of P) s + \langle N G \rangle s * P s\))

by (auto)

fix Q:‘s expect

assume sound Q

with assms show sound (\(\lambda s. \langle G \rangle s * wp body Q s + \langle N G \rangle s * P s\))

by (intro wp-loop-step-sound [unfolded wp-eval, simplified, folded negate-embed], auto)

qed

**lemma wp-loop-step-unitary:**

**fixes** body:‘s prog

**assumes** hb: healthy (wp body)

**and** uP: unitary P and uQ: unitary Q

**shows** unitary (\(\lambda s. \langle G \rangle s * wp body Q s + \langle N G \rangle s * P s\))

**proof** (intro unitaryI2 nnegI bounded-byI)

fix s:‘s

from uQ hb have uwQ: unitary (wp body Q) by (auto)

with uP have 0 ≤ wp body Q s 0 ≤ P s by (auto)

thus 0 ≤ \(\langle G \rangle s * wp body Q s + \langle N G \rangle s * P s\)

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

from uP uwQ have wp body Q s ≤ 1 P s ≤ 1 by (auto)

hence \(\langle G \rangle s * wp body Q s + \langle N G \rangle s * P s ≤ \langle G \rangle s * 1 + \langle N G \rangle s * 1\)

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

also have ... = 1 by (simp add: negate-embed)

finally show \(\langle G \rangle s * wp body Q s + \langle N G \rangle s * P s ≤ 1\).

qed

**lemma lfp-loop-unitary:**

**fixes** body:‘s prog

**assumes** hb: healthy (wp body)

**and** uP: unitary P
shows unitary (lfp-exp (λ Q s. «G» s * wp body Q s + «N» G s * P s))
using assms by (blast intro:lfp-exp-unitary wp-loop-step-unitary)

From the lattice structure on transformers, we establish a transfinite induction principle for loops. We use this to show a number of properties, particularly subdistributivity, for loops. This proof follows the pattern of lemma lfp_ordinal_induct in HOL/Inductive.

lemma loop-induct:
fixes body :: ′s prog
assumes hwp: healthy (wp body)
and hwlp: nearly-healthy (wlp body)
— The body must be healthy, both in strict and liberal semantics.
and Limit: ∀ S. [ [ ∀ x∈S. P (fst x) (snd x); ∀ x∈S. feasible (fst x); ∀ x∈S. ∀ Q. unitary Q −→ unitary (snd x Q) ] ] ⇒ P (Sup-trans (fst ′ S)) (Inf-utrans (snd ′ S))
— The property holds at limit points.
and IH: ∀ t u. [ [ P t u; feasible t; ∀ Q. unitary Q =⇒ unitary (u Q) ] ] ⇒ P t u′
— The inductive step. The property is preserved by a single loop iteration.
and P-equiv: ∀ t t ′ u u ′. [ [ P t u; equiv-trans t t ′; equiv-utrans u u ′ ] ] ⇒ P t ′ u
— The property must be preserved by equivalence
shows P (wp (do G −→ body od)) (wlp (do G −→ body od))
— The property can refer to both interpretations simultaneously. The unifier will happily apply the rule to just one or the other, however.

proof (simp add:wp-eval)
let ?X t = wp (body ;; Embed t «G» ⊕ Skip)
let ?Y t = wlp (body ;; Embed t «G» ⊕ Skip)

let ?M = { x. P (fst x) (snd x) ∧
  feasible (fst x) ∧
  (∀ Q. unitary Q −→ unitary (snd x Q)) ∧
  le-trans (fst x) (lfp-trans ?X) ∧
  le-utrans (gfp-trans ?Y) (snd x) }

have fSup: feasible (Sup-trans (fst ′ ?M))
proof (intro feasibleI bounded-byI2 megI2)
fix Q::′s expect and b::real
assume nQ: nneg Q and bQ: bounded-by b Q
show Sup-trans (fst ′ ?M) Q ⊢ λs. b
  unfolding Sup-trans-def
  using nQ bQ by (auto intro!:Sup-exp-least)
show λs. 0 ⊢ Sup-trans (fst ′ ?M) Q
proof (cases)
  assume empty: ?M = {}
  show ?thesis by (simp add:Sup-trans-def Sup-exp-def empty)
next
  assume ne: ?M ≠ {}

then obtain $x$ where $xin: x \in ?M$ by auto  

hence $ffx$: feasible $(fst x)$ by(simp)  

with $nQ \ bQ$ have $\lambda s. \ 0 \vdash \ \cdot x \ Q$ by(auto)  

also from $xin$ have $fst x \ Q \vdash \ Sup-trans (fst \ ?M) \ Q$  

apply(intro Sup-trans-upper2[OF imageI - $nQ \ bQ$], assumption)  

apply(clarsimp, blast intro: sound-nneg[OF feasible-sound feasible-boundedD])  

done  

finally show $\lambda s. \ 0 \vdash \ Sup-trans (fst \ ?M) \ Q$ .  

qed
unfolding \( \text{Inf-utrans-def} \) by \( \text{subst } X, \text{ simp} \)

\}

finally show \( \lambda s. 0 \vdash \text{Inf-utrans} (\text{snd } ?M) P \)

qed

qed

have \( \text{wp-loop-mono} : \forall t u. [\text{le-trans } t u; \forall P. \text{sound } P \implies \text{sound } (t P); \\forall P. \text{sound } P \implies \text{sound } (u P)] \implies \text{le-trans } (?X t) (?X u) \)

proof (intro \text{le-transI} \text{ le-funI}, \text{ simp add: wp-eval})

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

assume le: \( \text{le-trans } t u \)

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 \( \text{healthy-monoD } [\text{OF hw} \text{p}] \) le sP have \( \text{wp body } (t P) \vdash \text{wp body } (u P) \)

by (auto)

hence \( \text{wp body } (t P) s \leq \text{wp body } (u P) s \) by (auto)

thus \( <G> s * \text{wp body } (t P) s \leq <G> s * \text{wp body } (u P) s \) by (auto intro: \text{mult-left-mono})

qed

have \( \text{wlp-loop-mono} : \forall t u. [\text{le-utrans } t u; \forall P. \text{unitary } P \implies \text{unitary } (t P); \\forall P. \text{unitary } P \implies \text{unitary } (u P)] \implies \text{le-utrans } (?Y t) (?Y u) \)

proof (intro \text{le-utransI} \text{ le-funI}, \text{ simp add: wp-eval})

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

assume le: \( \text{le-trans } t u \)

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 \( \text{le uP have } \text{wlp body } (t P) \vdash \text{wlp body } (u P) \)

by (auto intro: nearly-\text{healthy-monoD } [\text{OF hwlp}])

hence \( \text{wlp body } (t P) s \leq \text{wlp body } (u P) s \) by (auto)

thus \( <G> s * \text{wlp body } (t P) s \leq <G> s * \text{wlp body } (u P) s \)

by (auto intro: \text{mult-left-mono})

qed

from \( \text{hw} \text{p} \) have \( hX: \forall t. \text{healthy } t \implies \text{healthy } (?X t) \)

by (auto intro: \text{healthy-intros})

from \( \text{hwlp} \) have \( hY: \forall t. \text{nearly-healthy } t \implies \text{nearly-healthy } (?Y t) \)

by (auto intro: \text{healthy-intros})

have \( \text{PLimit}: P (\text{Sup-trans } (\text{fst } ?M)) (\text{Inf-utrans } (\text{snd } ?M)) \)

by (auto intro: \text{Limit})

have \( \text{feasible-lfp-loop} : \)
4.4. CONTINUITY AND INDUCTION FOR LOOPS

feasible (lfp-trans \( ?X \))

\[ \text{proof}(\text{intro feasibleI bounded-byI2 nnegI2,}
\quad \text{simp-all add:wp-LoopI [simplified wp-eval] soundI2 hwp}) \]

\[ \text{fix } P::'s \text{ expect and } b::\text{real} \]

\[ \text{assume } bP: \text{ bounded-by } b \text{ P and } nP: \text{ nneg } P \]

\[ \text{hence } sP: \text{ sound } P \text{ by(auto)} \]

\[ \text{show } \text{lfp-exp} (\lambda Q \text{ s.} \ « \ G » \ s * \ wp \ \text{body} \ Q \ s + \ « \ N \ G » \ s * P \ s) \nleq \ \lambda . \ s. \ b \]

\[ \text{proof}(\text{intro lfp-exp-lowerbound le-funI}) \]

\[ \text{fix } s::'s \]

\[ \text{from } bP \ nP \ \text{have nnb: } 0 \leq b \text{ by(auto)} \]

\[ \text{hence } \text{sound} (\lambda s. b) \text{ bounded-by } b \text{ (} \lambda s. b \text{) by(auto)} \]

\[ \text{with } \text{hwp have bounded-by } b \text{ (wp body (} \lambda s. b)) \text{ by(auto)} \]

\[ \text{with } bP \ \text{have wp body (} \lambda s. b) \leq b \text{ P s \leq b by(auto)} \]

\[ \text{hence } \ « G » \ s * \ wp \ \text{body} \ (\lambda s. b) \ s + \ « N \ G » \ s * P \ s \leq \ « G » \ s * b \]

\[ \text{by(auto intro:add-mono mult-left-mono)} \]

\[ \text{thus } \ « G » \ s * \ wp \ \text{body} \ (\lambda s. b) \ s + \ « N \ G » \ s \leq b \]

\[ \text{by(simp add:negate-embed algebra-simps)} \]

\[ \text{from nnb show sound} (\lambda s. b) \text{ by(auto)} \]

\[ \text{qed} \]

\[ \text{from } \text{hwp sP show} \ \lambda s. 0 \nleq \text{lfp-exp} (\lambda Q \text{ s.} \ « \ G » \ s * \ wp \ \text{body} \ Q \ s + \ « \ N \ G » \ s * P \ s) \]

\[ \text{by(blast intro!:lfp-exp-greatest lfp-loop-fp)} \]

\[ \text{qed} \]

\[ \text{have unitary-gfp:} \]

\[ \bigwedge P. \ \text{unitary } P \implies \text{unitary (gfp-trans } ?Y \ P \big) \]

\[ \text{proof}(\text{intro unitaryI2 nnegI2 bounded-byI2,}
\quad \text{simp-all add:wlp-LoopI [simplified wp-eval] hwp}) \]

\[ \text{fix } P::'s \text{ expect} \]

\[ \text{assume } uP: \text{ unitary } P \]

\[ \text{show } \lambda s. 0 \nleq \text{gfp-exp} (\lambda Q \text{ s.} \ « \ G » \ s * \ wp \ \text{body} \ Q \ s + \ « \ N \ G » \ s * P \ s) \]

\[ \text{proof(rule gfp-exp-upperbound[OF le-funI])} \]

\[ \text{fix } s::'s \]

\[ \text{from } \text{hwp uP have } 0 \leq \text{wp body} \ (\lambda s. 0) \ s 0 \leq P \ s \text{ by(auto dest!:unitary-sound)} \]

\[ \text{thus } 0 \nleq \ « G » \ s * \ wp \ \text{body} \ (\lambda s. 0) \ s + \ « N \ G » \ s * P \ s \]

\[ \text{by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)} \]

\[ \text{show unitary} (\lambda s. 0) \text{ by(auto)} \]

\[ \text{qed} \]

\[ \text{show } \text{gfp-exp} (\lambda Q \text{ s.} \ « \ G » \ s * \ wp \ \text{body} \ Q \ s + \ « \ N \ G » \ s * P \ s) \nleq \ \lambda s. 1 \]

\[ \text{by(auto intro:gfp-exp-least)} \]

\[ \text{qed} \]

\[ \text{have } fX: \]

\[ \bigwedge t. \ \text{feasible } t \implies \text{feasible (} ?X \ t \big) \]

\[ \text{proof}(\text{intro feasibleI nnegI bounded-byI, simp-all add:wp-eval}) \]

\[ \text{fix } t::'s \text{ trans and } Q::'s \text{ expect and } b::\text{real and } s::'s \]

\[ \text{assume } f:\text{ feasible } t \text{ and } bQ: \text{ bounded-by } b \ Q \text{ and } nQ: \text{ nneg } Q \]

\[ \text{hence } \text{nneg} (t \ Q) \text{ bounded-by } b \ (t \ Q) \text{ by(auto)} \]
moreover hence \( stQ \): sound \((tQ)\) by\(\text{auto}\)
ultimately have \( wp\ \text{body}\ \((tQ)\) \(s \leq b\) using \(\text{hwp}\) by\(\text{auto}\)
moreover from \( bQ \) have \( Q \ s \leq b \) by\(\text{auto}\)
ultimately have \( G \ s * wp\ \text{body}\ \((tQ)\) \(s + (1 - G s) * Q s \leq\
\(G s * b + (I - G s) * b\)
by\(\text{auto intro:}\text{add-mono}\ \text{mult-left-mono}\)
thus \( G s * wp\ \text{body}\ \((tQ)\) \(s + (1 - G s) * Q s \leq b\)
by\(\text{simp add:algebra-simps}\)

from \( nQ \) \( stQ \) \( hwp\) have \( 0 \leq wp\ \text{body}\ \((tQ)\) \(s 0 \leq Q s\) by\(\text{auto}\)
thus \( 0 \leq G s * wp\ \text{body}\ \((tQ)\) \(s + (1 - G s) * Q s\)
by\(\text{auto intro:}\text{add-nonneg-nonneg}\ \text{mult-nonneg-nonneg}\)

\text{qed}

have \( uY:\)
\[\forall t \ \((\forall P. \ \text{unitary}\ P \implies \text{unitary}\ \((tP))\) \implies \text{unitary}\ P \implies \text{unitary}\ \((?Y tP))\)
proof\(\text{(intro unitaryI2 nnegI bounded-byI simp-all add:wp-eval)}\)
fix t::\(s\)\text{ trans and P::} \(s\)\text{'s expect and s::} \(s\)
assume u: \(\forall P. \ \text{unitary}\ P \implies \text{unitary}\ \((tP))\)
and uP: \(\text{unitary}\ P\)
hence uP: \(\text{unitary}\ \((tP)\) by\(\text{auto}\)
with \(\text{hwp}\) have ubtP: \(\text{unitary}\ \((wp\ \text{body}\ \((tP))\) by\(\text{auto}\)
with uP have \(0 \leq P s 0 \leq \text{wp\ \text{body}\ \((tP)\) s by\(\text{auto}\)
thus \(0 \leq G s * \text{wp\ \text{body}\ \((tP)\) s + (1 - G s) * Q s\)
by\(\text{auto intro:}\text{add-nonneg-nonneg}\ \text{mult-nonneg-nonneg}\)

from uP ubtP have \(P s \leq 1 \text{ wp\ \text{body}\ \((tP)\) s \leq 1}\) by\(\text{auto}\)
hence \(G s * \text{ wp\ \text{body}\ \((tP)\) s + (1 - G s) * P s \leq G s * 1 + (1 - G s) s * 1}\)
by\(\text{blast intro:}\text{add-mono}\ \text{mult-left-mono}\)
also have \(\ldots = 1\) by\(\text{(simp add:algebra-simps)}\)
finally show \(G s * \text{ wp\ \text{body}\ \((tP)\) s + (1 - G s) * P s \leq 1}\)
\text{qed}

have fu-lfp: \(\text{le\-trans}\ \((\text{Sup\-trans}\ (\text{fst} \ ?M))\) \((\text{lfp\-trans} \ ?X)\)
using \(\text{feasible-nnegD}[\text{OF feasible-lfp-loop}]\)
by\(\text{(intro le\-transI[\text{OF Sup\-trans\-least2}] blast+)}\)
hence \(\text{le\-trans}\ \((?X (\text{Sup\-trans}\ (\text{fst} \ ?M)))\) \((?X (\text{lfp\-trans} \ ?X))\)
by\(\text{(auto intro:wp\-loop\-mono feasible\-sound}[\text{OF fSup}]\)
\text{feasible\-sound}[\text{OF feasible-lfp-loop}]\)
also have \(\text{equiv\-trans}\ \ldots\) \((\text{lfp\-trans} \ ?X)\)
proof\(\text{(rule iffD1[\text{OF equiv\-trans\-comm}, OF lfp\-trans\-unfold], iprove intro:wp\-loop\-mono)}\)
fix t::\(\text{'s trans and P::} \(s\)\text{'s expect}\)
assume st: \(\text{\(\forall Q.\) sound}\ Q \implies \text{sound}\ \((tQ))\)
and sP: \(\text{sound}\ P\)
show sound \(\text{?X tP}\)
proof\(\text{(intro soundI2 bounded-byI nnegI simp-all add:wp-eval)}\)
fix s::\(s\)
from sP st hwp have \(0 \leq P s 0 \leq \text{wp\ \text{body}\ \((tP)\) s by\(\text{auto}\)
thus $0 \leq \langle G \rangle s \ast \text{wp body} (tP) s + (1 - \langle G \rangle s) \ast P s$

by (blast intro:add-nonneg-nonneg mult-nonneg-nonneg)

from $sP$ st have bounded-by (bound-of (tP)) (tP) by (auto)

with $sP$ st hwp have bounded-by (bound-of (tP)) (wp body (tP)) by (auto)

hence wp body (tP) $s \leq$ bound-of (tP) by (auto)

moreover from $sP$ st hwp have $P s \leq$ bound-of $P$ by (auto)

moreover have $\langle G \rangle s \leq 1 \ast 1 - \langle G \rangle s \leq 1$ by (auto)

moreover from $sP$ st hwp have $0 \leq$ wp body (tP) $s \leq P s$ by (auto)

moreover have $(0 :: \text{real}) \leq 1$ by (simp)

ultimately show $\langle G \rangle s \ast \text{wp body} (tP) s + (1 - \langle G \rangle s) \ast P s \leq 1 \ast \text{bound-of} (tP) + 1 \ast \text{bound-of} P$

by (blast intro:add-mono mult-mono)

qed

next

let $?fp = \lambda R s. \text{bound-of} R$

show le-trans $?X$ $?fp$ 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 $?X (\text{Sup-trans} (fst' ?M)) (\text{lfp-trans} ?X)$.

have fu-gfp: le-trans (gfp-trans $?Y$) (Inf-utrans (snd' ?M))

by (auto intro:Inf-utrans-greatest unitary-gfp)

have equiv-utrans (gfp-trans $?Y$) (?Y (gfp-trans $?Y$))

by (auto intro!:gfp-trans-unfold wlp-loop-mono uY)

also from fu-gfp have le-utrans (?Y (gfp-trans $?Y$)) (?Y (Inf-utrans (snd' ?M)))

by (auto intro:wlp-loop-mono uInf unitary-gfp)

finally have ge-gfp: le-utrans (gfp-trans $?Y$) (?Y (Inf-utrans (snd' ?M)))

from PLimit [X uY] Sup uInf have $P (?X (\text{Sup-trans} (fst' ?M))) (?Y (\text{Inf-utrans} (snd' ?M)))$

by (iprover intro:IH)

moreover from $\text{Sup}$ have feasible $?X (\text{Sup-trans} (fst' ?M))$ by (rule $fX$

moreover have $\bigwedge P. \text{unitary} P \implies \text{unitary} (?Y (\text{Inf-utrans} (snd' ?M))) P$

by (auto intro:uY uInf)

moreover note le-lfp ge-gfp

ultimately have pair-in: $?X (\text{Sup-trans} (fst' ?M)), ?Y (\text{Inf-utrans} (snd' ?M)) \in ?M$

by (simp)

have $?X (\text{Sup-trans} (fst' ?M)) \in \text{fst' ?M}$

by (rule imageI[OF pair-in, of $fX$, simplified])

hence le-trans $?X (\text{Sup-trans} (fst' ?M)) (\text{Sup-trans} (fst' ?M))$

proof (rule le-transI[OF Sup-trans-upper2] where $t = ?X (\text{Sup-trans} (fst' ?M))$

and $S = \text{fst' ?M}]])

fix $P :: s$ expect

assume $sP$: sound $P$

thus nneg $P$ by (auto)
from sP show bounded-by (bound-of P) P by(auto) 
from sP show \(\forall u \in \text{fst} \cdot \text{?M}. \forall Q. \neg\neg Q \land \text{bounded-by} (\text{bound-of P}) Q \rightarrow \neg\neg (u Q) \land \text{bounded-by} (\text{bound-of P}) (u Q)\) 
  by(auto) 
qed 

hence le-trans (\text{lfp-trans} ?X) (\text{Sup-trans} (\text{fst} \cdot \text{?M})) 
  by(auto intro:lfp-trans-lowerbound feasible-sound[OF fSup]) 

with fu-lfp have eqt: equiv-trans (\text{Sup-trans} (\text{fst} \cdot \text{?M})) (\text{lfp-trans} ?X) 
  by(rule le-trans-antisym) 

hence \text{le-utrans} (\text{Inf-utrans} (\text{snd} \cdot \text{?M})) \in \text{snd} \cdot \text{?M} 
  by(rule imageI[OF pair-in, of snd, simplified]) 

hence le-utrans (\text{Inf-utrans} (\text{snd} \cdot \text{?M})) (\text{?Y} (\text{Inf-utrans} (\text{snd} \cdot \text{?M}))) 
  by(intro Inf-utrans-lower, auto) 

hence le-utrans (\text{Inf-utrans} (\text{snd} \cdot \text{?M})) (\text{gfp-trans} ?Y) 
  by(blast intro: gfp-trans-upperbound uInf) 

with fu-gfp have equ: equiv-utrans (\text{Inf-utrans} (\text{snd} \cdot \text{?M})) (\text{gfp-trans} ?Y) 
  by(auto intro:le-utrans-antisym) 

from PLimit eqt eqt 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 = ((λx. wp (body ;; Embed x « G ⊕ Skip)) ^^ i) (λP s. 0) 

lemma iterates-0[simp]: 
  iterates body G 0 = (λP s. 0) 
  by(simp add:iterates-def) 

lemma iterates-Suc[simp]: 
  iterates body G (Suc i) = wp (body ;; Embed (iterates body G i) « Gs ⊕ Skip) 
  by(simp add:iterates-def) 

All iterates are healthy.

lemma iterates-healthy: 
  healthy (wp body) ⇒ healthy (iterates body G i) 
  by(induct i, auto intro:healthy-intros) 

The iterates are an ascending chain.

lemma iterates-increasing: 
  fixes body::'s prog
  assumes hb: healthy (wp body)
shows le-trans (iterates body G i) (iterates body G (Suc i))
proof(induct i)
show le-trans (iterates body G 0) (iterates body G (Suc 0))
proof(simp add: iterates-def, rule le-transI)
  fix P::'s expect
  assume sound P
  with hb have sound (wp (body ;; Embed (λP s. 0) « G » ⊕ Skip) P)
    by(auto intro:wp-loop-step-sound)
  thus λs. 0 ⊢ wp (body ;; Embed (λP s. 0) « G » ⊕ Skip) P
    by(auto)
qed

fix i
assume IH: le-trans (iterates body G i) (iterates body G (Suc i))
have equiv-trans (iterates body G (Suc i))
  (wp (body ;; Embed (iterates body G i) « G » ⊕ Skip))
  by(simp)
also from iterates-healthy[OF hb]
have le-trans ... (wp (body ;; Embed (iterates body G (Suc i)) « G » ⊕ Skip))
  by(blast intro:wp-loop-step-mono[OF hb IH])
also have equiv-trans ...
  (iterates body G (Suc (Suc i)))
  by(simp)
finally show le-trans (iterates body G (Suc i)) (iterates body G (Suc (Suc i)))
  .
qed

lemma wp-loop-step-bounded:
  fixes t::'s trans and Q::'s expect
  assumes nQ: nneg Q
  and bQ: bounded-by b Q
  and ht: healthy t
  and hb: healthy (wp body)
  shows bounded-by b (wp (body ;; Embed t « G » ⊕ Skip) Q)
proof(rule bounded-byI, simp add: wp-eval)
  fix s::'
  from nQ bQ have sQ: sound Q by(auto)
  with bQ ht have sound (t Q) bounded-by b (t Q) by(auto)
  with hb have bounded-by b (wp body (t Q)) by(auto)
  with bQ have wp body (t Q) s ≤ b Q s ≤ b by(auto)
  hence «G» s * wp body (t Q) s + (1-«G» s) * Q s ≤
    «G» s * b + (1-«G» s) * b
    by(auto intro:add-mono mult-left-mono)
  also have ... = b by(simp add: algebra-simps)
  finally show «G» s * wp body (t Q) s + (1-«G» s) * Q s ≤ b .
qed

This is the key result: The loop is equivalent to the supremum of its iterates.
This proof follows the pattern of lemma continuous__lfp in HOL/Library/Continuity.

lemma lfp-iterates:
  fixes body::'s prog
assumes \( hb: \text{healthy (wp body)} \)
and \( cb: \text{bd-cts (wp body)} \)
shows \( \text{equiv-trans (wp (do G \rightarrow body od)) (Sup-trans (range (iterates body G)))} \)
(is equiv-trans ?X ?Y)
proof (rule le-trans-antisym)
let \( ?F = \lambda x. \text{wp (body ;; Embed x } G \oplus \text{ Skip)} \)
let \( ?bot = \lambda (P ::'s \Rightarrow \text{real}) s.'s. 0::\text{real} \)

have \( HF: \bigwedge i. \text{healthy ((?F } ^{< i} ) ?bot)} \)
proof
  fix \( i \) from \( hb \) show (?thesis \( i \))
  by (induct \( i \), simp-all add: healthy-intros)
qed

from \( \text{iterates-healthy[OF } hb \)\]
have \( \bigwedge i. \text{feasible (iterates body G } i \))
by (auto)
hence \( \text{Sup: feasible (Sup-trans (range (iterates body G))}) \)
by (auto intro: feasible-Sup-trans)

\[
\begin{align*}
\text{fix } i \\
\text{have } \text{le-trans ((?F } ^{< i} ) ?bot) ?X \text{ )}
\end{align*}
\]
proof (induct \( i \))
show \( \text{le-trans ((?F } ^{< 0} ) ?bot) ?X \)
proof (simp, intro le-transI)
fix \( P::'s \) expect
with \( hb \) healthy-wp-loop
have \( \text{sound (wp (\mu x. \text{body ;; Embed } (\lambda P s. \text{bound-of } P) « G \oplus \text{ Skip}) P)} \)
by (auto)
thus \( \lambda s. 0 \vdash \text{wp (\mu x. \text{body ;; x } « G \oplus \text{ Skip}) P}) \)
by (auto)
qed

fix \( i \)
assume \( IH: \text{le-trans ((?F } ^{< i} ) ?bot) ?X \)
have \( \text{equiv-trans ((?F } ^{< (Suc i)} ) ?bot) (?F ((?F } ^{< i} ) ?bot)) \text{ by (simp)} \)
also have \( \text{le-trans ... (?F } ?X) \)
proof (rule wp-loop-step-mono[OF \( hb \ IH \)])
fix \( P::'s \) expect
assume \( sP: \text{sound } P \)
with \( hb \) healthy-wp-loop
show \( \text{sound (wp (\mu x. \text{body ;; x } « G \oplus \text{ Skip}) P}) \)
by (auto)
from \( sP \) show \( \text{sound ((?F } ^{< i} ) ?bot P}) \)
by (rule healthy-sound[OF \( HF \)])
qed
also { from \( hb \) have \( X: \text{le-trans (wp (body ;; Embed (\lambda P s. \text{bound-of } P) « G \oplus \text{ Skip}) P}) \) }
4.4. CONTINUITY AND INDUCTION FOR LOOPS

\[(\lambda P. s. \text{bound-of } P)\]
\[\text{by (intro le-transI, simp add: wp-eval, auto intro: lfp-loop-fp [unfolded negate-embed])}\]
\[\text{have equiv-trans (?F ?X) ?X}\]
\[\text{apply (simp only: wp-eval)}\]
\[\text{have equiv-trans (?F ?X) ?X}\]
\[\text{apply (simp only: wp-eval)}\]
\[\text{by (intro iffD1 [OF equiv-trans-comm, OF lfp-trans-unfold])}\]
\[\text{by (intro le-trans [Suc i] bot ?X). qed}\]

\[\text{finally show le-trans ((?F ^ suc i) bot) ?X. qed}\]
\[\text{hence } \forall i. \text{le-trans (iterates body } G i) (wp \text{ do } G \rightarrow \text{ body } od)\]
\[\text{thus le-trans ?Y ?X}\]

\[\text{show le-trans ?X ?Y}\]
\[\text{proof (simp only: wp-eval, rule lfp-trans-lowerbound)}\]
\[\text{from hb cb have bd-cts-tr ?F by (rule cts-wp-loopstep)}\]
\[\text{with iterates-increasing [OF hb] iterates-healthy [OF hb]}\]
\[\text{have equiv-trans (?F ?Y) (Sup-trans (range (?F o (iterates body } G))}}\]
\[\text{by (auto intro: healthy-feasibleD bd-cts-trD)}\]
\[\text{also have le-trans (Sup-trans (range (?F o (iterates body } G))) ?Y}\]
\[\text{proof (rule le-transI)}\]
\[\text{fix } P::'s \text{ expect}\]
\[\text{assume } sP::\text{ sound } P\]
\[\text{show (Sup-trans (range (?F o (iterates body } G))) P \rightarrow ?Y P}\]
\[\text{proof (rule Sup-trans-least2, clarsimp)}\]
\[\text{show } \forall u \in \text{range (}(\lambda x. \text{ wp (body } ; \text{ Embed } x \quad G \oplus \text{ Skip})) \circ \text{ iterates body } G).\]
\[\forall R. \text{ nneg } R \land \text{ bounded-by (bound-of } P) R \rightarrow \text{ nneg (u R) \land bounded-by (bound-of } P) (u R)\]
\[\text{proof (clarsimp, intro conjI)}\]
\[\text{fix } Q::'s \text{ expect and } i\]
\[\text{assume } nQ::\text{ nneg } Q \land bQ::\text{ bounded-by (bound-of } P) Q\]
\[\text{hence sound } Q \text{ by (auto)}\]
\[\text{moreover from iterates-healthy [OF hb]}\]
\[\text{have } \forall P. \text{ sound } P \rightarrow \text{ sound (iterates body } G i P) \text{ by (auto)}\]
\[\text{moreover note hb}\]
\[\text{ultimately have sound (wp (body } ; \text{ Embed (iterates body } G i) : G \oplus \text{ Skip}) Q)\]
\[\text{by (iprover intro: wp-loop-step-sound)}\]
\[\text{thus nneg (wp (body ; Embed (iterates body } G i) \quad G \oplus \text{ Skip}) Q)\]
\[\text{by (auto)}\]
\[\text{from nQ bQ iterates-healthy [OF hb] hb}\]
Therefore, evaluated at a given point (state), the sequence of iterates gives a sequence of real values that converges on that of the loop itself.

**corollary** loop-iterates:

```plaintext
fixes body::'s prog
assumes hb: healthy (wp body)
    and cb: bd-cts (wp body)
    and sP: sound P
shows (λi. iterates body G i s) →→ wp (do G →→ body od) P s
proof
  let ?X = {f s | f. f ∈ {t P | t. t ∈ range (iterates body G)}}
```
have closure-Sup: Sup ?X ∈ closure ?X
proof (rule closure-contains-Sup, simp, clarsimp)
  fix i
  from sP have bounded-by (bound-of P) P by (auto)
  with iterates-healthy[of hb] sP have \( \bigwedge j. \) bounded-by (bound-of P) (iterates body G j P)
    by (auto)
  thus iterates body G i P s ≤ bound-of P by (auto)
qed

have (\( \lambda i. \) iterates body G i P s) ----> Sup \{ f s \mid f ∈ \{ t P \mid t ∈ range (iterates body G) \} \}
proof (rule LIMSEQ-I)
  fix r :: real
  assume posr: 0 < r
  with closure-Sup obtain y where yin: y ∈ ?X and ey: dist y (Sup ?X) < r
    by (simp only: closure-approachable, blast)
  from yin obtain i where yit: y = iterates body G i P s by (auto)
  { 
    fix j
    have i ≤ j ----> le-trans (iterates body G i) (iterates body G j)
      proof (induct j, simp, clarify)
        fix k
        assume IH: i ≤ k ----> le-trans (iterates body G i) (iterates body G k)
        and le: i ≤ Suc k
        show le-trans (iterates body G i) (iterates body G (Suc k))
          proof (cases i = Suc k, simp)
            assume i ≠ Suc k
            with le have i ≤ 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
      qed
  }
  with sP have \( \forall j ≥ i. \) iterates body G i P s ≤ iterates body G j P s
    by (auto)
  moreover {
    from sP have bounded-by (bound-of P) P by (auto)
    with iterates-healthy[of hb] sP have \( \bigwedge j. \) bounded-by (bound-of P) (iterates body G j P)
      by (auto)
    hence \( \bigwedge j. \) iterates body G j P s ≤ bound-of P by (auto)
    hence \( \bigwedge j. \) iterates body G j P s ≤ Sup ?X
      by (intro cSup-upper bdd-aboveI, auto)
  }
  ultimately have \( \bigwedge j. \) i ≤ j ----> 
    norm (iterates body G j P s - Sup ?X) ≤ 
    norm (iterates body G i P s - Sup ?X)
    by (auto)
also from ey yit have norm (iterates body G i P s − Sup {?X}) < r 
by(simp add:dist-real-def)

finally show ∃ no. ∀ n≥no. norm (iterates body G n P s −
Sup {f s | f ∈ {t P | t ∈ range (iterates body G)}})
< r
by(auto)
qed

due to

moreover
from hb cb sP have wp do G −→ body od P s = Sup-trans (range (iterates body G)) P s
by(simp add: equiv-transD[OF lfp-iterates])

moreover have ...
Sup {f s | f ∈ {t P | t ∈ range (iterates body G)}}
by(simp add: Sup-trans-def Sup-exp-def)

ultimately show ?thesis by (simp)

qed

The iterates themselves are all continuous.

\textbf{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)

\textbf{proof}(induct i, simp-all)
have range (λ (n::nat) (s::s). 0::real) = {λs. 0::real}
by(auto)
thus bd-cts (λP (s::s). 0)
by(intro bd-ctsI, simp add:o-def Sup-exp-def)

next
fix i
assume IH: bd-cts (iterates body G i)
thus bd-cts (wp (body;; Embed (iterates body G i) s G ⊕ Skip))
healthy-intros iterates-healthy cb hb)

qed

Therefore so is the loop itself.

\textbf{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)

\textbf{proof}(rule bd-ctsI)
fix M::nat ⇒ 's expect and b::real
assume chain: ∃i. M i ⊢ M (Suc i)
and sM: ∃i. sound (M i)
and bM: ∃i. bounded-by b (M i)

from sM bM iterates-healthy[OF hb]
have ∃j i. bounded-by b (iterates body G i (M j)) by(blast)
4.4. CONTINUITY AND INDUCTION FOR LOOPS

hence \( iB : \forall j \in s. \text{iterates body } G i (M j) s \leq b \) by(auto)

from \( sM bM \) have \( sSup: \text{sound (Sup-exp (range } M) ) \)
  by(auto intro:Sup-exp-sound)
with \( \text{lfp-iterates}(OF \, sb \, cb) \)
have \( \text{wp do } G \rightarrow \text{body od (Sup-exp (range } M) ) = \)
  \( \text{Sup-trans (range (iterates body } G)) \) (Sup-exp (range } M) )
  by(simp add:equiv-transD)
also { from \( \text{chain } sM \, bM \)
  have \( \forall i. \text{iterates body } G i (\text{Sup-exp (range } M) ) = \text{Sup-exp (range (iterates body } G i o M) ) \)}
  by(blast intro:bd-ctsD cts-iterates[OF \, sb \, cb])
  hence \( \{ t (\text{Sup-exp (range } M)) \mid t. \, t \in \text{range (iterates body } G) \} = \)
  \( \{ \text{Sup-exp (range (t o M)) } \mid t. \, t \in \text{range (iterates body } G) \} \)
  by(auto intro:sym)
  hence \( \text{Sup-trans (range (iterates body } G)) \) (Sup-exp (range } M) ) =
  \( \text{Sup-exp (Sup-exp (range (t o M)) } \mid t. \, t \in \text{range (iterates body } G) \} \)
  by(simp add:Sup-trans-def)
}
  also { have \( \forall s. \, \{ f s \mid f. \, \exists t. \, f = (\lambda s. \, \text{Sup } \{ f s \mid f. \, f \in \text{range (t o M)} \} ) \wedge \)
    \( t \in \text{range (iterates body } G) \} = \)
    \( \text{range (\lambda i. \, \text{Sup (range (\lambda j. \, \text{iterates body } G i (M j) s)) ) )} \)
    (is \( \forall s. \, ?X s = ?Y s \))
  proof(intro antisym subsetI)
  fix \( s \, x \)
  assume \( x \in ?X s \)
  then obtain \( t \) where \( \text{rwx: } x = \text{Sup } \{ f s \mid f. \, f \in \text{range (t o M)} \} \)
    \( \wedge t \in \text{range (iterates body } G) \) by(auto)
  then obtain \( i \) where \( t = \text{iterates body } G i \) by(auto)
  with \( \text{rux} \) have \( x = \text{Sup } \{ f s \mid f. \, f \in \text{range (\lambda j. \, \text{iterates body } G i (M j)) } \}
    \) by(simp add:o-def)
  moreover have \( \{ f s \mid f. \, f \in \text{range (\lambda j. \, \text{iterates body } G i (M j)) } \} = \)
    \( \text{range (\lambda j. \, \text{iterates body } G i (M j) s) } \) by(auto)
  ultimately have \( x = \text{Sup (range (\lambda j. \, \text{iterates body } G i (M j) s) )} \)
    by(simp)
  thus \( x \in \text{range (\lambda i. \, \text{Sup (range (\lambda j. \, \text{iterates body } G i (M j) s) ) ) } \}
    \) by(auto)
next
fix \( s \, x \)
assume \( x \in ?Y s \)
then obtain \( i \) where \( A: x = \text{Sup (range (\lambda j. \, \text{iterates body } G i (M j) s) )} \)
  by(auto)

have \( \forall s. \, \{ f s \mid f. \, f \in \text{range (\lambda j. \, \text{iterates body } G i (M j)) } = \)
  \( \text{range (\lambda j. \, \text{iterates body } G i (M j) s) } \) by(auto)
  hence \( B: (\lambda s. \, \text{Sup (range (\lambda j. \, \text{iterates body } G i (M j) s)) ) = \)
    \( (\lambda s. \, \text{Sup } \{ f s \mid f. \, f \in \text{range (iterates body } G i o M) \} ) \)
\textbf{CHAPTER 4. THE PGCL LANGUAGE}

by (simp add: o-def)

have \( C \): \text{iterates body} \( G \) \( i \in \text{range} \) \( \text{iterates body} \( G \) \) by (auto)

have \( \exists f. \ x = f \ s \land \)
\( (\exists t. \ f = (\lambda s. \ \text{Sup} \ \{ f \ s \mid f \in \text{range} \ (t \circ M) \}) \land \)
\( t \in \text{range} \ \text{iterates body} \( G \) \)
\by (iprover intro: A B C)

thus \( x \in ?X \) \( s \) by (simp)

qed

\textbf{hence} \( \text{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: \text{Sup-exp-def})

\textbf{also have} \( (\lambda s. \ \text{Sup} \ (\text{range} \ (\lambda i. \ \text{Sup} \ (\text{range} \ (\lambda j. \ \text{iterates body} \ G \ i \ (M \ j) \ s)))))) = \)
\( (\lambda s. \ \text{Sup} \ (\text{range} \ (\lambda (i,j). \ \text{iterates body} \ G \ i \ (M \ j) \ s)))) \)
(is \( ?X = ?Y \))

\textbf{proof} (rule ext, rule antisym)

fix \( s::s \)

show \( ?Y \ s \leq ?X \ s \)

\textbf{proof} (rule cSup-least, blast, clarify)

fix \( i \ j::\text{nat} \)

from \( iB \) \textbf{have} \( \text{iterates body} \ G \ i \ (M \ j) \ s \leq \text{Sup} \ (\text{range} \ (\lambda j. \ \text{iterates body} \ G \ i \ (M \ j) \ s)) \)

\by (intro cSup-upper bdd-above1, auto)

\textbf{also from} \( iB \) \textbf{have} \( ... \leq \text{Sup} \ (\text{range} \ (\lambda j. \ \text{iterates body} \ G \ i \ (M \ j) \ s)) \)

\by (intro cSup-upper cSup-least bdd-above1, (blast intro: cSup-least)+)

\textbf{finally show} \( \text{iterates body} \ G \ i \ (M \ j) \ s \leq \)
\( \text{Sup} \ (\text{range} \ (\lambda j. \ \text{iterates body} \ G \ i \ (M \ j) \ s)) \).

qed

\textbf{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)

\textbf{thus} \( ?X \ s \leq ?Y \ s \)

\by (intro cSup-least, blast, clarify, simp, blast intro: cSup-least)

\textbf{qed}

\textbf{also have} \( ... = (\lambda s. \ \text{Sup} \ (\text{range} \ (\lambda i. \ \text{Sup} \ (\text{range} \ (\lambda j. \ \text{iterates body} \ G \ i \ (M \ j) \ s))))) \)
(is \( ?X = ?Y \))

\textbf{proof} (rule ext, rule antisym)

fix \( s::s \)

\textbf{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)

\textbf{thus} \( ?Y \ s \leq ?X \ s \)

\by (intro cSup-least, blast, clarify, simp, blast intro: cSup-least)

\textbf{show} \( ?X \ s \leq ?Y \ s \)

\textbf{proof} (rule cSup-least, blast, clarify)

fix \( i \ j::\text{nat} \)
4.4. CONTINUITY AND INDUCTION FOR LOOPS

\[
\begin{align*}
\text{from } iB & \text{ have iterates body } G \ i \ (M\ j) \ s \ \leq \ \text{Sup} \ (\text{range} \ (\lambda i. \ \text{iterates body} \ G \ i \ (M\ j) \ s)) \\
& \quad \text{by (intro cSup-upper bdd-aboveI, auto)} \\
& \quad \text{also from } iB \ \text{have } \ldots \ \leq \ \text{Sup} \ (\text{range} \ (\lambda j. \ \text{Sup} \ (\text{range} \ (\lambda i. \ \text{iterates body} \ G \ i \ (M\ j) \ s)))) \\
& \quad \text{by (intro cSup-upper cSup-least bdd-aboveI, blast, blast intro:cSup-least)} \\
& \quad \text{finally show iterates body } G \ i \ (M\ j) \ s \ \leq \\
& \quad \quad \text{Sup} \ (\text{range} \ (\lambda j. \ \text{Sup} \ (\text{range} \ (\lambda i. \ \text{iterates body} \ G \ i \ (M\ j) \ s)))).
\end{align*}
\]

\text{qed}

\text{qed}

\text{also }

\text{have } \bigwedge s. \ \text{range} \ (\lambda i. \ \text{iterates body} \ G \ i \ (M\ j) \ s) = \\
\quad \{ f s \ | f. \ \exists \ t. \ f = t \ (M\ j) \wedge \ t \in \ \text{range} \ (\text{iterates body} \ G) \} \\
& \quad \text{by (auto)} \\
& \quad \text{hence } (\lambda s. \ \text{Sup} \ (\text{range} \ (\lambda i. \ \text{iterates body} \ G \ i \ (M\ j) \ s))) \in \\
& \quad \quad \text{range} \ ((\lambda P s. \ \text{Sup} \ \{ f s \ | f. \ \exists \ t. \ f = t \ P \wedge \ t \in \ \text{range} \ (\text{iterates body} \ G) \}) \circ M) \\
& \quad \quad \quad \text{by (simp add: o-def cong del: strong-SUP-cong)}
\]

\text{ultimately show } \exists \ ?Y \ s \ \text{by (auto)}

\text{next}

\text{fix } s \ x \\
& \quad \text{assume } \exists \ ?Y \ s \\
& \text{then obtain } P \ \text{where rux: } x = P \ s \\
& \quad \quad \text{and Pin: } P \in \ \text{range} \ ((\lambda P s. \ \text{Sup} \ \{ f s \ | f. \ \exists \ t. \ f = t \ (M\ j) \wedge \ t \in \ \text{range} \ (\text{iterates body} \ G) \}) \circ M) \\
& \quad \quad \quad \text{by (auto)} \\
& \text{then obtain } j \ \text{where } P = (\lambda s. \ \text{Sup} \ \{ f s \ | f. \ \exists \ t. \ f = t \ (M\ j) \wedge \ t \in \ \text{range} \ (\text{iterates body} \ G) \}) \\
& \quad \quad \quad \text{by (auto)} \\
& \text{also }

\text{have } \bigwedge s. \ \{ f s \ | \ \exists \ t. \ f = t \ (M\ j) \wedge \ t \in \ \text{range} \ (\text{iterates body} \ G) \} = \\
& \quad \quad \quad \text{range} \ (\lambda i. \ \text{iterates body} \ G \ i \ (M\ j) \ s) \ \text{by (auto)} \\
& \quad \quad \text{hence } (\lambda s. \ \text{Sup} \ \{ f s \ | \ \exists \ t. \ f = t \ (M\ j) \wedge \ t \in \ \text{range} \ (\text{iterates body} \ G) \}) = \\
& \quad \quad \quad (\lambda s. \ \text{Sup} \ (\text{range} \ (\lambda i. \ \text{iterates body} \ G \ i \ (M\ j) \ s))) \\
& \quad \quad \quad \text{by (simp)}
\]

\text{finally have } x = \text{Sup} \ (\text{range} \ (\lambda i. \ \text{iterates body} \ G \ i \ (M\ j) \ s)) \\
& \quad \quad \text{by (simp add: rux)} \\
& \quad \quad \text{thus } x \in \ ?X \ s \ \text{by (simp)}
qed

\hence (\lambda s. \Sup (\range (\lambda j. \Sup (\range (\lambda i. \text{iterates body G i} (M j) s)))))) = 
\Sup (\range (\Suptrans (\range (\text{iterates body G}) o M)))
\by (\text{simp add: Sup-exp-def Sup-trans-def cong del: strong-SUP-cong})

also have \Sup (\range (\Suptrans (\range (\text{iterates body G}) o M))) = 
\Sup (\range (wp do G \longrightarrow body od o M))
\by (\text{simp add: o-def equiv-transD [OF lfp-iterates, OF hb cb, OF sM]})

finally show wp do G \longrightarrow body od (\Sup (\range M)) = 
\Sup (\range (wp do G \longrightarrow body od o M)) .

qed

lemmas cts-intros =
cts-wp-Abort cts-wp-Skip
cts-wp-Seq cts-wp-PC
cts-wp-DC cts-wp-Embed
cts-wp-Apply cts-wp-SetDC
cts-wp-SetPC cts-wp-Bind
cts-wp-repeat

end

4.5 Sublinearity

theory Sublinearity imports Embedding Healthiness LoopInduction begin

4.5.1 Nonrecursive Primitives

Sublinearity of non-recursive programs is generally straightforward, and follows from the algebraic properties of the underlying operations, together with healthiness.

lemma sublinear-wp-Skip:
sublinear (wp Skip)
\by (auto simp: wp-eval)

lemma sublinear-wp-Abort:
sublinear (wp Abort)
\by (auto simp: wp-eval)

lemma sublinear-wp-Apply:
sublinear (wp (Apply f))
\by (auto simp: wp-eval)

lemma sublinear-wp-Seq:
fixes x :: 's prog
assumes slx: sublinear (wp x) and sly: sublinear (wp y)
and hx: healthy (wp x) and hy: healthy (wp y)
shows sublinear (wp (x ;; y))
4.5. SUBLINEARITY

proof\(\text{rule sublinearI, simp add:wp-eval}\)
  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 \(a * \text{wp} x (\text{wp} y P) s + b * \text{wp} x (\text{wp} y Q) s ≤ \text{wp} x (\lambda s. a * \text{wp} y P s + b * \text{wp} y Q s) s\)
  by(blast intro:sublinearD)

also \{\}
  from sP sQ nna nnb nnc sly
  have \(\lambda s. a * \text{wp} y P s + b * \text{wp} y Q s ≤ \text{wp} y (\lambda s. a * P s + b * Q s) s\)
  by(blast intro:sublinearD)

moreover from sP sQ by
  have sound (\text{wp} y P) and sound (\text{wp} y Q) by(auto)

moreover with nna nnb nnc
  have sound (\lambda s. a * \text{wp} y P s + b * \text{wp} y Q s) by(auto intro!:sound-intros tminus-sound)

moreover from sP sQ nna nnb nnc
  have sound (\lambda s. a * P s + b * Q s) by(auto intro!:sound-intros tminus-sound)

moreover with by have sound (\text{wp} y (\lambda s. a * P s + b * Q s) s) by(blast)

ultimately
  have \(\text{wp} x (\lambda s. a * \text{wp} y P s + b * \text{wp} y Q s) s ≤ \text{wp} x (\lambda s. a * P s + b * Q s) s\)
  by(blast intro!:le-funD[OF mono-transD[OF healthy-monoD[OF \text{OF}\hspace{1em}]]])

}\}

finally show \(a * \text{wp} x (\text{wp} y P) s + b * \text{wp} x (\text{wp} y Q) s ≤ \text{wp} x (\lambda s. a * P s + b * Q s) s\).

qed

lemma sublinear-wp-PC:
  fixes x::'s prog
  assumes slx: sublinear (\text{wp} x) and sly: sublinear (\text{wp} y)
  and uP: unitary P
  shows sublinear (\text{wp} (x p\oplus y))

proof\(\text{rule sublinearI, simp add:wp-eval}\)
  fix R::'s ⇒ real and Q::'s ⇒ real and s::'s
  and a::real and b::real and c::real
  assume sR: sound R and sQ: sound Q
  and nna: 0 ≤ a and nnb: 0 ≤ b and nnc: 0 ≤ c

  have \(a * (P s * \text{wp} x Q s + (1 - P s) * \text{wp} y Q s) + \)
      \(b * (P s * \text{wp} x R s + (1 - P s) * \text{wp} y R s) ≤ c = \)
      \((P s * a * \text{wp} x Q s + (1 - P s) * a * \text{wp} y Q s) + \)
      \((P s * b * \text{wp} x R s + (1 - P s) * b * \text{wp} y R s) ≤ c\)
  by(simp add:field-simps)
also have ... = (P s ∗ a ∗ wp x Q s + P s ∗ b ∗ wp x R s) +
   (1 − P s) ∗ a ∗ wp y Q s + (1 − P s) ∗ b ∗ wp y R s) ⊓ c
   by(simp add:ac-simps)
also have ... = P s ∗ (a ∗ wp x Q s + b ∗ wp x R s) +
   (1 − P s) ∗ (a ∗ wp y Q s + b ∗ wp y R s) ⊓ (P s ∗ c + (1 − P s) ∗ c)
   by(simp add:field-simps)
also have ... ≤ (P s ∗ (a ∗ wp x Q s + b ∗ wp x R s) ⊓ P s ∗ c) +
   (1 − P s) ∗ (a ∗ wp y Q s + b ∗ wp y R s) ⊓ (1 − P s) ∗ c)
   by(rule tminus-add-mono)
also { from uP have 0 ≤ P s and 0 ≤ 1 − P s
   by(auto simp:sign-simps)
   hence (P s ∗ (a ∗ wp x Q s + b ∗ wp x R s) ⊓ P s ∗ c) +
       (1 − P s) ∗ (a ∗ wp y Q s + b ∗ wp y R s) ⊓ (1 − P s) ∗ c) =
       P s ∗ (a ∗ wp x Q s + b ∗ wp x R s ⊓ c) +
       (1 − P s) ∗ (a ∗ wp y Q s + b ∗ wp y R s ⊓ c)
   by(simp add:tminus-left-distrib)
}
also { from sQ sR nna nmb nnc slx
   have a ∗ wp x Q s + b ∗ wp x R s ⊓ c ≤
       wp x (λs. a ∗ Q s + b ∗ R s ⊓ c) s
   by(blast)
   moreover from sQ sR nna nmb nnc sly
   have a ∗ wp y Q s + b ∗ wp y R s ⊓ c ≤
       wp y (λs. a ∗ Q s + b ∗ R s ⊓ c) s
   by(blast)
   moreover from uP have 0 ≤ P s and 0 ≤ 1 − P s
   by(auto simp:sign-simps)
   ultimately have P s ∗ (a ∗ wp x Q s + b ∗ wp x R s ⊓ c) +
       (1 − P s) ∗ (a ∗ wp y Q s + b ∗ wp y R s ⊓ c) ≤
       P s ∗ wp x (λs. a ∗ Q s + b ∗ R s ⊓ c) s +
       (1 − P s) ∗ wp y (λs. a ∗ Q s + b ∗ R s ⊓ c) s
   by(blast intro:add-mono mult-left-mono)
}
finally show a ∗ (P s ∗ wp x Q s + (1 − P s) ∗ wp y Q s) +
    b ∗ (P s ∗ wp x R s + (1 − P s) ∗ wp y R s) ⊓ c ≤
    P s ∗ wp x (λs. a ∗ Q s + b ∗ R s ⊓ c) s +
    (1 − P s) ∗ wp y (λs. a ∗ Q s + b ∗ R s ⊓ c) s .
qed
4.5. **SUBLINEARITY**

**Lemma** sublinear-wp-DC:

- **Fixes** $x$'s prog
- **Assumes** slx: sublinear (wp $x$) and sly: sublinear (wp $y$)
- **Shows** sublinear (wp ($x \sqcap y$))

**Proof** (rule sublinearI, simp only: 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 $sR$: sound $R$ and $sQ$: sound $Q$
  - and $nna$: $0 \leq a$ and $nnb$: $0 \leq b$ and $nnc$: $0 \leq c$

from $nna$ $nnb$

- have $a * \operatorname{min} (\operatorname{wp} x \ Q \ s) (\operatorname{wp} y \ Q \ s) +$
  - $b * \operatorname{min} (\operatorname{wp} x \ R \ s) (\operatorname{wp} y \ R \ s) \sqcup c =$
  - $\operatorname{min} (a * \operatorname{wp} x \ Q \ s) (a * \operatorname{wp} y \ Q \ s) +$
  - $\operatorname{min} (b * \operatorname{wp} x \ R \ s) (b * \operatorname{wp} y \ R \ s) \sqcup c$
  - by (simp add: min-distrib)

also have ...

by (auto intro!: tminus-left-mono)

also have ...

by (rule min-tminus-distrib)

also { from $slx \ sQ \ sR \ nna \ nnb \ nnc$

  - have $a * \operatorname{wp} x \ Q \ s + b * \operatorname{wp} x \ R \ s \sqcup c \leq$
    - $\operatorname{wp} x (\lambda s. a * Q \ s + b * R \ s \sqcup c) \ s$
    - by (blast)

  moreover from $sly \ sQ \ sR \ nna \ nnb \ nnc$

  - have $a * \operatorname{wp} y \ Q \ s + b * \operatorname{wp} y \ R \ s \sqcup c \leq$
    - $\operatorname{wp} y (\lambda s. a * Q \ s + b * R \ s \sqcup c) \ s$
    - by (blast)

  ultimately

  - have $\operatorname{min} (a * \operatorname{wp} x \ Q \ s + b * \operatorname{wp} x \ R \ s \sqcup c)$
    - $(a * \operatorname{wp} y \ Q \ s + b * \operatorname{wp} y \ R \ s \sqcup c) \leq$
    - $\operatorname{min} (\operatorname{wp} x (\lambda s. a * Q \ s + b * R \ s \sqcup c) \ s)$
    - $(\operatorname{wp} y (\lambda s. a * Q \ s + b * R \ s \sqcup c) \ s)$
      - by (auto)

  }

finally show $a * \operatorname{min} (\operatorname{wp} x \ Q \ s) (\operatorname{wp} y \ Q \ s) +$

$\operatorname{min} (\operatorname{wp} x \ R \ s) (\operatorname{wp} y \ R \ s) \sqcup c \leq$

$\operatorname{min} (\operatorname{wp} x (\lambda s. a * Q \ s + b * R \ s \sqcup c) \ s)$

$(\operatorname{wp} y (\lambda s. a * Q \ s + b * R \ s \sqcup c) \ s)$

qed

As for continuity, we insist on a finite support.

**Lemma** sublinear-wp-SetPC:
fixes $p \vdash a \Rightarrow 's \ prog$
assumes $slp: \forall s. a \in \supp (P \ s) \implies \text{sublinear} \ (wp \ (p \ a))$
and $\text{sum}: \forall s. (\sum a \in \supp (P \ s). P \ s \ a) \leq 1$
and $\text{nnP}: \forall s. 0 \leq P \ s \ a$
and $\text{fin}: \forall s. \text{finite} (\supp (P \ s))$
shows $\text{sublinear} \ (wp \ (\text{SetPC} \ p \ P))$

**proof** (rule sublinearI, simp add:wp-eval)
fix $R::'s \Rightarrow \text{real}$ and $Q::'s \Rightarrow \text{real}$ and $s::'s$
and $a::\text{real}$ and $b::\text{real}$ and $c::\text{real}$
assume $sR: \text{sound} \ R$ and $sQ: \text{sound} \ Q$
and $\text{nn}: 0 \leq a$ and $\text{nnb}: 0 \leq b$ and $\text{nnc}: 0 \leq c$
have $a \ast (\sum a' \in \supp (P \ s). P \ s \ a' \ast wp \ (p \ a') \ Q \ s) +$
\hspace{1em} $b \ast (\sum a' \in \supp (P \ s). P \ s \ a' \ast wp \ (p \ a') \ R \ s) \ominus c =$
\hspace{1em} $(\sum a' \in \supp (P \ s). P \ s \ a' \ast (a \ast wp \ (p \ a') \ Q \ s + b \ast wp \ (p \ a') \ R \ s)) \ominus c$
by (simp add:field-simps sum-distrib-left sum-distrib)
also have ... \leq 
\hspace{1em} $(\sum a' \in \supp (P \ s). P \ s \ a' \ast (a \ast wp \ (p \ a') \ Q \ s + b \ast wp \ (p \ a') \ R \ s)) \ominus P \ s \ a' \ast c$

**proof** (rule tminus-right-antimono)
have $(\sum a' \in \supp (P \ s). P \ s \ a' \ast c) \leq (\sum a' \in \supp (P \ s). P \ s \ a' \ast c)$
by (simp add:sum-distrib-right)
also from sum and nnc have ... \leq 1 \ast c
by (rule mult-right-mono)
finally show $(\sum a' \in \supp (P \ s). P \ s \ a' \ast c) \leq c$ by (simp)
qed
also from fin
have ... \leq $(\sum a' \in \supp (P \ s). P \ s \ a' \ast (a \ast wp \ (p \ a') \ Q \ s + b \ast wp \ (p \ a') \ R \ s) \ominus P \ s \ a' \ast c)$
by (blast intro: tminus-sum-mono)
also have ... = $(\sum a' \in \supp (P \ s). P \ s \ a' \ast (a \ast wp \ (p \ a') \ Q \ s + b \ast wp \ (p \ a') \ R \ s) \ominus c)$
by (simp add:nnP tminus-left-distrib)
also {
  from slp sQ sR sna nnb nnc
  have $\forall a'. a' \in \supp (P \ s) \implies a \ast wp \ (p \ a') \ Q \ s + b \ast wp \ (p \ a') \ R \ s \ominus c \leq$
\hspace{1em} $wp \ (p \ a') \ (\lambda s. a \ast Q \ s + b \ast R \ s \ominus c) \ s$
by (blast)
with nnP
have $(\sum a' \in \supp (P \ s). P \ s \ a' \ast wp \ (p \ a') \ (\lambda s. a \ast Q \ s + b \ast R \ s \ominus c) \ s) \leq$
\hspace{1em} $(\sum a' \in \supp (P \ s). P \ s \ a' \ast wp \ (p \ a') \ (\lambda s. a \ast Q \ s + b \ast R \ s \ominus c) \ s)$
by (blast intro:sum-mono mult-left- mono)
}
finally
\hspace{1em} show $a \ast (\sum a' \in \supp (P \ s). P \ s \ a' \ast wp \ (p \ a') \ Q \ s) +$
\hspace{1em} $b \ast (\sum a' \in \supp (P \ s). P \ s \ a' \ast wp \ (p \ a') \ R \ s) \ominus c \leq$
\hspace{1em} $(\sum a' \in \supp (P \ s). P \ s \ a' \ast wp \ (p \ a') \ (\lambda s. a \ast Q \ s + b \ast R \ s \ominus c) \ s)$. 
qed
4.5. **SUBLINEARITY**

**lemma** sublinear-wp-SetDC:

- **fixes** \( p \)' : \( a \Rightarrow ' s \ \text{proo} \)
- **assumes** \( slp. \ \lambda s. a \in S \ s \Rightarrow \text{sublinear} \ (wp \ (p \ a)) \)
  - and \( hp. \ \lambda s. a \in S \ s \Rightarrow \text{healthy} \ (wp \ (p \ a)) \)
  - and \( nc. \ \lambda s. S \ s \neq \{ \} \)
- **shows** sublinear \( (wp \ (SetDC \ p \ S)) \)

**proof** (\( \text{rule \ sublinearI}, \ \text{simp add:wp-eval}, \ \text{rule cInf-greatest} \))

**fix** \( P::'s \Rightarrow \text{real} \ \text{and} \ Q::'s \Rightarrow \text{real} \ \text{and} \ s::'s \ \text{and} \ x \ y \)

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

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

- and \( \text{nna: } 0 \leq a \ \text{and} \ \text{nmb: } 0 \leq b \ \text{and} \ \text{nnc: } 0 \leq c \)

**from** \( \text{ne show} \ (\lambda pr. \ wp \ (p \ pr) \ (\lambda s. a \ast P \ s + b \ast Q \ s \ominus c) \ s) \ \ominus S \ s \neq \{ \} \) by(auto)

**assume** \( yin: y \in (\lambda pr. \ wp \ (p \ pr) \ (\lambda s. a \ast P \ s + b \ast Q \ s \ominus c) \ s) \ \ominus S \ s \)

**then obtain** \( x \text{ where } xin: x \in S \ s \ \text{and} \ rwy: y = wp \ (p \ x) \ (\lambda s. a \ast P \ s + b \ast Q \ s \ominus c) \ s \)

by(auto)

**from** \( xin \ hp \ sP \ nna \)

**have** \( a \ast \inf ((\lambda a. \ wp \ (p \ a) \ P \ s) \ominus S \ s) \leq a \ast wp \ (p \ x) \ P \ s \)

by(intro mult-left-mono[OF cInf-lower]bdd-belowI[where \( m=0 \)], blast+)

**moreover** **from** \( xin \ hp \ sQ \ nmb \)

**have** \( b \ast \inf ((\lambda a. \ wp \ (p \ a) \ Q \ s) \ominus S \ s) \leq b \ast wp \ (p \ x) \ Q \ s \)

by(intro mult-left-mono[OF cInf-lower]bdd-belowI[where \( m=0 \)], blast+)

**ultimately**

**have** \( a \ast \inf ((\lambda a. \ wp \ (p \ a) \ P \ s) \ominus S \ s) + \)

\( b \ast \inf ((\lambda a. \ wp \ (p \ a) \ Q \ s) \ominus S \ s) \ominus c \leq \)

\( a \ast wp \ (p \ x) \ P \ s + b \ast wp \ (p \ x) \ Q \ s \ominus c \)

by(blast intro:minus-left-mono add-mono)

**also from** \( xin \ slp \ sP \ sQ \ nna \ nmb \ nnc \)

**have** \( \leq wp \ (p \ x) \ (\lambda s. a \ast P \ s + b \ast Q \ s \ominus c) \ s \)

by(blast)

**finally show** \( a \ast \inf ((\lambda a. \ wp \ (p \ a) \ P \ s) \ominus S \ s) + b \ast \inf ((\lambda a. \ wp \ (p \ a) \ Q \ s) \ominus S \ s) \ominus c \leq y \)

by(simp add:rwy)

qed

**lemma** sublinear-wp-Embed:

- sublinear \( t \Rightarrow \text{sublinear} \ (wp \ (Embed \ t)) \)

by(simp add:wp-eval)

**lemma** sublinear-wp-repeat:

[ sublinear \( (wp \ p) \); healthy \( (wp \ p) \) ] \( \Rightarrow \) sublinear \( (wp \ (\text{repeat} \ n \ p)) \)

by(induct \( n \), simp-all add:sublinear-wp-Seq sublinear-wp-Skip healthy-wp-repeat)

**lemma** sublinear-wp-Bind:
\[ \text{sub-distrib-wp-loop} \implies \text{sublinear \ (wp \ (a \ (f \ s)))} \]
by(rule sublinearI, simp add:wp-eval, auto)

### 4.5.2 Sublinearity for Loops

We break the proof of sublinearity loops into separate proofs of sub-distributivity and sub-additivity. The first follows by transfinite induction.

**Lemma** sub-distrib-wp-loop:

- **Fixes** body::'s prog
- **Assumes** sdb: sub-distrib (wp body)
- and hb: healthy (wp body)
- and nhb: nearly-healthy (wp body)

**Shows** sub-distrib (wp (do G \longrightarrow body od))

**Proof** –

1. **Have** \( \forall P \ s. \ \text{sound} \ P \implies \text{wp} \ (\text{do} \ G \implies \text{body od}) \ P \ s \oplus 1 \leq \text{wp} \ (\text{do} \ G \implies \text{body od}) \ (\lambda s. \ P \ s \oplus 1) \ s \)
2. **Proof** (rule loop-induct[OF hb nhb], safe)
   - **Fix** \( S::'s \ trans \times 's \ trans \ \text{set} \) and \( P::'s \ expect \) and \( s::'s \)
   - **Assume** saS: \( \forall x:S. \ \forall P. \ \text{sound} \ P \implies \text{fst} x \ P \ s \oplus 1 \leq \text{fst} x \ (\lambda s. \ P \ s \oplus 1) \ s \)
   - and \( sP: \ \text{sound} \ P \)
   - and \( FS: \forall x:S. \ \text{feasible} \ (\text{fst} x) \)

From \( sP \) have \( sPm: \ \text{sound} \ (\lambda s. \ P \ s \oplus 1) \) by(auto intro: tminus-sound)

1. **Have** \( \text{mnSup}: \ \forall s. \ 0 \leq \ \text{Sup-trans} \ (\text{fst} \ s:S) \ (\lambda s. \ P \ s \oplus 1) \ s \)
2. **Proof** (cases \( S=\{\} \), simp add: Sup-trans-def Sup-exp-def)
   - **Fix** \( s \)
   - **Assume** \( S \neq \{\} \)
3. **Then obtain** \( x \) where \( \text{xin: } x:S \) by(auto)
4. **With** \( \text{fs} sPm \) **have** \( 0 \leq \text{fst} x \ (\lambda s. \ P \ s \oplus 1) \ s \) by(auto)
5. **Also from** \( \text{xin} \ FS \ sPm \) **have** \( ... \leq \text{Sup-trans} \ (\text{fst} \ s:S) \ (\lambda s. \ P \ s \oplus 1) \ s \) by(auto intro!: le-funD[OF Sup-trans-upper2])
6. **Finally show** \( ?\text{thesis s} \).

**Qed**

1. **Have** \( \forall x. \ \text{fst} x \ P \ s \leq (\text{fst} x \ P \ s \oplus 1) + 1 \) by(simp add:tminus-def)
2. **Also from** \( \text{saS} \ sP \)
3. **Have** \( \forall x. \ \forall x:S \implies (\text{fst} x \ P \ s \oplus 1) + 1 \leq \text{fst} x \ (\lambda s. \ P \ s \oplus 1) \ s + 1 \) by(auto intro: add-right-mono)
4. **Also** \( \)
   - **From** \( sP \) **have** \( \text{sound} \ (\lambda s. \ P \ s \oplus 1) \) by(auto intro:tminus-sound)
   - **With** \( \text{fs} \) **have** \( \forall x. \ \forall x:S \implies \text{fst} x \ (\lambda s. \ P \ s \oplus 1) \ s + 1 \leq \text{Sup-trans} \ (\text{fst} \ s:S) \ (\lambda s. \ P \ s \oplus 1) \ s + 1 \)
   - by(blast intro!: add-right-mono le-funD[OF Sup-trans-upper2])
5. **Finally have** \( \text{le}: \ \forall x:S. \ \text{fst} x \ P \ s \leq \text{Sup-trans} \ (\text{fst} \ s:S) \ (\lambda s. \ P \ s \oplus 1) \ s + 1 \) by(auto)
6. **Moreover from** \( \text{mnSup} \) **have** \( \forall s. \ 0 \leq \text{Sup-trans} \ (\text{fst} \ s:S) \ (\lambda s. \ P \ s \oplus 1) \ s + 1 \)
4.5. SUBLINEARITY

by(auto intro:add-nonneg-nonneg)
ultimately
have leSup: Sup-trans (fst ' S) P s ≤ Sup-trans (fst ' S) (λs. P s ⊗ 1) s + 1
unfolding Sup-trans-def
by(intro le-funD[of Sup-exp-least], auto)

show Sup-trans (fst ' S) P s ⊗ 1 ≤ Sup-trans (fst ' S) (λs. P s ⊗ 1) s
proof(cases Sup-trans (fst ' S) P s ≤ 1, simp-all add:nnSup)
from leSup have Sup-trans (fst ' S) P s − 1 ≤
Sup-trans (fst ' S) (λs. P s ⊗ 1) s + 1 − 1
by(auto)
thus Sup-trans (fst ' S) P s − 1 ≤ Sup-trans (fst ' S) (λs. P s ⊗ 1) s
by(simp)
qed
next
fix t::'s trans and P::'s expect and s::'s
assume IH: ∀P s. sound P t P s ⊗ 1 ≤ t (λa. P a ⊗ 1) s
and ft: feasible t
and sP: sound P

from sP have sound (λs. P s ⊗ 1) by(auto intro:tminus-sound)
with ft have s2: sound (t (λs. P s ⊗ 1)) by(auto)
from sP ft have sound (t P) by(auto)

hence s3: sound (λs. t P s ⊗ 1) by(auto intro:tminus-sound)

show wp (body :: Embed t (« G » ⊕ Skip) P s ⊗ 1 ≤ wp (body :: Embed t (« G » ⊕ Skip) (λa. P a ⊗ 1) s
proof(simp add:wp-eval)
have "« G » s * wp body (t P) s + (1 − « G » s) * P s ⊗ 1 =
" « G » s * wp body (t P) s + (1 − « G » s) * P s ⊗ (« G » s + (1 − « G » s))
by(simp)
also have "... ≤ (« G » s * wp body (t P) s ⊗ « G » s) +
(1 − « G » s) * P s ⊗ (1 − « G » s))
by(rule tminus-add-mono)
also have "... = (« G » s * (wp body (t P) s ⊗ 1) + (1 − « G » s) * (P s ⊗ 1))
by(simp add:tminus-left-distrib)
also { from ft sP have wp body (t P) s ⊗ 1 ≤ wp body (λs. t P s ⊗ 1) s
by(auto intro:sub-distribD[of sdb])
also { from IH sP have λs. t P s ⊗ 1 ⊢ t (λs. P s ⊗ 1) by(auto)
with sP ft s2 s3 have wp body (λs. t P s ⊗ 1) s ≤ wp body (t (λs. P s ⊗ 1)) s
by(blast intro:le-funD[of mono-transD, OF healthy-monoD, OF hh])
} finally have "« G » s * (wp body (t P) s ⊗ 1) + (1 − « G » s) * (P s ⊗ 1) ≤
« G » s * wp body (t (λs. P s ⊗ 1)) s + (1 − « G » s) * (P s ⊗ 1)
by(auto intro:add-right-mono mult-left-mono)\)
finally show $G \cdot s \ast \text{wp body} (t \cdot P) \cdot s + (1 - G \cdot s) \ast P \cdot s \ominus 1 \leq G \cdot s \ast \text{wp body} ((t \cdot \lambda s. \cdot P) \cdot s + (1 - G \cdot s) \ast (P \cdot s \ominus 1))$.

qed

next

fix \( t', \cdot 's \) trans and \( P:\cdot 's \) expect and \( s:\cdot 's \)

assume IH: \( \forall P. \text{sound } P \rightarrow t \cdot P \ominus 1 \leq t' (\lambda a. \cdot P \cdot a \ominus 1) \cdot s \)

and eq: equiv-trans \( t, t' \) and \( s : \text{sound } P \)

from \( s \cdot P \) have \( t' \cdot P \ominus 1 \leq t' (\lambda s. \cdot P \cdot s \ominus 1) \cdot s \)

also from \( s \cdot P \) IH have \( ... \leq t (\lambda s. \cdot P \cdot s \ominus 1) \cdot s \) by(auto)

also { from \( s \cdot P \) have sound \( (\lambda s. \cdot P \cdot s \ominus 1) \cdot s \) by(simp add: equiv-transD[OF eq])

hence \( t (\lambda s. \cdot P \cdot s \ominus 1) \cdot s = t' (\lambda s. \cdot P \cdot s \ominus 1) \cdot s \) by(simp add: equiv-transD[OF eq])

} finally show \( t' \cdot P \ominus 1 \leq t' (\lambda s. \cdot P \cdot s \ominus 1) \cdot s \).

qed

thus \( ? \) thesis by(auto intro!: sub-distribI)

qed

For sub-additivity, we again use the limit-of-iterates characterisation. Firstly, all iterates are sublinear:

lemma sublinear-iterates:
assumes \( hb: \text{healthy } (\text{wp body}) \)
and \( sb: \text{sublinear } (\text{wp body}) \)
shows sublinear (iterates body \( G \cdot i \))

by(induct \( i \), auto intro!: sublinear-wp-PC sublinear-wp-Seq sublinear-wp-Skip sublinear-wp-Embed
assms healthy-intros iterates-healthy)

From this, sub-additivity follows for the limit (i.e. the loop), by appealing to the property at all steps.

lemma sub-add-wp-loop:
fixes \( \cdot 's \) prog
assumes \( sb: \text{sublinear } (\text{wp body}) \)
and \( cb: \text{bd-cts } (\text{wp body}) \)
and \( hwp: \text{healthy } (\text{wp body}) \)
shows sub-add (wp \( (\text{do } G \rightarrow \text{body od}) \))

proof
fix \( P, Q:: 's \) expect and \( s:: 's \)
assume \( s \cdot P: \text{sound } P \) and \( s \cdot Q: \text{sound } Q \)

from \( hwp cb s \cdot P \) have \((\lambda i. \text{iterates body } G \cdot i \cdot P) \rightarrow \text{wp } G \rightarrow \text{body od } P \cdot s \)

by(rule loop-iterates)

moreover
from \( hwp cb s \cdot Q \) have \((\lambda i. \text{iterates body } G \cdot i \cdot Q) \rightarrow \text{wp } G \rightarrow \text{body od } Q \cdot s \)

by(rule loop-iterates)

ultimately
4.5. **SUBLINEARITY**

have $(\lambda_i. \text{iterates body } G i P s + \text{iterates body } G i Q s) \longrightarrow
wp \text{ do } G \longrightarrow \text{body od } P s + wp \text{ do } G \longrightarrow \text{body od } Q s$

by (rule tendsto-add)

moreover {
  from sublinear-subadd$[(\text{OF sublinear-iterates, OF hwp sb,})$
  $\text{OF healthy-feasibleD(OF iterates-healthy, OF hwp)}] sP sQ$
  have $\bigwedge_i. \text{iterates body } G i P s + \text{iterates body } G i Q s \leq \text{iterates body } G i (\lambda s. P s + Q s) s$
  by (rule sub-addD)
}

moreover {
  from $sP sQ$ have sound $(\lambda s. P s + Q s)$ by (blast intro: sound-intros)
  with hwp cb have $(\lambda_i. \text{iterates body } G i (\lambda s. P s + Q s) s) \longrightarrow
wp \text{ do } G \longrightarrow \text{body od } (\lambda s. P s + Q s) s$
  by (blast intro: loop-iterates)
}

ultimately show $wp \text{ do } G \longrightarrow \text{body od } P s + wp \text{ do } G \longrightarrow \text{body od } Q s \leq \text{wp do } G \longrightarrow \text{body od } (\lambda s. P s + Q s) s$

by (blast intro: LIMSEQ-le)

qed

**Lemma** sublinear-wp-loop:

fixes body::'s prog

assumes hh: 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 $\longrightarrow$ body od))

using sublinear-sub-distrib[OF sb] sublinear-subadd[OF sb]
  $\text{hb healthy-feasibleD[OF hb]}$

by (iprover intro: sd-sa-sublinear[OF - - healthy-wp-loop[OF hb]]
  sub-distrib-wp-loop sub-add-wp-loop assms)

**Lemmas** sublinear-intros =

sublinear-wp-Abort
sublinear-wp-Skip
sublinear-wp-Apply
sublinear-wp-Seq
sublinear-wp-PC
sublinear-wp-DC
sublinear-wp-SetPC
sublinear-wp-SetDC
sublinear-wp-Embed
sublinear-wp-repeat
sublinear-wp-Bind
sublinear-wp-loop

end
4.6 Determinism

theory Determinism imports WellDefined begin

We provide a set of lemmas for establishing that appropriately restricted programs are fully additive, and maximal in the refinement order. This is particularly useful with data refinement, as it implies correspondence.

4.6.1 Additivity

lemma additive-wp-Abort:
additive (wp (Abort))
by(auto simp:wp-eval)

wlp Abort is not additive.

lemma additive-wp-Skip:
additive (wp (Skip))
by(auto simp:wp-eval)

lemma additive-wp-Apply:
additive (wp (Apply f))
by(auto simp:wp-eval)

lemma additive-wp-Seq:
fixes a::ʼs 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::ʼs ⇒ real and Q::ʼs ⇒ real and s::ʼs
assume sP: sound P and sQ: sound Q
note hb = well-def-wp-healthy[OF wb]
from addb sP sQ
have wp b (λs. P s + Q s) = (λs. wp b P s + wp b Q s)
by(blast dest:additiveD)
with adda sP sQ hb
show wp a (wp b (λs. P s + Q s)) s =
wp a (wp b P) s + (wp a (wp b Q)) s
by(auto intro:fun-cong[OF additiveD])
qed

lemma additive-wp-PC:
[ additive (wp a); additive (wp b) ] ⇒ additive (wp (a p⊔ b))
by(rule additiveI, simp add:additiveD field-simps wp-eval)

DC is not additive.
4.6. DETERMINISM

lemma additive-wp-SetPC:
\[
\begin{align*}
\forall x. x \in \text{supp}(p s) \implies & \text{additive}(\text{wp}(a x)) \\
\forall s. \text{finite}(\text{supp}(p s)) \implies & \text{additive}(\text{wp}(\text{SetPC} a p))
\end{align*}
\]
by (rule additiveI,
    simp add: wp-eval additiveD distrib-left sum_distrib)

lemma additive-wp-Bind:
\[
\begin{align*}
\forall x. & \text{additive}(\text{wp}(a (f x))) \\
\forall s. & \text{finite}(\text{supp}(p s)) \implies \text{additive}(\text{wp}(\text{Bind} f a))
\end{align*}
\]
by (simp add: wp-eval additive-def)

lemma additive-wp-Embed:
\[
\begin{align*}
\forall t. & \text{additive}(\text{wp}(\text{Embed} t))
\end{align*}
\]
by (simp add: wp-eval)

lemma additive-wp-repeat:
additive(\text{wp} a) \implies \text{well-def} a \implies additive(\text{wp}(\text{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(\text{wp} \text{Skip})
by (simp add: maximal-def wp-eval)

lemma max-wp-Apply:
maximal(\text{wp} (\text{Apply} f))
by (auto simp: wp-eval o_def)

lemma max-wp-Seq:
\[
\begin{align*}
\forall a b. & \text{maximal}(\text{wp} a); \text{maximal}(\text{wp} b) \implies \text{maximal}(\text{wp}(a ;; b))
\end{align*}
\]
by (simp add: wp-eval maximal-def)

lemma max-wp-PC:
\[
\begin{align*}
\forall a b. & \text{maximal}(\text{wp} a); \text{maximal}(\text{wp} b) \implies \text{maximal}(\text{wp}(a P b))
\end{align*}
\]
by (rule maximalI, simp add: maximalD field-simps wp-eval)

lemma max-wp-DC:
\[
\begin{align*}
\forall a b. & \text{maximal}(\text{wp} a); \text{maximal}(\text{wp} b) \implies \text{maximal}(\text{wp}(a d b))
\end{align*}
\]
by (rule maximalI, simp add: wp-eval maximalD)

lemma max-wp-SetPC:
\[
\begin{align*}
\forall s a. & a \in \text{supp}(P s) \implies \text{maximal}(\text{wp}(p a); \forall s. (\sum a \in \text{supp}(P s). P s a) =
\end{align*}
\]
\[ \text{maximal} \left( \wp (\text{SetPC} p P) \right) \]
by \((\text{auto simp:maximalD wp-def SetPC-def sum-distrib-right[symmetric]})\)

**lemma max-wp-SetDC:**
fixes \(p::'a \Rightarrow 's \text{ prog}\)
assumes \(mp: \forall s. a \in S \implies \text{maximal} \left( \wp (p a) \right)\)
and \(ne: \forall S. S \neq \{\}\)
shows \(\text{maximal} \left( \wp (\text{SetDC} p S) \right)\)
**proof** (rule maximalI, rule ext, unfold wp-eval)
fix \(c::\text{real} \) and \(s::'s\)
assume \(0 \leq c\)
\[\begin{align*}
\text{hence } \inf \left( (\lambda a. \wp (p a) (\lambda -. c) s) \cdot S s \right) &= \inf \left( (\lambda -. c) \cdot S s \right) \\
\text{using } mp \text{ by (simp add:maximalD cong:image-cong)}
\end{align*}\]
also \{
from \(ne\) obtain \(a\) where \(a \in S\) by blast
\[\begin{align*}
\text{hence } \inf \left( (\lambda -. c) \cdot S s \right) &= c \\
\text{by (auto simp add: image-constant-conv cong del: strong-INF-cong)}
\end{align*}\}
\}
finally show \(\inf \left( (\lambda a. \wp (p a) (\lambda -. c) s) \cdot S s \right) = c\).
\[\text{qed}\]

**lemma max-wp-Embed:**
maximal \(t \implies \text{maximal} \left( \wp (\text{Embed} t) \right)\)
by (simp add:wp-eval)

**lemma max-wp-repeat:**
\(\text{maximal} \left( \wp a \right) \implies \text{maximal} \left( \wp (\text{repeat} \ n \ a) \right)\)
by (induct \(n\), simp-all add:max-wp-Skip max-wp-Seq)

**lemma max-wp-Bind:**
assumes \(ma: \forall s. \text{maximal} \left( \wp (a (f s)) \right)\)
shows \(\text{maximal} \left( \wp (\text{Bind} f a) \right)\)
**proof** (rule maximalI, rule ext, simp add:wp-eval)
fix \(c::\text{real} \) and \(s\)
assume \(0 \leq c\)
with \(ma\) have \(\wp (a (f s)) (\lambda -. c) = (\lambda -. c)\) \text{by(blast)}
thus \(\wp (a (f s)) (\lambda -. c) s = c\) \text{by(auto)}
\[\text{qed}\]

**lemmas** max-intros =
max-wp-Skip max-wp-Apply
max-wp-Seq max-wp-PC
max-wp-DC max-wp-SetPC
max-wp-SetDC max-wp-Embed
max-wp-Bind max-wp-repeat

A healthy transformer that terminates is maximal.

**lemma** healthy-term-max:
4.6. DETERMINISM

assumes $ht$: healthy $t$
    and $trm$: $\lambda s. 1 \vdash t (\lambda s. 1)$
shows maximal $t$

proof
    intro maximalI ext
    fix $c::\mathbb{real}$ and $s$
assume $nnc$: $0 \leq c$

have $t (\lambda s. c) s = t (\lambda s. 1 + c) s$ by simp
also from $nnc$ healthy-scalingD [OF $ht$]
have ... = $c * t (\lambda s. 1) s$ by simp add: scalingD
also {
    from $ht$ have $t (\lambda s. 1) \vdash \lambda s. 1$ by (auto)
    with $trm$ have $t (\lambda s. 1) = (\lambda s. 1)$ by (auto)
    hence $c * t (\lambda s. 1) s = c$ by simp
}
finally show $t (\lambda s. c) s = c$ .
qed

4.6.3 Determinism

lemma det-wp-Skip:
determ (wp Skip)
using max-intros fa-intros by (blast)

lemma det-wp-Apply:
determ (wp (Apply $f$))
by (intro determI fa-intros max-intros)

lemma det-wp-Seq:
determ (wp $a$) $\implies$ determ (wp $b$) $\implies$ well-def $b \implies$ determ (wp ($a ;; b$))
by (intro determI fa-intros max-intros, auto)

lemma det-wp-PC:
determ (wp $a$) $\implies$ determ (wp $b$) $\implies$ determ (wp ($a \oplus b$))
by (intro determI fa-intros max-intros, auto)

lemma det-wp-SetPC:
$\forall x. x \in \text{supp} (p s) \implies$ determ (wp ($a \{ x \}$))
$\forall x. \text{finite} (\text{supp} (p s)) \implies$
$\forall s. \text{sum} (p s) (\text{supp} (p s)) = 1 \implies$
determ (wp (SetPC $a \{ p \}$))
by (intro determI fa-intros max-intros, auto)

lemma det-wp-Bind:
$\forall x. \text{determ} (\text{wp} (a \{ f \{ x \} \})) \implies$ determ (wp (Bind $f$ $a$))
by (intro determI fa-intros max-intros, auto)

lemma det-wp-Embed:
determ $t \implies$ determ (wp (Embed $t$))
CHAPTER 4. THE PGCL LANGUAGE

by(simp add:wp-eval)

lemma det-wp-repeat:
determ (wp a) ⇒ well-def a ⇒ determ (wp (repeat n a))
by(intro determI fa-intros max-intros, auto)

lemmas determ-intros =
det-wp-Skip det-wp-Apply
det-wp-Seq det-wp-PC
det-wp-SetPC det-wp-Bind
det-wp-Embed det-wp-repeat
end

4.7 Well-Defined Programs.

theory WellDefined imports
  Healthiness
  Sublinearity
  LoopInduction
begin

The definition of a well-defined program collects the various notions of healthiness and well-behavedness that we have so far established: healthiness of the strict and liberal transformers, continuity and sublinearity of the strict transformers, and two new properties. These are that the strict transformer always lies below the liberal one (i.e. that it is at least as strict, recalling the standard embedding of a predicate), and that expectation conjunction is distributed between then in a particular manner, which will be crucial in establishing the loop rules.

4.7.1 Strict Implies Liberal

This establishes the first connection between the strict and liberal interpretations (wp and wlp).

definition wp-under-wlp :: 's prog ⇒ bool
where
wp-under-wlp prog ≡ ∀ P. unitary P −→ wp prog P ⊢ wp prog P

lemma wp-under-wlpI[intro]:
[ P, unitary P ⇒ wp prog P ⊢ wp prog P ] ⇒ wp-under-wlp prog
unfolding wp-under-wlp-def by(simp)

lemma wp-under-wlpD[dest]:
[ wp-under-wlp prog, unitary P ] ⇒ wp prog P ⊢ wp prog P
unfolding wp-under-wlp-def by(simp)
4.7. WELL-DEFINED PROGRAMS.

\begin{itemize}
\item \textbf{lemma wp-under-le-trans:}
\item \textit{wp-under-wlp} a $\Rightarrow$ \textit{le-utrans} (wp a) (wlp a)
\item \textbf{by (blast)}
\end{itemize}

\begin{itemize}
\item \textbf{lemma wp-under-wlp-Abort:}
\item \textit{wp-under-wlp} Abort
\item \textbf{by (rule wp-under-wlpI, unfold wp-eval, auto)}
\end{itemize}

\begin{itemize}
\item \textbf{lemma wp-under-wlp-Skip:}
\item \textit{wp-under-wlp} Skip
\item \textbf{by (rule wp-under-wlpI, unfold wp-eval, blast)}
\end{itemize}

\begin{itemize}
\item \textbf{lemma wp-under-wlp-Apply:}
\item \textit{wp-under-wlp} (Apply f)
\item \textbf{by (auto simp: wp-eval)}
\end{itemize}

\begin{itemize}
\item \textbf{lemma wp-under-wlp-Seq:}
\item \textbf{assumes h-wlp-a: nearly-healthy} (wlp a)
\item \textbf{and h-wp-b: healthy} (wp b)
\item \textbf{and h-wlp-b: nearly-healthy} (wlp b)
\item \textbf{and wp-a-a: wp-under-wlp} a
\item \textbf{and wp-a-b: wp-under-wlp} b
\item \textbf{shows wp-under-wlp} (a ;; b)
\item \textbf{proof (rule wp-under-wlpI, unfold wp-eval o-def)}
\item \textbf{fix} P :: a $\Rightarrow$ real \textbf{assume} uP : unitary P
\item \textbf{with h-wp-b} \textbf{have} anitary (wp b P) \textbf{by (blast)}
\item \textbf{with wp-a-a} \textbf{have} wp a (wp b P) $\triangleright$ wp a (wp b P) \textbf{by (auto)}
\item \textbf{also} \{ \textbf{from wp-a-b} \textbf{and uP} \textbf{have} wp a P $\triangleright$ wlp a (wp b P) \textbf{by (blast)}
\item \textbf{with h-wlp-a} \textbf{and h-wlp-b} \textbf{and h-wp-b} \textbf{and uP}
\item \textbf{have wp a (wp b P) $\triangleright$ wp a (wlp b P)}
\item \textbf{by (blast intro:nearly-healthy-monoD[OF h-wlp-a])} \}
\item \textbf{finally} \textbf{show} wp a (wp b P) $\triangleright$ wp a (wlp b P) .
\textbf{qed}
\end{itemize}

\begin{itemize}
\item \textbf{lemma wp-under-wlp-PC:}
\item \textbf{assumes h-wp-a: healthy} (wp a)
\item \textbf{and h-wlp-a: nearly-healthy} (wlp a)
\item \textbf{and h-wp-b: healthy} (wp b)
\item \textbf{and h-wlp-b: nearly-healthy} (wlp b)
\item \textbf{and wp-a-a: wp-under-wlp} a
\item \textbf{and wp-a-b: wp-under-wlp} b
\item \textbf{and uP: unitary P}
\item \textbf{shows wp-under-wlp} (a $\oplus$ b)
\item \textbf{proof (rule wp-under-wlpI, unfold wp-eval, rule le-funI)}
\item \textbf{fix Q :: a $\Rightarrow$ real} \textbf{and} s
\item \textbf{assume} uQ : unitary Q
CHAPTER 4. THE PGCL LANGUAGE

from uP have \( P s \leq 1 \) by\( \text{blast} \)
hence \( 0 \leq 1 - P s \) by\( \text{simp} \)
moreover
from uQ and \( \text{wp-u-b} \) have \( \text{wp} \ b \ Q s \leq \text{wlp} \ b \ Q s \) by\( \text{blast} \)
ultimately
have \((1 - P s) \ast \text{wp} \ b \ Q s \leq (1 - P s) \ast \text{wlp} \ b \ Q s \)
by\( \text{blast intro: mult-left-mono} \)
moreover {
from uQ and \( \text{wp-u-a} \) have \( \text{wp} \ a \ Q s \leq \text{wlp} \ a \ Q s \) by\( \text{blast} \)
with uP have \( P s \ast \text{wp} \ a \ Q s \leq P s \ast \text{wlp} \ a \ Q s \)
by\( \text{blast intro: mult-left-mono} \)
}
ultimately
show \( P s \ast \text{wp} \ a \ Q s + (1 - P s) \ast \text{wp} \ b \ Q s \leq P s \ast \text{wlp} \ a \ Q s + (1 - P s) \ast \text{wlp} \ b \ Q s \)
by\( \text{blast intro: add-mono} \)
qed

lemma \( \text{wp-under-wlp-DC} \):
assumes \( \text{wp-u-a: \ wp-under-wlp} \ a \)
and \( \text{wp-u-b: \ wp-under-wlp} \ b \)
shows \( \text{wp-under-wlp} (a \prod b) \)
proof\( (\text{rule wp-under-wlpI}, \text{unfold wp-eval}, \text{rule le-funI}) \)
fix \( Q: 'a \Rightarrow \text{real} \) and \( s \)
assume \( uQ: \text{unitary} \ Q \)
from \( \text{wp-u-a uQ have wp} \ a \ Q s \leq \text{wlp} \ a \ Q s \) by\( \text{blast} \)
moreover
from \( \text{wp-u-b uQ have wp} \ b \ Q s \leq \text{wlp} \ b \ Q s \) by\( \text{blast} \)
ultimately
show \( \text{min} (\text{wp} \ a \ Q s) (\text{wp} \ b \ Q s) \leq \text{min} (\text{wlp} \ a \ Q s) (\text{wlp} \ b \ Q s) \)
by\( \text{auto} \)
qed

lemma \( \text{wp-under-wlp-SetPC} \):
assumes \( \text{wp-u-f: } \bigwedge \ a. \ a \in \text{supp} (P s) \implies \text{wp-under-wlp} (f \ a) \)
and \( nP: \bigwedge \ a. \ a \in \text{supp} (P s) \implies 0 \leq P s a \)
shows \( \text{wp-under-wlp} (\text{SetPC} f P) \)
proof\( (\text{rule wp-under-wlpI}, \text{unfold wp-eval}, \text{rule le-funI}) \)
fix \( Q: 'a \Rightarrow \text{real} \) and \( s \)
assume \( uQ: \text{unitary} \ Q \)
from \( \text{wp-u-f uQnP} \)
show \( \sum a \in \text{supp} (P s). \ P s a \ast \text{wp} (f \ a) \ Q s \leq \sum a \in \text{supp} (P s). \ P s a \ast \text{wlp} (f \ a) \ Q s \)
by\( \text{auto intro!: sum-mono mult-left-mono} \)
qed
4.7. WELL-DEFINED PROGRAMS.

**Lemma** `wp-under-wlp-SetDC`:

**Assumes**

wp-u-f: \( \forall s. a \in S \Rightarrow \text{wp-under-wlp} (f a) \)

and hf: \( \forall s. a \in S \Rightarrow \text{healthy} (wp (f a)) \)

and nS: \( \forall s. S s \neq \{\} \)

**Shows** `wp-under-wlp (SetDC f S)`

**Proof** (rule `wp-under-wlpI`, rule `le-funI`, unfold `wp-eval`)

fix `Q::'a \Rightarrow real` and `s`

assume `uQ`: unitary `Q`

**Show** `Inf ((\lambda a. wp (f a) Q s) ' S s) \leq Inf ((\lambda a. wlp (f a) Q s) ' S s)`

**Proof** (rule `cInf-mono`)

from `nS` show `((\lambda a. wlp (f a) Q s) ' S s) \neq \{}` by (blast)

fix `x` assume `xin`: `x \in (\lambda a. wlp (f a) Q s) ' S s`

then obtain `a` where `ain`: `a \in S s` and `xrw: x = wlp (f a) Q s`

by (blast)

with `wp-u-f uQ`

have `wp (f a) Q s \leq wlp (f a) Q s` by (blast)

moreover from `ain` have `wp (f a) Q s \in (\lambda a. wp (f a) Q s) ' S s`

by (blast)

ultimately show `\exists y\in (\lambda a. wp (f a) Q s) ' S s. y \leq x`

by (auto simp: `xrw`)

**Next**

fix `y` assume `yin`: `y \in (\lambda a. wp (f a) Q s) ' S s`

then obtain `a` where `ain`: `a \in S s` and `yrw: y = wp (f a) Q s`

by (blast)

with `hf aQ` have `unitary (wp (f a) Q)` by (auto)

with `yrw` show `0 \leq y` by (auto)

qed

**Lemma** `wp-under-wlp-Embed`:

wp-under-wlp (Embed `t`)

by (rule `wp-under-wlpI`, unfold `wp-eval`, blast)

**Lemma** `wp-under-wlp-loop`:

fixes `body::'s prog`

**Assumes**

`hwp`: healthy (wp `body`)

and `hwlp`: nearly-healthy (wp `body`)

and `wp-under`: wp-under-wlp body

**Shows** `wp-under-wlp (do G \rightarrow body od)`

**Proof** (rule `wp-under-wlpI`)

fix `P::'s expect`

assume `uP`: unitary `P` hence `sP`: sound `P` by (auto)

let `?X Q s = "G" s * wp body Q s + "N G" s * P s`

let `?Y Q s = "G" s * wlp body Q s + "N G" s * P s`
show \( \text{wp (do } G \rightarrow \text{ body od} ) \ P \vdash \text{wlwp (do } G \rightarrow \text{ body od} ) \ P \)

**proof** (simp add: hwp wlwp sP uP wp-Loop1 wlwp-Loop1, rule gfp-exp-upperbound)

thm lfp-loop-fp

from hwp sP have lfp-exp \(?X = ?X \cdot \text{(lfp-exp } ?X)\)

by (rule lfp-wp-loop-unfold)

hence lfp-exp ?X \(\vdash ?X \cdot \text{(lfp-exp } ?X)\) by (simp)

also {

from hwp uP have wp body (lfp-exp ?X) \(\vdash \text{wlwp body } \cdot \text{(lfp-exp } ?X)\)

by (auto intro: wp-under-wlpD [OF wp-under lfp-loop-unitary])

hence ?X \(\vdash ?X \cdot \text{(lfp-exp } ?X)\) by (auto intro: add-mono mult-left-mono)

}

finally show lfp-exp \(?X \vdash ?Y \cdot \text{(lfp-exp } ?X)\).

from hwp uP show unitary (lfp-exp ?X)

by (auto intro: lfp-loop-unitary)

qed

qed

**lemma** wp-under-wlp-repeat:

\[ \text{healthy } (\text{wp } a) ; \text{nearly-healthy } (\text{wlwp } a) ; \text{wp-under-wlp } a \]

\(\Rightarrow\) wp-under-wlp (repeat \(n a\))

by (induct \(n\), auto intro!: wp-under-wlp-Skip wp-under-wlp-Seq healthy-intros)

**lemma** wp-under-wlp-Bind:

\[ \text{\(\forall s. \text{wp-under-wlp } (a (f s))\)} \]

\(\Rightarrow\) wp-under-wlp (Bind \(f a\))

unfolding wp-under-wlp-def by (auto simp: wp-eval)

**lemmas** wp-under-wlp-intros =

wp-under-wlp-Abort wp-under-wlp-Skip
wp-under-wlp-Apply wp-under-wlp-Seq
wp-under-wlp-PC wp-under-wlp-DC
wp-under-wlp-SetPC wp-under-wlp-SetDC
wp-under-wlp-Embed wp-under-wlp-loop
wp-under-wlp-repeat wp-under-wlp-Bind

### 4.7.2 Sub-Distributivity of Conjunction

definition

sub-distrib-pconj :: \(\forall s \text{ prog } \Rightarrow \text{ bool}\)

**where**

(sub-distrib-pconj prog \(\equiv\)

\(\forall P Q. \text{ unitary } P \rightarrow \text{ unitary } Q \rightarrow \text{wlwp prog } P \&\& \text{wp prog } Q \vdash \text{wp prog } (P \&\& Q)\)

**lemma** sub-distrib-pconj[intro]:

\[\forall P Q. \text{ unitary } P; \text{ unitary } Q \rightarrow \text{wlwp prog } P \&\& \text{wp prog } Q \vdash \text{wp prog } (P \&\& Q)\]

\(\Rightarrow\) sub-distrib-pconj prog
4.7. WELL-DEFINED PROGRAMS.

unfolding \textit{sub-distrib-pconj-def} \textbf{by}(simp)

\textbf{lemma} \textit{sub-distrib-pconjD\{dest\}}:
\[ \forall P Q. \lbrack \text{sub-distrib-pconj \ prog; unitary \ P; unitary \ Q} \rbrack \implies wp \ prog \ P \&\& wp \ prog \ Q \iff wp \ prog \ (P \&\& Q) \]

\textbf{unfolding} \textit{sub-distrib-pconj-def} \textbf{by}(simp)

\textbf{lemma} \textit{sdp-Abort}:
\textit{sub-distrib-pconj \ Abort} \\
\textbf{by}(rule \ sub-distrib-pconjI, unfold \ wp-eval, auto \ intro:exp-conj-rzero)

\textbf{lemma} \textit{sdp-Skip}:
\textit{sub-distrib-pconj \ Skip} \\
\textbf{by}(rule \ sub-distrib-pconjI, simp \ add:wp-eval)

\textbf{lemma} \textit{sdp-Seq}: \\
\textbf{fixes} a and b \\
\textbf{assumes} \textit{sdp-a}: \textit{sub-distrib-pconj} a \\
\textbf{and} \textit{sdp-b}: \textit{sub-distrib-pconj} b \\
\textbf{and} \textit{h-wp-a}: \textit{healthy} (wp a) \\
\textbf{and} \textit{h-wp-b}: \textit{healthy} (wp b) \\
\textbf{and} \textit{h-wlp-b}: \textit{nearly-healthy} (wlp b) \\
\textbf{shows} \textit{sub-distrib-pconj} (a ;; b) \\
\textbf{proof}(rule \ sub-distrib-pconjI, unfold \ wp-eval \ a-def)
\textbf{fix} P::'a \Rightarrow real and Q::'a \Rightarrow real \\
\textbf{assume} uP: \textit{unitary} P and uQ: \textit{unitary} Q \\
\textbf{with} \textit{h-wp-b} and \textit{h-wlp-b} \\
\textbf{have} wp a (wp b P) \&\& wp a (wp b Q) \iff wp a (wp b P \&\& wp b Q) \\
\textbf{proof}(intro!:sub-distrib-pconjD[OF sdp-a])
\textbf{also} \{ \\
\textbf{from} \textit{sdp-b} \textit{and} \textit{uP} \textit{and} \textit{uQ} \\
\textbf{have} wp b P \&\& wp b Q \iff wp b (P \&\& Q) \textbf{by}(blast) \\
\textbf{with} \textit{h-wp-a} \textit{h-wp-b} \textit{h-wlp-b} \textit{uP} \textit{uQ} \\
\textbf{have} \textit{wp} a (wp b P \&\& wp b Q) \iff \textit{wp} a (wp b (P \&\& Q)) \\
\textbf{proof}(blast \ intro!:mono-transD[OF \textit{healthy-monoD}, \textit{OF} \textit{h-wp-a}] \textit{unitary-sound} \textit{unitary-intros} \textit{sound-intros}) \\
\} \\
\textbf{finally} \textbf{show} \textit{wp} a (wp b P) \&\& wp a (wp b Q) \iff wp a (wp b (P \&\& Q)) . \\
\textbf{qed}

\textbf{lemma} \textit{sdp-Apply}:
\textit{sub-distrib-pconj} (Apply f) \\
\textbf{by}(rule \ sub-distrib-pconjI, simp \ add:wp-eval)

\textbf{lemma} \textit{sdp-DC}:
\textbf{fixes} a::'s \textit{prog} and b \\
\textbf{assumes} \textit{sdp-a}: \textit{sub-distrib-pconj} a \\
\textbf{and} \textit{sdp-b}: \textit{sub-distrib-pconj} b
and \(h\text{-wp-a:} \text{healthy (wp a)}\)
and \(h\text{-wp-b:} \text{healthy (wp b)}\)
and \(h\text{-wp-b:} \text{nearly-healthy (wp b)}\)

shows \(\text{sub-distrib-pconj} (a \sqcap b)\)

proof\(\text{(rule sub-distrib-pconjI, unfold wp-eval, rule le-funI)}\)

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

have \(((\lambda s. \text{min (wp a P s) (wp b P s)}) \&\&
(\lambda s. \text{min (wp a Q s) (wp b Q s)}))\ s \leq
\text{min (wp a P s \& wp a Q s) (wp b P s \& wp b Q s)}\)

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

also \{ 
    have \(((\lambda s. \text{wp a P s \& wp a Q s)} = \text{wp a P \&\& wp a Q)}\)
    by(simp add:exp-conj-def)
    also from \(\text{sdp-a uP uQ have ... \text{ wp a (P \&\& Q)}\)
    by(blast dest:sub-distrib-pconjD)
    finally have \(\text{wp a P s \& wp a Q s} \leq \text{wp a (P \&\& Q) s}\)
    by(rule le-funD)

    moreover \{ 
        have \(((\lambda s. \text{wp b P s \& wp b Q s)} = \text{wp b P \&\& wp b Q)}\)
        by(simp add:exp-conj-def)
        also from \(\text{sdp-b uP uQ have ... \text{ wp b (P \&\& Q)}\)
        by(blast)
        finally have \(\text{wp b P s \& wp b Q s} \leq \text{wp b (P \&\& Q) s}\)
        by(rule le-funD)
    \}

    ultimately
    have \text{min (wp a P s \& wp a Q s) (wp b P s \& wp b Q s)} \leq
    \text{min (wp a (P \&\& Q) s) (wp b (P \&\& Q) s) by(auto)\}
\}
finally
show \(((\lambda s. \text{min (wp a P s) (wp b P s)}) \&\&
(\lambda s. \text{min (wp a Q s) (wp b Q s)}))\ s \leq
\text{min (wp a (P \&\& Q) s) (wp b (P \&\& Q) s)}\).

qed

lemma \(\text{sdp-PC:}\)
fixes \(a::'s \text{ prog and } b\)
assumes \(\text{sdp-a: sub-distrib-pconj a and sdp-b: sub-distrib-pconj b}\)
and \(h\text{-wp-a: healthy (wp a)}\)
and \(h\text{-wp-b: healthy (wp b)}\)
and \(h\text{-wp-b: nearly-healthy (wp b)}\)
and \(uP::\text{unitary P}\)
shows \(\text{sub-distrib-pconj} (a \oplus b)\)

proof\(\text{(rule sub-distrib-pconjI, unfold wp-eval, rule le-funI)}\)

fix \(Q::'s \Rightarrow \text{real and } R::'s \Rightarrow \text{real and } s::'s\)
assume \(uQ::\text{unitary Q and } uR::\text{unitary R}\)
4.7. WELL-DEFINED PROGRAMS.

have \( nnA: 0 \leq P s \) and \( nnB: P s \leq 1 \)
using \( uP \) by(auto simp:sign-simps)
note \( nn \equiv nnA \ nnB \)

have \((\lambda s. \ P s * \ wp a \ Q s + (1 - P s) * \ wp b \ Q s) \land\land\)
\((\lambda s. \ P s * \ wp a \ R s + (1 - P s) * \ wp b \ R s)) s =
\((P s * \ wp a \ Q s + (1 - P s) * \ wp b \ Q s) +\)
\((P s * \ wp a \ R s + (1 - P s) * \ wp b \ R s)) \ominus 1\)
by(simp add:exp-conj-def pconj-def)
also have \( \ldots = P s * \ (wp a \ Q s + wp a \ R s) +\)
\((1 - P s) * \ (wp b \ Q s + wp b \ R s) \ominus 1\)
by(simp add:field-simps)
also have \( \ldots = P s * \ (wp a \ Q s + wp a \ R s) +\)
\((1 - P s) * \ (wp b \ Q s + wp b \ R s) \ominus\)
\((P s + (1 - P s))\)
by(simp)
also have \( \ldots \leq (P s * \ (wp a \ Q s + wp a \ R s) \ominus P s) +\)
\(((1 - P s) * \ (wp b \ Q s + wp b \ R s) \ominus (1 - P s))\)
by(rule tminus-add-mono)
also have \( \ldots = (P s * \ (wp a \ Q s + wp a \ R s \ominus I)) +\)
\(((1 - P s) * \ (wp b \ Q s + wp b \ R s \ominus I))\)
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 \( sdp-a \ sdp-b \ uQ \ aR \) \n\begin{align*}
\text{have } & P s * (wp a \ Q \land\land \ wp a \ R) s \leq P s * \ wp a (Q \land\land R) s \\
& \text{and } (1 - P s) * (wp b \ Q \land\land \ wp b \ R) s \leq (1 - P s) * \ wp b (Q \land\land R) s \\
& \text{by } (simp-all add: entailsD mult-left-mono nn sub-distrib-pconjD) \\
\text{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 \\
& \text{by (auto)}
\end{align*}
\}
finally show \((\lambda s. \ P s * \ wp a \ Q s + (1 - P s) * \ wp b \ Q s) \land\land\)
\((\lambda s. \ P s * \ wp a \ R s + (1 - P s) * \ wp b \ R s)) s \leq
\((P s * \ wp a (Q \land\land R) s + (1 - P s) * \ wp b (Q \land\land R) s)\).

qed

lemma \textit{sdp-Embed}:
\[
[ A \ P, Q, \ \text{unitary } P; \ \text{unitary } Q ] \Rightarrow t \ P \land\land \ t \ Q + \ t (P \land\land Q) \]
\Rightarrow \text{sub-distrib-pconj (Embed t)}
by(auto simp:wp-eval)

lemma \textit{sdp-repeat}:
\text{fixes } a::'s \ \text{prog}
\text{assumes } \textit{sdp}: \text{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 hwlp: nearly-healthy (wlp (repeat n a))
    and hwp: healthy (wp (repeat n a))
    by (auto intro: healthy-intros)
moreover from uP and hwlp have unitary (wlp (repeat n a) P) by (blast)
ultimately have wp a (wp (repeat n a) P) & & wp a (wp (repeat n a) Q) \=" wp a (wp (repeat n a) P) & & wp (repeat n a) Q) by (blast)
using sdp by (blast)
also {
  from hwlp have nearly-healthy (wlp (repeat n a)) by (rule healthy-intros)
  with uP have sound (wlp (repeat n a) P) by (auto)
  moreover from hwlp uQ have sound (wp (repeat n a) Q)
    by (auto intro: healthy-intros)
  ultimately have sound (wlp (repeat n a) P & & wp (repeat n a) Q)
    by (rule exp-conj-sound)
  moreover {
    from uP uQ have sound (P & & Q) by (auto intro: exp-conj-sound)
    with hwlp 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 \=" wp (repeat n a) (P & & Q)
  by (blast)
ultimately have wp a (wp (repeat n a) P & & wp (repeat n a) Q) \=" wp a (wp (repeat n a) (P & & Q))
  by (rule mono-transD[OF healthy-monoD, OF hwlp])}
finally show wp a (wp (repeat n a) P) & & wp a (wp (repeat n a) Q) \=" wp a (wp (repeat n a) (P & & Q))
qed

lemma sdp-SetPC:
fixes p :: 'a ⇒ 's prog
assumes sdp: \(\forall s. a \in \text{supp} (P s) \implies \text{sub-distrib-pconj (p a)}\)
    and fin: \(\forall s. \text{finite (supp (P s))}\)
    and nnp: \(\forall s. a. 0 \leq P s a\)
    and sub: \(\forall s. \text{sum (P s) (supp (P s))} \leq 1\)
4.7. WELL-DEFINED PROGRAMS.

shows \texttt{sub-distrib-pconj (SetPC p P)}

proof (rule sub-distrib-pconjI, simp add:wp-eval, rule le-funI)

fix \texttt{Q:’s ⇒ real and R:’s ⇒ real and s:’s}

assume \texttt{uQ: unitary Q and uR: unitary R}

have \((\lambda s. \sum a\in\text{supp} (P s). P s a * wp (p a) Q s) \&\&
          (\lambda s. \sum a\in\text{supp} (P s). P s a * wp (p a) R s)) s =
          (\sum a\in\text{supp} (P s). P s a * wp (p a) Q s) + (\sum a\in\text{supp} (P s). P s a * wp (p a) R s)\)
          \&\& 1
        by (simp add:exp-conj-def pconj-def)

also have \((\sum a\in\text{supp} (P s). P s a * wp (p a) Q s + wp (p a) R s)) \&\& 1
        by (simp add: sum.distrib field-simps)

also from sub

have \((\sum a\in\text{supp} (P s). P s a * (wp (p a) Q s + wp (p a) R s)) \&\&
          (\sum a\in\text{supp} (P s). P s a))
        by (rule tminus-right-antimono)

also from fin

have \((\sum a\in\text{supp} (P s). P s a * (wp (p a) Q s + wp (p a) R s)) \&\& P s a)
        by (rule tminus-sum-mono)

also from nnp

have \((\sum a\in\text{supp} (P s). P s a * (wp (p a) Q s + wp (p a) R s \&\& 1))
        by (simp add: tminus-left-distrib)

also have \((\sum a\in\text{supp} (P s). P s a * (wp (p a) Q \&\& wp (p a) R) s)
        by (simp add: pconj-def exp-conj-def)

also \{
  from \texttt{sdp uQ aR}
  have \((\forall a. a \in \text{supp} (P s) \implies wp (p a) Q \&\& wp (p a) R) \implies wp (p a) (Q \&\& R))
          by (blast intro: sub-distrib-pconjD)
        with nnp
        have \((\sum a\in\text{supp} (P s). P s a * (wp (p a) Q \&\& wp (p a) R) s) \leq
          (\sum a\in\text{supp} (P s). P s a * (wp (p a) (Q \&\& R)) s)
        by (blast intro: sum-mono mult-left-mono)
    \}

finally show \((\lambda s. \sum a\in\text{supp} (P s). P s a * wp (p a) Q s) \&\&
          (\lambda s. \sum a\in\text{supp} (P s). P s a * wp (p a) R) s) \leq
          (\sum a\in\text{supp} (P s). P s a * wp (p a) (Q \&\& R) s) .

qed

lemma \texttt{sdp-SetDC:}

fixes \texttt{p:’a ⇒ ’s prog}

assumes \texttt{sdp:} \(\forall s. a \in S s \implies \text{sub-distrib-pconj} (p a)

and \texttt{hp:} \(\forall s. a \in S s \implies \text{healthy} (wp (p a))

and \texttt{hwp:} \(\forall s. a \in S s \implies \text{nearly-healthy} (wp (p a))

and \texttt{nc:} \(\forall s. S s \neq \{\}

shows \texttt{sub-distrib-pconj (SetDC p S)}

proof (rule sub-distrib-pconjI, rule le-funI)

fix \texttt{P:’s ⇒ real and Q:’s ⇒ real and s:’s}

assume \texttt{uP: unitary P and uQ: unitary Q}
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:**

**Proof:**

1. **Fixes** `body :: 's prog`
2. **Assumes** `sdp-body : sub-distrib-pconj body`
   - and `hwlp : nearly-healthy (wp body)`
   - and `hwp : healthy (wp body)`
3. **Shows** `sub-distrib-pconj (do G -> body od)`

**Proof:** (rule `sub-distrib-pconjI`, rule `loop-induct` (`OF hwp hwlp`))

**Assume** `uP : unitary P` and `uQ : unitary Q`

- and `fst : ∀ x ∈ S. feasible (fst x)`
- and `snd : ∀ x ∈ S. ∀ Q. unitary Q → unitary (snd x Q)`
- and `IH : ∀ x ∈ S. snd x P & & fst x Q ⊢ fst x (P & & Q)`
4.7. WELL-DEFINED PROGRAMS.

\[ \text{show } \text{Inf-utrans} (\text{snd} \ ' S) \ P \land \text{Sup-trans} (\text{fst} \ ' S) \ Q \vdash \\
\text{Sup-trans} (\text{fst} \ ' S) \ (P \land Q) \]

\textbf{proof(cases)}

\begin{itemize}
  \item assume \( S = \{\} \)
  \begin{itemize}
    \item \text{thus} ?\text{thesis}
      \begin{itemize}
        \item by(simp add:Inf-trans-def Sup-trans-def Inf-utrans-def
          \text{Inf-exp-def Sup-exp-def exp-conj-def})
      \end{itemize}
  \end{itemize}

  \item next
    \begin{itemize}
      \item assume \( \neq S \neq \{\} \)
      \begin{itemize}
        \item let \(?f s = 1 + \text{Sup-trans} (\text{fst} \ ' S) \ (P \land Q) s - \text{Inf-utrans} (\text{snd} \ ' S) \ P s\)
        \item from \( \neq \) obtain \( t \) where \( \text{tin: } t \in \text{fst} \ ' S \) by(auto)
        \item from \( \neq \) obtain \( u \) where \( \text{uin: } u \in \text{snd} \ ' S \) by(auto)
        \item from \( \text{tin} \ \text{ffst} \ \text{uP uQ} \) have \( \land s. \ 0 \leq t (P \land Q) s \) by(auto)
        \begin{itemize}
          \item also \{ \begin{itemize}
              \item from \( \text{ffst} \ \text{tin} \ \text{have} \ \text{le: } \text{le-utrans} t (\text{Sup-trans} (\text{fst} \ ' S)) \)
              \item by(auto intro:Sup-trans-upper)
              \item with \( uP uQ \) have \( \land s. \ t (P \land Q) s \leq \text{Sup-trans} (\text{fst} \ ' S) (P \land Q) s \)
              \item by(auto intro:exp-conj-unitary)
          \end{itemize}
        \} \end{itemize}
        \item finally have \( \text{nn-rhs: } \land s. \ 0 \leq \text{Sup-trans} (\text{fst} \ ' S) (P \land Q) s \).
        \item have \( \land R. \ \text{Inf-utrans} (\text{snd} \ ' S) \ P \land R \vdash \text{Sup-trans} (\text{fst} \ ' S) (P \land Q) \implies R \leq ?f \)
        \item proof(rule contrapos-pp, assumption)
          \begin{itemize}
            \item fix \( R \)
            \item assume \( \neg R \leq ?f \)
            \item then obtain \( s \) where \( \neg R s \leq ?f s \) by(auto)
            \item hence \( gt: \ ?f s < R s \) by(simp)
          \end{itemize}
        \item from \( \text{nn-rhs} \) have \( g1: \ 1 \leq 1 + \text{Sup-trans} (\text{fst} \ ' S) (P \land Q) s \) by(auto)
        \item hence \( \text{Sup-trans} (\text{fst} \ ' S) (P \land Q) s = \text{Inf-utrans} (\text{snd} \ ' S) P s \land ?f s \)
          \begin{itemize}
            \item by(simp add:pcconj-def)
          \end{itemize}
        \item also from \( g1 \) have \( .. = \text{Inf-utrans} (\text{snd} \ ' S) P s + ?f s - 1 \)
          \begin{itemize}
            \item by(simp)
          \end{itemize}
        \item also from \( gt \) have \( .. < \text{Inf-utrans} (\text{snd} \ ' S) P s + R s - 1 \)
          \begin{itemize}
            \item by(simp)
          \end{itemize}
        \item also \{ \begin{itemize}
              \item with \( g1 \) have \( 1 \leq \text{Inf-utrans} (\text{snd} \ ' S) P s + R s \)
              \item by(simp)
              \item hence \( \text{Inf-utrans} (\text{snd} \ ' S) P s + R s - 1 = \text{Inf-utrans} (\text{snd} \ ' S) P s \land R s \)
              \item by(simp add:pcconj-def)
          \end{itemize}
        \} \end{itemize}
      \end{itemize}
    \end{itemize}
\end{itemize}
have \((\text{Inf-utrans} \ (\text{snd} \ S) \ P \ \&\& \ R) \ s \leq \text{Sup-trans} \ (\text{fst} \ S) \ (P \ \&\& \ Q) \ s\)
by(simp add:exp-conj-def)

thus \((\text{Inf-utrans} \ (\text{snd} \ S) \ P \ \&\& \ R) \not\vdash \text{Sup-trans} \ (\text{fst} \ S) \ (P \ \&\& \ Q)\)
by(auto)

qed

moreover have \(\forall t \in \text{fst} \ S. \ \text{Inf-utrans} \ (\text{snd} \ S) \ P \ \&\& \ t Q \not\vdash \text{Sup-trans} \ (\text{fst} \ S) \ (P \ \&\& \ Q)\)
proof
fix \(t\)
assume \(t \in \text{fst} \ S\)
then obtain \(x\)
where \(x \in S\) and \(fx: \ t = \text{fst} \ x\)
by(auto)

from \(xin\) have \(\text{snd} \ x \in \text{snd} \ S\)
by(auto)

with \(uP\) usnd have \(\text{Inf-utrans} \ (\text{snd} \ S) \ P \not\vdash \text{snd} \ x \ P\)
by(auto intro:le-utransD[OF Inf-utrans-lower])

hence \(\text{Inf-utrans} \ (\text{snd} \ S) \ P \ \&\& \ \text{fst} \ x \ Q \not\vdash \text{snd} \ x \ P \ \&\& \ \text{fst} \ x \ Q\)
by(auto intro:entails-frame)

also from \(xin\) IH have \(...\not\vdash \text{fst} \ x \ (P \ \&\& \ Q)\)
by(auto)

also from \(xin\) \(\text{fist} \ \text{exp-conj-unitary}[OF uP uQ]\)
have \(...\not\vdash \text{Sup-trans} \ (\text{fst} \ S) \ (P \ \&\& \ Q)\)
by(auto intro:le-utransD[OF Sup-trans-upper])

finally show \(\text{Inf-utrans} \ (\text{snd} \ S) \ P \ \&\& \ t Q \not\vdash \text{Sup-trans} \ (\text{fst} \ S) \ (P \ \&\& \ Q)\)
by(simp add:fx)

qed
ultimately have \(bt: \forall t \in \text{fst} \ S. \ t Q \not\vdash \ ? f\)
by(blast)

have \(\text{Sup-trans} \ (\text{fst} \ S) \ Q = \text{Sup-exp} \ \{t \ Q | t, t \in \text{fst} \ S\}\)
by(simp add:Sup-trans-def)

also have \(...\not\vdash \ ? f\)
proof(rule Sup-exp-least)
from \(bt\) show \(\forall R \in \{t \ Q | t, t \in \text{fst} \ S\}. \ R \not\vdash \ ? f\)
by(blast)

from \(ne\) obtain \(t\)
where \(tin: \ t \in \text{fst} \ S\)
by(auto)

with \(\text{fist} uQ\) have \(\text{unitary} \ (t \ Q)\)
by(auto)

hence \(\lambda s. 0 \not\vdash \ t \ Q\)
by(auto)

also from \(tin\) \(bt\) have \(...\not\vdash \ ? f\)
by(auto)

finally show \(\text{mneg} \ (\lambda s. 1 + \text{Sup-trans} \ (\text{fst} \ S) \ (P \ \&\& \ Q) \ s)\)
by(auto)

qed
finally have \(\text{Inf-utrans} \ (\text{snd} \ S) \ P \ \&\& \ \text{Sup-trans} \ (\text{fst} \ S) \ Q \not\vdash \ ? f\)
by(auto intro:entails-frame)

also from \(\text{nn-rhs}\) have \(...\not\vdash \text{Sup-trans} \ (\text{fst} \ S) \ (P \ \&\& \ Q)\)
by(simp add:exp-conj-def pconj-def)

finally show \(? \text{thesis}\).
qed

next
4.7. WELL-DEFINED PROGRAMS.

\begin{verbatim}
fix P Q: 's expect and t u: 's trans
assume uP: unitary P and uQ: unitary Q
and ft: feasible t
and uu: \( \forall Q. \) unitary Q \( \Rightarrow \) unitary (u Q)
and IH: u P && t Q \( \vdash \) t (P && Q)
show wp (body :: Embed u \( \cdot \) G, Skip) P &&
  wp (body :: Embed t \( \cdot \) G, Skip) Q \( \vdash \)
  wp (body :: Embed t \( \cdot \) G, Skip) (P && Q)
proof
  (rule le-fun1, simp add: wp-eval exp-conj-def pconj-def)
  fix s: 's
  have \( \langle G \rangle \) s * wlp body (u P) s + (1 - \( \langle G \rangle \) s) * P s +
    \( \langle G \rangle \) s * wlp body (t Q) s + (1 - \( \langle G \rangle \) s) * Q s \( \vdash \) 1 =
    \( \langle G \rangle \) s * wlp body (u P) s + \( \langle G \rangle \) s * wlp body (t Q) s +
    \( \langle 1 - \langle G \rangle \rangle \) s * P s + (1 - \( \langle G \rangle \) s) * Q s \( \vdash \) (\( \langle G \rangle \) s + (1 - \( \langle G \rangle \) s))
  by (simp add: ac-simps)
  also have \( \ldots \) \( \leq \)
    \( \langle G \rangle \) s * wlp body (u P) s + \( \langle G \rangle \) s * wlp body (t Q) s \( \vdash \) \( \langle G \rangle \) s +
    \( \langle 1 - \langle G \rangle \rangle \) s * P s + (1 - \( \langle G \rangle \) s) * Q s \( \leq \) (\( \langle G \rangle \) s)
  by (rule tminus-add-mono)
  also have \( \ldots = \)
    \( \langle G \rangle \) s * (wlp body (u P) s + wlp body (t Q) s \( \vdash \) 1) +
    (1 - \( \langle G \rangle \) s) * (P s + Q s \( \vdash \) 1)
  by (simp add: tminus-left-distrib distrib-left)
also \{
  from uP uQ ft uu
  have wlp body (u P) \&\& wlp body (t Q) \( \vdash \) wlp body (u P \&\& t Q)
    by (auto intro: sub-distrib-peonjD (OF sdp-body))
  also from IH unitary-sound (OF uP) unitary-sound (OF uQ) ft
    unitary-sound (OF uu (OF uP))
  have \( \ldots \) \( \leq \) wlp body (t (P \&\& Q))
    by (blast intro: mono-transD (OF healthy-monoD, OF huq) exp-conj-sound)
  finally have wlp body (u P) s + wlp body (t Q) s \( \vdash \) 1 \( \leq \)
    wlp body (t (\( \lambda s. \) P s + Q s \( \vdash \) 1)) s
  by (auto simp: exp-conj-def pconj-def)
  hence \( \langle G \rangle \) s * (wlp body (u P) s + wlp body (t Q) s \( \vdash \) 1) +
    (1 - \( \langle G \rangle \) s) * (P s + Q s \( \vdash \) 1) \( \leq \)
    \( \langle G \rangle \) s * wlp body (t (\( \lambda s. \) P s + Q s \( \vdash \) 1)) s +
    (1 - \( \langle G \rangle \) s) * (P s + Q s \( \vdash \) 1)
  by (auto intro: add-right-mono mult-left-mono)
}\)
finally
show \( \langle G \rangle \) s * wlp body (u P) s + (1 - \( \langle G \rangle \) s) * P s +
  \( \langle G \rangle \) s * wlp body (t Q) s + (1 - \( \langle G \rangle \) s) * Q s \( \vdash \) 1 \( \leq \)
  \( \langle G \rangle \) s * wlp body (t (\( \lambda s. \) P s + Q s \( \vdash \) 1)) s +
  (1 - \( \langle G \rangle \) s) * (P s + Q s \( \vdash \) 1).
qed

next
fix P Q: 's expect and t u: 't's trans
assume unitary P unitary Q
\end{verbatim}
equiv-trans \( t \) \( t' \) equiv-utrans \( u \) \( u' \)
\( u \ P \&\& t \ Q \vdash t \ (P \&\& Q) \)
thus \( u' \ P \&\& t' \ Q \vdash t' \ (P \&\& Q) \)
by (simp add: equiv-transD unitary-sound equiv-utransD exp-conj-unitary)
qed

lemmas sdp-intros =
  sdp-Abort sdp-Skip sdp-Apply
  sdp-Seq sdp-DC sdp-PC
  sdp-SetPC sdp-SetDC sdp-Embed
  sdp-repeat sdp-Bind sdp-loop

4.7.3 The Well-Defined Predicate.

definition well-def :: '\ s \ prog \Rightarrow \ bool
where
  well-def \ prog \equiv \begin{array}{c}
  \text{healthy} (\ wp \ prog) \land \ \text{nearly-healthy} (\ wlp \ prog) \\
  \land \ \wp\text{-under-wlp} \ prog \land \ \text{sub-distrib-pconj} \ prog \\
  \land \ \text{sublinear} (\ wp \ prog) \land \ \text{bd-cts} (\ wp \ prog)
\end{array}

lemma well-defI[intro]:
  \[ \begin{array}{l}
  \text{healthy} (\ wp \ prog); \ \text{nearly-healthy} (\ wlp \ prog); \\
  \wp\text{-under-wlp} \ prog; \ \text{sub-distrib-pconj} \ prog; \ \text{sublinear} (\ wp \ prog); \\
  \text{bd-cts} (\ wp \ prog) \end{array} \Rightarrow \ well-def \ prog
\]
unfolding well-def_def by (simp)

lemma well-def-wp-healthy[dest]:
  well-def \ prog \Longrightarrow \ \text{healthy} (\ wp \ prog)
unfolding well-def_def by (simp)

lemma well-def-wlp-nearly-healthy[dest]:
  well-def \ prog \Longrightarrow \ \text{nearly-healthy} (\ wlp \ prog)
unfolding well-def_def by (simp)

lemma well-def-wp-under[dest]:
  well-def \ prog \Longrightarrow \ \wp\text{-under-wlp} \ prog
unfolding well-def_def by (simp)

lemma well-def-sdp[dest]:
  well-def \ prog \Longrightarrow \ \text{sub-distrib-pconj} \ prog
unfolding well-def_def by (simp)

lemma well-def-wp-sublinear[dest]:
  well-def \ prog \Longrightarrow \ \text{sublinear} (\ wp \ prog)
unfolding well-def_def by (simp)

lemma well-def-wp-cts[dest]:
well-def prog ⇒ bd-cts (wp prog)

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

**lemmas** wd-dests =
well-def-wp-healthy well-def-wlp-nearly-healthy
well-def-wp-under well-def-sdp
well-def-wp-sublinear well-def-wp-cts

**lemma** wd-Abort:
well-def Abort
by(blast intro:healthy-wp-Abort nearly-healthy-wlp-Abort
 wp-under-wlp-Abort sdp-Abort sublinear-wp-Abort
 cts-wp-Abort)

**lemma** wd-Skip:
well-def Skip
by(blast intro:healthy-wp-Skip nearly-healthy-wlp-Skip
 wp-under-wlp-Skip sdp-Skip sublinear-wp-Skip
 cts-wp-Skip)

**lemma** wd-Apply:
well-def (Apply f)
by(blast intro:healthy-wp-Apply nearly-healthy-wlp-Apply
 wp-under-wlp-Apply sdp-Apply sublinear-wp-Apply
 cts-wp-Apply)

**lemma** wd-Seq:
[ well-def a; well-def b ] ⇒ well-def (a ;; b)
by(blast intro:healthy-wp-Seq nearly-healthy-wlp-Seq
 wp-under-wlp-Seq sdp-Seq sublinear-wp-Seq
 cts-wp-Seq)

**lemma** wd-PC:
[ well-def a; well-def b; unitary P ] ⇒ well-def (a p⪈ 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 ] ⇒ well-def (a ∩ b)
by(blast intro:healthy-wp-DC nearly-healthy-wlp-DC
 wp-under-wlp-DC sdp-DC sublinear-wp-DC
 cts-wp-DC)

**lemma** wd-SetDC:
[ x s. x ∈ S s ⇒ well-def (a x); \( \forall s. S s \neq {} \); \( \forall s. \text{finite} (S s) \) ] ⇒ well-def (SetDC a S)
by(simp add: cts-wp-SetDC ex-in-conv healthy-intros(17) healthy-intros(18) sdp-intros(8)
 sublinear-intros(8) well-def-def wp-under-wlp-intros(8))
lemma \textit{wd-SetPC}:
\[
\begin{aligned}
\forall s. x \in (\text{supp} (p s)) &\implies \text{well-def} (a x); \\
\forall s. \text{unitary} (p s); &\forall s. \text{finite} (\text{supp} (p s)); \\
\forall s. \text{sum} (p s) (\text{supp} (p s)) \leq 1 &\implies \text{well-def} (\text{SetPC} a p)
\end{aligned}
\]
\text{by (prover intro: well-defI healthy-wp-SetPC nearly-healthy-wlp-SetPC wp-under-wlp-SetPC sdp-SetPC sublinear-wp-SetPC cts-wp-SetPC dest:wd-dests unitary-sound sound-nneg [OF unitary-sound nnegD])}

lemma \textit{wd-Embed}:
fixes \(t::'s trans\)
assumes \(ht: \text{healthy} t\) and \(st: \text{sublinear} t\) and \(ct: \text{bd-cts} t\)
shows \(\text{well-def} (\text{Embed} t)\)
proof (intro well-defI)
from \(ht\) show \(\text{healthy} (\text{wp} (\text{Embed} t))\) nearly-healthy (\text{wp} (\text{Embed} t))
by (simp add: wp-def wp-def Embed-def healthy-nearly-healthy+)
from \(st\) show \(\text{sublinear} (\text{wp} (\text{Embed} t))\) by (simp add: wp-under-wlp-def wp-eval)
show \(\text{wp-under-wlp} (\text{Embed} t)\) by (simp add: wp-under-wlp-def wp-eval)
show \(\text{sub-distrib-pconj} (\text{Embed} t)\)
by (rule sub-distrib-pconjI, auto intro:le-funI [OF sublinearD [OF \(st\) where \(a=1\) and \(b=1\) and \(c=1\), simplified]])
simp: exp-conj-def pconj-def wp-def Embed-def)
from \(ct\) show \(\text{bd-cts} (\text{wp} (\text{Embed} t))\)
by (simp add: wp-def Embed-def)
qed

lemma \textit{wd-repeat}:
well-def \(a\) \implies well-def (repeat \(n\ a\))

lemma \textit{wd-Bind}:
\[
\begin{aligned}
\forall s. \text{well-def} (a (f s)) &\implies \text{well-def} (\text{Bind} f a)
\end{aligned}
\]
\text{by (blast intro: healthy-wp-Bind nearly-healthy-wlp-Bind wp-under-wlp-Bind sdp-Bind sublinear-wp-Bind cts-wp-Bind)}

lemma \textit{wd-loop}:
well-def body \implies well-def (do \(G \longrightarrow\) body od)
\text{by (blast intro: healthy-wp-loop nearly-healthy-wlp-loop wp-under-wlp-loop sdp-loop sublinear-wp-loop cts-wp-loop)}

lemmas \(\text{wd-intros} =\)
\(\text{wd-Abort} \ \text{wd-Skip} \ \text{wd-Apply} \)
\(\text{wd-Embed} \ \text{wd-Seq} \ \text{wd-PC} \)
\(\text{wd-DC} \ \text{wd-SetPC} \ \text{wd-SetDC} \)
\(\text{wd-Bind} \ \text{wd-repeat} \ \text{wd-loop} \)
4.8 The Loop Rules

theory Loops imports WellDefined begin

Given a well-defined body, we can annotate a loop using an invariant, just as in the classical setting.

4.8.1 Liberal and Strict Invariants.

A probabilistic invariant generalises a boolean one: it entails itself, given the loop guard.

definition wp-inv :: (′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:
    ∧ I. (∀ s. «G» s * I s ≤ wp body I s) =⇒ 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:
    ∧ I. (∀ s. «G» s * I s ≤ wlp body I s) =⇒ wlp-inv G body I
    by (simp add: wlp-inv-def)

lemma wlp-invD:
    wlp-inv G body I =⇒ «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 (rule le-funI)
    fix s
    show «G» & & «I» s ≤ wp body «I» s
    proof (cases G s)
        case False
        with hb show ?thesis
        by (auto simp: exp-conj-def)
next
  case True
  hence («G» && «I») s = «G» s * «I» s
    by (simp add: exp-conj-def)
  also from inv have «G» s * «I» s ≤ wp body «I» s
    by (simp add: wp-inv-def)
  finally show ?thesis .
qed

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 ≤ wlp do G → body od ?P)
proof -
  let ?f Q s = «G» s * wlp body Q s + «N G» s * ?P s
  have I ⊢ ⊢ gfp-exp ?f
    proof (rule gfp-exp-upperbound [OF - uI])
      have I = (λs. («G» s + «N G» s) * I s) by (simp add: negate-embed)
      also have ... = (λs. «G» s * I s + «N G» s * I s)
        by (simp add: algebra-simps)
      also have ... = (λs. «G» s * («G» s * I s) + «N G» s * («N G» s * I s))
        by (simp add: embed-bool-idem algebra-simps)
      also have ... ⊢ (λs. «G» s * wlp body I s + «N G» s * («N G» s * I s))
        using inv by (auto dest: wlp-invD intro: add-mono mult-left-mono)
      finally show I ⊢ (λs. «G» s * wlp body I s + «N G» s * («N G» s * I s)) .
  qed
  also from uI well-def-wlp-nearly-healthy [OF wd] have ...
    = wlp do G → body od ?P
    by (auto intro: wlp-Loop1 [symmetric] unitary-intros)
  finally show ?thesis .
qed

4.8.3 Total Correctness

The first total correctness lemma for loops which terminate with probability 1 [McIver and Morgan, 2004, Lemma 7.3.1, §7, p. 186].

lemma wp-Loop:
  assumes wd: well-def body
    and inv: wlp-inv G body I
    and unit: unitary I
4.8. THE LOOP RULES

shows I && wp (do G → body od) (λs. 1) ⊢ wp (do G → body od) (λs. «N
G» s * I s)
(is I && ?T ⊢ wp ?loop ?X)

proof –

We first appeal to the liberal loop rule:

from assms have I && ?T ⊢ wp ?loop ?X && ?T
by(blast intro:exp-conj-mono-left wp-Loop)

Next, by sub-conjunctivity:

also {
from wd have sdp-loop: sub-distrib-pconj (do G → 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 equiv-trans (lfp-trans ?X)
  (wp (body :: Embed (lfp-trans ?X) « G ⊕ Skip))

proof(intro lfp-trans-unfold)

fix t::'s trans and P::'s expect

assume st: Q. sound Q ⇒ sound (t Q)
and sP: sound P

with h show sound (?X t P)
by(rule wp-loop-step-sound)

next

fix u::'s trans

assume le-trans t u (P. sound P ⇒ sound (t P))
(P. sound P ⇒ sound (u P))

with h show le-trans (wp (body :: Embed t « G ⊕ Skip))
  (wp (body :: Embed u « G ⊕ Skip))

by(iprover intro:wp-loop-step-mono)

next
let \( ?v = \lambda P . \text{bound-of } P \)
from \( h \) show \( \text{le-trans} (\text{wp } (\text{body };; \text{Embed } ?v \circ G \oplus \text{Skip}) ) ?v \)
by\( (\text{intro le-transI, simp add: wp-loop-fp[unfolded negate-embed]})) \)
fix \( P :: s \) expect
assume sound \( P \) thus sound \( (?v \circ P) \) by\( (\text{auto}) \)
qed
also have \( \text{equiv-trans} ... \)
(\( \lambda P . \text{«N G» s } \circ P s + \text{«G» s } \circ \text{wp body } (\text{Embed } (\text{lfp-trans } ?X) ) P s) \)
by\( (\text{rule equiv-transI, simp add: wp-eval algebra-simps negate-embed}) \)
finally show \( \text{lfp-trans } ?X P = \)
(\( \lambda s . \text{«N G» s } \circ P s + \text{«G» s } \circ \text{wp body } (\text{lfp-trans } ?X P) s) \)
using \( sP \) unfolding \( \text{wp-eval} \) by\( (\text{blast}) \)
qed

lemma \( \text{wp-loop-nguard} : \)
\[
[ \text{healthy } (\text{wp body}); \text{sound } P; \neg G s ] \Rightarrow \text{wp } \text{do } G \rightarrow \text{body } \text{od } P s = P s
\]
by\( (\text{subst wp-loop-unfold, simp-all}) \)

lemma \( \text{wp-loop-guard} : \)
\[
[ \text{healthy } (\text{wp body}); \text{sound } P; G s ] \Rightarrow \\
\text{wp } \text{do } G \rightarrow \text{body } \text{od } P s = \text{wp } (\text{body };; \text{do } G \rightarrow \text{body } \text{od}) P s
\]
by\( (\text{subst wp-loop-unfold, simp-all add: wp-eval}) \)
end

4.9 The Algebra of pGCL

theory Algebra imports WellDefined begin

Programs in pGCL have a rich algebraic structure, largely mirroring that for GCL. We show that programs form a lattice under refinement, with \( a \sqcap b \) and \( a \sqcup b \) as the meet and join operators, respectively. We also take advantage of the algebraic structure to establish a framework for the modular decomposition of proofs.

4.9.1 Program Refinement

Refinement in pGCL relates to refinement in GCL exactly as probabilistic entailment relates to implication. It turns out to have a very similar algebra, the rules of which we establish shortly.

definition refines :: \( 's \text prog \Rightarrow 's \text prog \Rightarrow \text bool \) \( (\text{infix } \sqsubseteq \text{ 70}) \)
where
\( \text prog \sqsubseteq \text prog' \equiv \forall P. \text{sound } P \rightarrow \text{wp } \text prog P \vdash \text{wp } \text prog' P \)

lemma refinesI [intro]:
\[
[ \forall P. \text{sound } P \rightarrow \text{wp } \text prog P \vdash \text{wp } \text prog' P ] \Rightarrow \text prog \sqsubseteq \text prog'
\]
4.9. THE ALGEBRA OF PGCL

unfolding refines-def by (simp)

lemma refinesD[dest]:
\[ \text{prog} \sqsubseteq \text{prog'}; \text{sound } P \Rightarrow \text{wp } \text{prog} P \equiv \text{wp } \text{prog'} P \]
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 ≃)
where
\[ \text{prog} \equiv \text{prog'} \equiv \forall P. \text{sound } P \Rightarrow \text{wp } \text{prog} P = \text{wp } \text{prog'} P \]

lemma pequivI[intro]:
\[ \forall P. \text{sound } P \Rightarrow \text{wp } \text{prog} P = \text{wp } \text{prog'} P \]
unfolding pequiv-def by (simp)

lemma pequivD[dest, simp]:
\[ \text{prog} \equiv \text{prog'} ; \text{sound } P \Rightarrow \text{wp } \text{prog} P = \text{wp } \text{prog'} P \]
unfolding pequiv-def by (simp)

lemma pequiv-equiv-trans:
\[ a \equiv b \iff \text{equiv-trans } (\text{wp } a) (\text{wp } b) \]
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]:
\[ a \sqcap b = b \sqcap a \]
unfolding DC-def by (simp add: ac-simps)

lemma DC-assoc[ac-simps]:
\[ a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c \]
unfolding DC-def by (simp add: ac-simps)

lemma DC-idem:
\[ a \sqcap a = a \]
unfolding DC-def by (simp)

lemma AC-comm[ac-simps]:
\[ a \sqcup b = b \sqcup a \]
unfolding AC-def by (simp add: ac-simps)
lemma AC-associativity\(\text{[ac-simps]}\):
\[ a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c \]
unfolding AC-def by (simp add:ac-simps)

lemma AC-idem:
\[ a \sqcup a = a \]
unfolding AC-def by (simp)

lemma PC-quasi-comm:
\[ a \oplus b = b (\lambda s. \text{1 - } p s) \oplus a \]
unfolding PC-def by (simp add:algebra-simps)

lemma PC-idem:
\[ a \oplus a = a \]
unfolding PC-def by (simp add:algebra-simps)

lemma Seq-associativity\(\text{[ac-simps]}\):
\[ A ;; (B ;; C) = A ;; B ;; C \]
by (simp add:Seq-def o-def)

lemma Abort-refines[intro]:
well-def a \implies 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:
\[ (a \sqcap b) \sqsubseteq a \]
by (auto intro!:refinesI simp:wp-eval)

lemma right-refines-DC:
\[ (a \sqcap b) \sqsubseteq b \]
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 \sqcap c)
proof
fix P::'s \Rightarrow real assume sP: sound P
with assms have wp a P \vdash wp b P and wp a P \vdash wp c P
by (auto dest:refinesD)
thus wp a P \vdash wp (b \sqcap c) P
by (auto simp:wp-eval intro:min.boundedI)
qed

lemma DC-mono:
fixes a::'s prog
assumes \( r_a b \) and \( r_c d \)
shows \((a \sqcup c) \sqsubseteq (b \sqcup d)\)

proof\( (\text{rule refinesI, unfold wp-eval, rule le-funI}) \)

fix \( P;\)'s \( \Rightarrow \) real and \( s;\)'s
assume \( s;P \)

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 \( r_a c \) and \( r_b c \)
shows \( (a \sqcup b) \sqsubseteq c \)

proof

fix \( P;\)'s \( \Rightarrow \) real assume \( s;P \)
with assms have \( \bigwedge s. \ wp a P s \leq wp c P s \)
and \( \bigwedge s. \ wp b P s \leq wp c P s \)
by(auto dest:refinesD)
thus \( wp (a \sqcup b) P \Rightarrow wp c P \)
unfolding wp-eval by(auto)

qed

lemma AC-mono:
fixes \( a;\)'s prog
assumes \( r_a b \) and \( r_c d \)
shows \((a \sqcup c) \sqsubseteq (b \sqcup d)\)

proof\( (\text{rule refinesI, unfold wp-eval, rule le-funI}) \)

fix \( P;\)'s \( \Rightarrow \) real and \( s;\)'s
assume \( s;P \)
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 m b) ⊆ (a p b)
proof(rule refinesI, unfold wp-eval, rule le-funI)
  fix s and P::'a ⇒ real assume sound: sound P
  from unit have nn-p: 0 ≤ p s by(blast)
  from unit have p s ≤ 1 by(blast)
  hence nn-np: 0 ≤ 1 − p s by(simp)
  show min (wp a P s) (wp b P s) ≤ p s * wp a P s + (1 − p s) * wp b P s
    proof(cases wp a P s ≤ wp b P s,
      simp-all add: min.absorb1 min.absorb2)
      case True note le = this
      have wp a P s = (p s + (1 − p s)) * wp a P s by(simp)
      also have ... = p s * wp a P s + (1 − p s) * wp a P s
        by(simp only: distrib-right)
      also {
        from le and nn-np have (1 − p s) * wp a P s ≤ (1 − p s) * wp b P s
          by(rule mult-left-mono)
        hence p s * wp a P s + (1 − p s) * wp a P s ≤
          p s * wp a P s + (1 − p s) * wp b P s
          by(rule add-left-mono)
      }
    finally show wp a P s ≤ p s * wp a P s + (1 − p s) * wp b P s .
  next
  case False
  then have le: wp b P s ≤ wp a P s by(simp)
  have wp b P s = (p s + (1 − p s)) * wp b P s by(simp)
  also have ... = p s * wp b P s + (1 − p s) * wp b P s
    by(simp only: distrib-right)
  also {
    from le and nn-p have p s * wp b P s ≤ p s * wp a P s
      by(rule mult-left-mono)
    hence p s * wp b P s + (1 − p s) * wp b P s ≤
      p s * wp a P s + (1 − p s) * wp b P s
      by(rule add-right-mono)
  }
  finally show wp b P s ≤ p s * wp a P s + (1 − p s) * wp b P s .
qed
qed

Laws depending on the arithmetic of a p b and a m b together

lemma PC-refines-AC:
  assumes unit: unitary p
  shows (a p b) ⊆ (a m 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 \( p \cdot s \cdot (1 - p) \cdot s \cdot (wp \cdot b \cdot P \cdot s \cdot \leq \max (wp \cdot a \cdot P \cdot s \cdot (wp \cdot b \cdot P \cdot s)) \)

**proof**

(case wp \( a \cdot P \cdot s \cdot \leq \wp \cdot b \cdot P \cdot s \))

**case** True

**note** leab = this

**with** unit nn-np

**have** p \( s \cdot wp \cdot a \cdot P \cdot s \cdot + (1 - p) \cdot s \cdot wp \cdot b \cdot P \cdot s \cdot \leq \)

\( p \cdot s \cdot wp \cdot b \cdot P \cdot s \cdot + (1 - p) \cdot s \cdot wp \cdot a \cdot P \cdot s \)

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

**also have** ... = \( wp \cdot b \cdot P \cdot s \)

by (auto simp: field-simps)

**also from** leab

**have** ... = \( max (wp \cdot a \cdot P \cdot s \cdot (wp \cdot b \cdot P \cdot s)) \)

by (auto)

**finally show** ?thesis .

next

**case** False

**note** leba = this

**with** unit nn-np

**have** p \( s \cdot wp \cdot a \cdot P \cdot s \cdot + (1 - p) \cdot s \cdot wp \cdot b \cdot P \cdot s \cdot \leq \)

\( p \cdot s \cdot wp \cdot a \cdot P \cdot s \cdot + (1 - p) \cdot s \cdot wp \cdot a \cdot P \cdot s \)

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

**also have** ... = \( wp \cdot a \cdot P \cdot s \)

by (auto simp: field-simps)

**also from** leba

**have** ... = \( max (wp \cdot a \cdot P \cdot s \cdot (wp \cdot b \cdot P \cdot s)) \)

by (auto)

**finally show** ?thesis .

qed

Laws depending on the arithmetic of \( a \bigcup b \) and \( a \bigcap b \) together

**lemma** DC-refines-AC:

\( (a \bigcap b) \subseteq (a \bigcup b) \)

by (auto intro!: refinesI simp: wp-eval)

Laws Involving Refinement and Equivalence

**lemma** pr-trans[trans];

fixes A::'a prog

**assumes** prAB: \( A \subseteq B \)

and prBC: \( B \subseteq C \)

**shows** \( A \subseteq C \)

**proof**

fix P::'a ⇒ real

assume sP: sound P

with prAB have wp A P ⊢ wp B P by (blast)

also from sP and prBC have ... ⊢ wp C P by (blast)

finally show wp A P ⊢ ... .

qed
lemma pequiv-refl[intro!, simp]:
\[ a \simeq a \]
by (auto)

lemma pequiv-comm[ac-simps]:
\[ a \simeq b \iff b \simeq a \]
unfolding pequiv-def
by (rule iffI, safe, simp-all)

lemma pequiv-pr[dest]:
\[ a \simeq b \Rightarrow a \sqsubseteq b \]
by (auto)

lemma pequiv-trans[intro, trans]:
\[ [a \simeq b; b \simeq c] \Rightarrow a \simeq c \]
unfolding pequiv-def by (auto intro!, order-trans)

lemma pequiv-pr-trans[intro, trans]:
\[ [a \sqsubseteq b; b \simeq c] \Rightarrow a \sqsubseteq c \]
unfolding pequiv-def refines-def by (simp)

Refinement induces equivalence by antisymmetry:

lemma pequiv-antisym:
\[ [a \sqsubseteq b; b \sqsubseteq a] \Rightarrow a \simeq b \]
by (auto intro! antisym)

lemma pequiv-DC:
\[ [a \simeq c; b \simeq d] \Rightarrow (a \cap b) \simeq (c \cap d) \]
by (auto intro!: DC-mono pequiv-antisym simp: ac-simps)

lemma pequiv-AC:
\[ [a \simeq c; b \simeq d] \Rightarrow (a \cup b) \simeq (c \cup d) \]
by (auto intro!: AC-mono pequiv-antisym simp: ac-simps)

4.9.3 Deterministic Programs are Maximal

Any sub-additive refinement of a deterministic program is in fact an equivalence. Deterministic programs are thus maximal (under the refinement order) among sub-additive programs.

lemma refines-determ:
\begin{verbatim}
fixes a::'s prog
assumes da: determ (wp a)
    and wa: well-def a
    and wb: well-def b
\end{verbatim}
and \( dr : a \sqsubseteq b \)
shows \( a \simeq b \)

Proof by contradiction.

**proof** (rule pequivI, rule contrapos-pp)
from \( \text{wb have feasible (wp b) by(auto)} \)
with \( \text{wb have sub; sub-add (wp b)} \)
by (auto dest: sublinear-subadd[OF well-def-wp-sublinear])
fix \( P : \forall s \Rightarrow \text{real assume sP: sound P} \)

Assume that \( a \) and \( b \) are not equivalent:

assume \( nc: \text{wp a P }\neq \text{wp b P} \)

Find a point at which they differ. As \( a \sqsubseteq b \), \( \text{wp b P s} \) must be strictly greater than \( \text{wp a P s} \) here:

\[
\text{hence } \exists s. \text{wp a P s }< \text{wp b P s}
\]

**proof** (rule contrapos-np)
assume \( \neg(\exists s. \text{wp a P s }< \text{wp b P s}) \)

\[
\text{hence } \forall s. \text{wp b P s }\leq \text{wp a P s by(auto simp:not-less)}
\]

\[
\text{hence wp b P }\vdash \text{wp a P by(auto)}
\]
moreover from \( sp \text{ dr have wp a P }\vdash \text{wp b P by(auto)} \)
ultimately show \( \text{wp a P }\vdash \text{wp b P by(auto)} \)

qed
then obtain \( s \) where \( \text{less: wp a P s }< \text{wp b P s by(blast)} \)

Take a carefully constructed expectation:

let \( ?Pc = \lambda s. \text{bound-of P }\sim P s \)

have \( spc: \text{sound ?Pc} \)
**proof** (rule soundI)
from \( sp \text{ have } \forall s. 0 \leq P s \text{ by(auto)} \)

\[
\text{hence } \forall s. \text{?Pc s }\leq \text{bound-of P by(auto)}
\]
thus \( \text{bounded ?Pc by(blast)} \)
from \( sp \text{ have } \forall s. P s \leq \text{bound-of P by(auto)} \)

\[
\text{hence } \forall s. 0 \leq ?Pc s \text{ by(auto simp:sign-simps)}
\]
thus \( \text{nneg ?Pc by(auto)} \)

qed

We then show that \( \text{wp b} \) violates feasibility, and thus healthiness.

from \( sp \text{ have } 0 \leq \text{bound-of P by(auto)} \)
with \( da \text{ have } \text{bound-of P }= \text{wp a } (\lambda s. \text{bound-of P }) s \)
by (simp add: maximalD determ-maximalD)
also have \( ... = \text{wp a } (\lambda s. \text{?Pc s }+ P s) s \)
by (simp)
also from \( da spc \text{ have } ... = \text{wp a } ?Pc s + \text{wp a P s} \)
by (subst additiveD[OF determ-additiveD], simp-all add:ssP spc)
also from \( spc dr \text{ have } ... \leq \text{wp b } ?Pc s + \text{wp a P s} \)
by (auto)
also from \( less \text{ have } ... < \text{wp b } ?Pc s + \text{wp b P s} \)
by (auto)
also from \( sP \) \( sPc \) have ... \( \leq \) wp b (\( \lambda s. \ ?Pc s + P s \) s) by(blast)
finally have \( \neg \) wp b (\( \lambda s. \ bound-of P \) s) \( \leq \) bound-of P
by(simp)
thus \( \neg \) bounded-by (bound-of P) (wp b (\( \lambda s. \ bound-of P \))
by(auto)

next

However,

fix P::.s ⇒ real assume sP: sound P
hence nneg (\( \lambda s. \ bound-of P \)) by(auto)
moreover have bounded-by (bound-of P) (\( \lambda s. \ bound-of P \)) by(auto)
ultimately
show bounded-by (bound-of P) (wp b (\( \lambda s. \ bound-of P \))
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 \( \sqcap \) 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 : \( \lambda a b. \ a \simeq b \land well-def a \land well-def b \)
proof(rule part-equivpI)

have Skip \simeq Skip and well-def Skip by(auto intro:wd-intros)
thus \( \exists x. \ x \simeq x \land well-def x \land well-def x \) by(blast)
show symp (\( \lambda a b. \ a \simeq b \land well-def a \land well-def b \))
proof(rule sympI, safe)

fix a::'a prog and b
assume a \simeq b
hence equiv-trans (wp a) (wp b)
by(simp add:pequiv-equiv-trans)
thus b \simeq a by(simp add:ac-simps pequiv-equiv-trans)
qed
show transp (\( \lambda a b. \ a \simeq b \land well-def a \land well-def b \))
by(rule transpI, safe, rule pequiv-trans)
qed

instantiation program :: (type) semilattice-inf begin
lift-definition
less-eq-program :: 'a program ⇒ 'a program ⇒ bool is refines
proof(safe)
fix a::'a prog and b c d
assume a \simeq b hence b \simeq a by(simp add:ac-simps)
also assume a \sqsubseteq c
also assume \( c \simeq d \)
finally show \( b \sqsubseteq d \).

next
fix \( a::'a \ prog\) and \( b \ c \ d \)
assume \( a \simeq b \)
also assume \( b \sqsubseteq d \)
also assume \( c \simeq d \) hence \( d \simeq c \) by(simp add:ac-simps)
finally show \( a \sqsubseteq c \).

qed

lift-definition

less-program :: \( 'a \ prog \Rightarrow 'a \ prog \Rightarrow \ bool \)

is \( \lambda \ a \ b . \ a \sqsubseteq b \land \neg b \sqsubseteq a \)

proof(safe)
fix \( a::'a \ prog\) and \( b \ c \ d \)
assume \( a \simeq b \) hence \( b \simeq a \) by(simp add:ac-simps)
also assume \( a \sqsubseteq c \)
also assume \( c \simeq d \)
finally show \( b \sqsubseteq d \).

next
fix \( a::'a \ prog\) and \( b \ c \ d \)
assume \( a \simeq b \)
also assume \( b \sqsubseteq d \)
also assume \( c \simeq d \) hence \( d \simeq c \) by(simp add:ac-simps)
finally show \( a \sqsubseteq c \).

next
fix \( a \ b \) and \( c::'a \ prog\) and \( d \)
assume \( c \simeq d \)
also assume \( d \sqsubseteq b \)
also assume \( a \simeq b \) hence \( b \simeq a \) by(simp add:ac-simps)
finally have \( c \sqsubseteq a \).
moreover assume \( \neg c \sqsubseteq a \)
ultimately show \( False \) by(auto)

next
fix \( a \ b \) and \( c::'a \ prog\) and \( d \)
assume \( c \simeq d \) hence \( d \simeq c \) by(simp add:ac-simps)
also assume \( c \sqsubseteq a \)
also assume \( a \simeq b \)
finally have \( d \sqsubseteq b \).
moreover assume \( \neg d \sqsubseteq b \)
ultimately show \( False \) by(auto)

qed

lift-definition

inf-program :: \( 'a \ prog \Rightarrow 'a \ prog \Rightarrow 'a \ prog \) is DC

proof(safe)
fix \( a \ b \ c \ d::'s \ prog\)
assume \( a \simeq b \) and \( c \simeq d \)
thus \( (a \sqcap c) \simeq (b \sqcap d) \) by(rule pequiv-DC)
next
  fix a c::'s prog
  assume well-def a well-def c
  thus well-def (a ∩ c) by (rule wd-intros)
next
  fix a c::'s prog
  assume well-def a well-def c
  thus well-def (a ∩ c) by (rule wd-intros)
qed

instance
proof
  fix x y::'a program
  show (x < y) = (x ≤ y ∧ ¬ y ≤ x)
    by (transfer, simp)
  show x ≤ x
    by (transfer, auto)
  show inf x y ≤ x
    by (transfer, rule left-refines-DC)
  show inf x y ≤ y
    by (transfer, rule right-refines-DC)
  assume x ≤ y and y ≤ x thus x = y
    by (transfer, iprove intro:pequiv-antisym)
next
  fix x y z::'a program
  assume x ≤ y and y ≤ z
  thus x ≤ z
    by (transfer, iprove intro:pr-trans)
next
  fix x y z::'a program
  assume x ≤ y and x ≤ z
  thus x ≤ inf y z
    by (transfer, iprove intro:DC-refines)
qed
end

instantiation program :: (type) bot begin
lift-definition
  bot-program :: 'a program is Abort
  by (auto intro:wd-intros)
instance ..
end

lemma eq-det: \a b::'s prog. \[ a ≃ b; determ (wp a) \] \implies determ (wp b)
proof (intro determl additive1 maximal1)
  fix a b::'s prog and P::'s ⇒ real
    and Q::'s ⇒ real and s::'s
  assume da: determ (wp a)
assume \( sP: \text{sound P} \) and \( sQ: \text{sound Q} \)
and \( \text{eq. } a \simeq b \)

hence \( \text{wp b } (\lambda s. \text{P s } + \text{Q s}) \) \( s = \text{wp a } (\lambda s. \text{P s } + \text{Q s}) \)
by\((\text{simp add:sound-intros})\)
also from \( da \ sP \ sQ \)

have \( = \text{wp a } P s + \text{wp a } Q s \)
by\((\text{simp add: additiveD determ-additiveD})\)
also from \( eq \ sP \ sQ \)

have \( = \text{wp b } P s + \text{wp b } Q s \)
by\((\text{simp add: pequivD determ-maximalD maximalD})\)
finally show \( \text{wp b } (\lambda s. \text{P s } + \text{Q s}) \) \( s = \text{wp b } P s + \text{wp b } Q s \).

next
fix \( a \ b::'s \text{ prog} \) and \( c::\text{real} \)
assume \( da::\text{ determ} (\text{wp a}) \)
assume \( a \simeq b \) hence \( b \simeq a \) by\((\text{simp add: ac-simps})\)
moreover assume \( nn: 0 \leq c \)
ultimately have \( \text{wp b } (\lambda-. c) = \text{wp a } (\lambda-. c) \)
by\((\text{simp add: pequivD const-sound})\)
also from \( da \ nn \) have \( = (\lambda-. c) \)
by\((\text{simp add: determ-maximalD maximalD})\)
finally show \( \text{wp b } (\lambda-. c) = (\lambda-. c) \).

qed

lift-definition
\[ \text{pdeterm} :: 's \text{ program } \Rightarrow \text{ bool} \]
is \( \lambda a. \text{ determ } (\text{wp a}) \)
proof\(\text{(safe)}\)
fix \( a \ b::'s \text{ prog} \)
assume \( a \simeq b \) and \( \text{determ } (\text{wp a}) \)
thus \( \text{determ } (\text{wp b}) \) by\((\text{rule eq-det})\)
next
fix \( a \ b::'s \text{ prog} \)
assume \( a \simeq b \) hence \( b \simeq a \) by\((\text{simp add: ac-simps})\)
moreover assume \( \text{determ } (\text{wp b}) \)
ultimately show \( \text{determ } (\text{wp a}) \) by\((\text{rule eq-det})\)
qed

lemma determ-maximal:
\[ \begin{array}{c}
\text{[ pdeterm } a; a \leq x] \implies a = x \\
\text{by(transfer, auto intro:refines-determ)}
\end{array} \]

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
\[ \text{drefines } :: (\text{'}b \Rightarrow 'a) \Rightarrow (\text{'}b \Rightarrow \text{ bool}) \Rightarrow 'a \text{ prog } \Rightarrow 'b \text{ prog } \Rightarrow \text{ bool} \]
chapter 4. the pgcl language

where
\( d\text{refines } \varphi \ G \ A \ B \equiv \forall P \ Q. \ (\text{unitary } P \ \land \ \text{unitary } Q \ \land \ (P \vdash wp \ A \ Q)) \rightarrow (\langle G \rangle \ \& \ \& \ (P \ o \ \varphi) \vdash wp \ B \ (Q \ o \ \varphi)) \)

lemma \( d\text{refinesD}[\text{dest}] \):
\[
\{d\text{refines } \varphi \ G \ A \ B; \ \text{unitary } P; \ \text{unitary } Q; \ P \vdash wp \ A \ Q \} \implies \\
\langle G \rangle \ \& \ \& \ (P \ o \ \varphi) \vdash wp \ B \ (Q \ o \ \varphi)
\]
unfolding \( d\text{refines-def} \) by\( (\text{blast}) \)

We can alternatively use \( G \) as an assumption:

lemma \( d\text{refinesD2} \):
assumes \( dr \): \( d\text{refines } \varphi \ G \ A \ B \)
and \( uP \): \( \text{unitary } P \)
and \( uQ \): \( \text{unitary } Q \)
and \( wpA \): \( P \vdash wp \ A \ Q \)
and \( G \): \( G \ s \)
shows \( (P \ o \ \varphi) \ s \leq wp \ B \ (Q \ o \ \varphi) \ s \)
proof –
from \( uP \) have \( 0 \leq (P \ o \ \varphi) \ s \)
unfolding \( o\text{-def} \) by\( (\text{blast}) \)
with \( G \) have \( (P \ o \ \varphi) \ s = (\langle G \rangle \ \& \ \& \ (P \ o \ \varphi)) \ s \)
by\( (\text{simp add:exp-conj-def}) \)
also from \( \text{assms} \) have \( ... \leq wp \ B \ (Q \ o \ \varphi) \ s \)
by\( (\text{blast}) \)
finally show \( (P \ o \ \varphi) \ s \leq ... \).
qed

This additional form is sometimes useful:

lemma \( d\text{refinesD3} \):
assumes \( dr \): \( d\text{refines } \varphi \ G \ a \ b \)
and \( G \): \( G \ s \)
and \( uQ \): \( \text{unitary } Q \)
and \( wa \): \( \text{well-def } a \)
shows \( \text{wp } a \ Q \ (\varphi \ s) \leq \text{wp } b \ (Q \ o \ \varphi) \ s \)
proof –
let \( ?L \ s' = \text{wp } a \ Q \ s' \)
from \( uQ \ wa \) have \( sL \): \( \text{sound } ?L \)
by\( (\text{blast}) \)
from \( uQ \ wa \) have \( bL \): \( \text{bounded-by } 1 \ ?L \)
by\( (\text{blast}) \)

have \( ?L \vdash ?L \)
by\( (\text{simp}) \)
with \( sL \) and \( bL \) and \( \text{assms} \)
show \( \text{thesis} \)
by\( (\text{blast intro: drefinesD2}[OF } dr, \text{ where } P=\ ?L, \text{ simplified}] \)
qed

lemma \( d\text{refinesI}[\text{intro}] \):
\[
\{ \ \land \ P \ Q, \ \land \ \text{unitary } P; \ \text{unitary } Q; \ P \vdash wp \ A \ Q \} \implies \\
\langle G \rangle \ \& \ \& \ (P \ o \ \varphi) \vdash wp \ B \ (Q \ o \ \varphi)
\]
unfolding \( d\text{refines-def} \) by\( (\text{blast}) \)

Use \( G \) as an assumption, when showing refinement:
4.9. THE ALGEBRA OF PGCL

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 $withAs:\ \begin{array}{l}
\forall P\ Q\ s.\ [\ \text{unitary}\ P;\ \text{unitary}\ Q;\\
G\ s;\ P\vdash wp\ A\ Q\ ]\implies (P\circ\varphi)\ s\leq wp\ B\ (Q\circ\varphi)\ s
\end{array}$
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 $\begin{array}{l}
\forall s.\ G\ s\implies (P\circ\varphi)\ s\leq wp\ B\ (Q\circ\varphi)\ s
\end{array}$
using $\begin{array}{l}
withAs\ by(blast)
\end{array}$
moreover
from $uQ$ have $\text{unitary}\ (Q\circ\varphi)$
unfolding o-def by(blast)
moreover
from $uP$ have $\text{unitary}\ (P\circ\varphi)$
unfolding o-def by(blast)
ultimately
show $\begin{array}{l}
\langle G\rangle\ &\& (P\circ\varphi)\vdash wp\ B\ (Q\circ\varphi)
\end{array}$
using $\begin{array}{l}
wB\ by(blast\ intro:entails-pconj-assumption)
\end{array}$
qed

lemma dr-strengthen-guard:
fixes $a::'s\ prog$ and $b::'t\ prog$
assumes $fg: \begin{array}{l}
\forall s.\ F\ s\implies G\ s
\end{array}$
and $drab: drefines\ \varphi\ G\ a\ b$
shows $drefines\ \varphi\ F\ a\ b$
proof(intro drefinesI)
fix $P\ Q::'s\ expect$
assume $uP: unitary\ P$ and $uQ: unitary\ Q$
and $wp: P\vdash wp\ a\ Q$
from $fg$ have $\begin{array}{l}
\forall s.\ \langle F\rangle\ s\leq \langle G\rangle\ s\ by(simp\ add:embed-bool-def)
\end{array}$
hence $\begin{array}{l}
\langle F\rangle\ &\& (P\circ\varphi)\vdash (\langle G\rangle\ &\& (P\circ\varphi))\ by(auto\ intro:pconj-mono\ le-funI\ simp:exp-conj-def)
\end{array}$
also from $drab$ $uP\ uQ\ wp$ have $\begin{array}{l}
\vdash wp\ b\ (Q\circ\varphi)\ by(auto)
\end{array}$
finally show $\begin{array}{l}
\langle F\rangle\ &\& (P\circ\varphi)\vdash wp\ b\ (Q\circ\varphi)
\end{array}$.
qed

Probabilistic correspondence, pcorres, is equality on distribution transformers, modulo a guard. It is the analogue, for data refinement, of program equivalence for program refinement.

definition
 CHAPTER 4. THE PGCL LANGUAGE

let pcorres : ('b ⇒ 'a) ⇒ ('b ⇒ bool) ⇒ 'a prog ⇒ 'b prog ⇒ bool

where
pcorres ϕ G A B ←→
(∀ Q.  unitary Q   «G» && (wp A Q o ϕ) = «G» && wp B (Q o ϕ))

lemma pcorresI:
[ \[ \forall Q.  unitary Q   «G» && (wp A Q o ϕ) = «G» && wp B (Q o ϕ) \] ]   ⇒
pcorres ϕ G A B
by(simp add:pcorres-def)

Often easier to use, as it allows one to assume the precondition.

lemma pcorresI2[intro]:
fixes A :: 'a prog and B :: 'b prog
assumes withG: \[ \forall Q s.  [ [ unitary Q; G s ] ] = wp A Q o ϕ\] s = wp B (Q o ϕ) s
and wA: well-def A
and wB: well-def B
shows pcorres ϕ G A B
proof(rule pcorresI, rule ext)
fix Q: 'a ⇒ real and s: 'b
assume uQ: unitary Q
hence uQϕ: unitary (Q o ϕ) by(auto)
show («G» && (wp A Q o ϕ)) s = («G» && wp B (Q o ϕ)) s
proof(cases G s)
  case True note this
  moreover
  from well-def-wp-healthy[OF wA uQϕ have 0 ≤ wp A Q (ϕ s) by(blast)
  moreover
  from well-def-wp-healthy[OF wB uQϕ have 0 ≤ wp B (Q o ϕ) s by(blast)
  ultimately show ?thesis
    using uQ by(simp add:exp-conj-def withG)
next
  case False note this
  moreover
  from well-def-wp-healthy[OF wA uQϕ have wp A Q (ϕ s) ≤ 1 by(blast)
  moreover
  from well-def-wp-healthy[OF wB uQϕ have wp B (Q o ϕ) s ≤ 1
    by(blast dest!:healthy-bounded-byD intro:sound-nneg)
  ultimately show ?thesis by(simp add:exp-conj-def)
qed

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 \( \wp A Q (\varphi s) = \wp B (Q o \varphi) s \)

**proof**

- from \( uQ \) well-def-wp-healthy\([OF wA]\) have \( \theta \leq \wp A Q (\varphi s) \) by(auto)
- with \( G \) have \( \wp A Q (\varphi s) = \langle \varphi \rangle s \& \wp A Q (\varphi s) \) by(simp)

also 

- from \( pc uQ \) have \( \langle G \rangle s \& (wp A Q o \varphi) \) = \( \langle G \rangle \) &\& \( wp B (Q o \varphi) s \) by(rule pcorresD)
- hence \( \langle G \rangle s \& wp A Q (\varphi s) = \langle G \rangle s \& wp B (Q o \varphi) s \)

unfolding exp-conj-def o-def by(rule fun-cong)

also 

- from \( uQ \) have \( sound \) \( Q \) by(auto)
- hence \( sound (Q o \varphi) \) by(auto intro: sound-intros)
  
  with well-def-wp-healthy\([OF wB]\) have \( \theta \leq wp B (Q o \varphi) s \) by(auto)

- with \( G \) have \( \langle G \rangle s \& wp B (Q o \varphi) s = \wp B (Q o \varphi) s \) 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 \( rab: a \subseteq b \) and \( wb: \) well-def \( b \)
shows \( drefines (\lambda s. s) \) \( G a b \)

**proof** (intro drefinesI2 \( wb \), simp add:o-def)

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

assume \( sQ: unitary Q \)
assume \( P \vdash wp A Q \) hence \( P s \leq wp A Q s \) by(auto)
also from \( rab sQ \) have \( ... \leq wp b Q s \) by(auto)
finally show \( P s \leq wp b Q s \) .

qed

Data refinement is transitive:

**lemma** dr-trans[trans]:

fixes \( A::'a\ prog \) and \( B::'b\ prog \) and \( C::'c\ prog \)
assumes \( drAB: drefines \varphi G A B \)
and \( drBC: drefines \varphi' G' B C \)
and \( Gimp: \forall s. G' s \implies G (\varphi' s) \)
shows \( drefines (\varphi o \varphi') G' A C \)

**proof** (rule drefinesI)

fix \( P::'a \Rightarrow \) real and \( Q::'a \Rightarrow \) real and \( s::'a \)
assume \( uP: unitary P \) and \( uQ: unitary Q \)
and \( wpA: P \vdash wp A Q \)
have $\langle G' \rangle \land \langle G \circ \varphi' \rangle = \langle G' \rangle$

proof (rule ext, unfold exp-conj-def)

fix $x$

show $\langle G' \rangle x \land \langle G \circ \varphi' \rangle x = \langle G' \rangle x$ (is ?X)

proof (cases $G' x$)

  case False then show ?X by (simp)

next

  case True

  moreover with $G' x$

  have $(G \circ \varphi') x$ by (simp add: o-def)

  ultimately

  show ?X by (simp)

qed

qed

with $uP$

have $\langle G' \rangle \land (P \circ (\varphi \circ \varphi')) = \langle G' \rangle \land ((\langle G \rangle \land (P \circ \varphi)) \circ \varphi')$

by (simp add: exp-conj-assoc o-assoc)

also {

  from $uP \ uQ \ \text{wpA}$ and $\text{drAB}$

  have $\langle G \rangle \land (P \circ \varphi) \vdash \text{wp B} (Q \circ \varphi)$

  by (blast intro: drefinesD)

  with $\text{drBC}$ and $uP \ uQ$

  have $\langle G' \rangle \land ((\langle G \rangle \land (P \circ \varphi)) \circ \varphi') \vdash \text{wp C} ((Q \circ \varphi) \circ \varphi')$

  by (blast intro: unitary-intros drefinesD)
}

finally

show $\langle G' \rangle \land (P \circ (\varphi \circ \varphi')) \vdash \text{wp C} ((Q \circ (\varphi \circ \varphi'))$?

by (simp add: o-assoc)

qed

Data refinement composes with program refinement:

lemma $\text{pr-dr-trans}[\text{trans}]$:

assumes $\text{prAB}: A \sqsubseteq B$

and $\text{drBC}: \text{drefines} \varphi \ G \ B \ C$

shows $\text{drefines} \varphi \ G \ A \ C$

proof (rule drefinesI)

fix $P$ and $Q$

assume $uP$: unitary $P$

and $uQ$: unitary $Q$

and $\text{wpA}: P \vdash \text{wp A} \ Q$

note $\text{wpA}$

also from $uQ$ and $\text{prAB}$ have $\text{wp A} \ Q \vdash \text{wp B} \ Q$ by (blast)

finally have $P \vdash \text{wp B} \ Q$

with $uP \ uQ \ \text{drBC}$
show «G» && (P o ϕ) ⊢ wp C (Q o ϕ) by(blast intro:drefinesD)
qed

lemma dr-pr-trans[trans]:
  assumes drAB: drefines ϕ G A B
  assumes prBC: B ⊑ C
  shows drefines ϕ G A C
proof(rule drefinesI)
  fix P and Q
  assume uP: unitary P
  and uQ: unitary Q
  and wpA: P ⊢ wp A Q
  with drAB have «G» && (P o ϕ) ⊢ wp B (Q o ϕ) by(blast intro:drefinesD)
  also from uQ prBC have ... ⊢ wp C (Q o ϕ) by(blast)
  finally show «G» && (P o ϕ) ⊢ ...
qed

If the projection ϕ commutes with the transformer, then data refinement is reflexive:

lemma dr-refl:
  assumes wa: well-def a
  and comm: ∀Q. unitary Q =⇒ wp a Q o ϕ ⊢ wp a (Q o ϕ)
  shows drefines ϕ G a a
proof(intro drefinesI2 wa)
  fix P and Q and s
  assume wp: P ⊢ wp a Q
  assume uQ: unitary Q

  have (P o ϕ) s = P (ϕ s) by(simp)
  also from wp have ... ⊆ wp a Q (ϕ s) by(blast)
  also {
    from comm uQ have wp a Q o ϕ ⊢ wp a (Q o ϕ) by(blast)
    hence (wp a Q o ϕ) s ⊆ wp a (Q o ϕ) s by(blast)
    hence wp a Q (ϕ s) ⊆ ... by(simp)
  }
  finally show (P o ϕ) s ⊆ wp a (Q o ϕ) s .
qed

Correspondence implies data refinement

lemma pcorres-drefine:
  assumes corres: pcorres ϕ G A C
  and wc: well-def C
  shows drefines ϕ G A C
proof
  fix P and Q
  assume uP: unitary P and uQ: unitary Q
  and wpA: P ⊢ wp A Q
  from wpA have P o ϕ ⊢ wp A Q o ϕ by(simp add:o-def le-fun-def)
hence \( «G» \land \land (P \circ \varphi) \vdash «G» \land \land (wp A Q \circ \varphi) \)
by (rule exp-conj-mono-right)
also from \( \text{corres } uQ \)
have \( \ldots = «G» \land \land (wp C (Q \circ \varphi)) \) 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 \( nn-wpC: 0 \leq wp C (Q \circ \varphi) s \) by (blast)
show \( («G» \land \land wp C (Q \circ \varphi)) s \leq wp C (Q \circ \varphi) s \)
proof (cases \( G \) \( s \))
case True
with \( nn-wpC \) 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 \( nn-wpC \)
ultimately show \( ?\text{thesis} \) by (simp add: exp-conj-def)
qed
qed

finally show \( «G» \land \land (P \circ \varphi) \vdash wp C (Q \circ \varphi) \).

qed

Any \textit{data} refinement of a deterministic program is correspondence. This is the analogous result to that relating program refinement and equivalence.

\textbf{lemma} \( \text{drefines-determ}: \)
fixes \( a::'a \text{ prog} \) and \( b::'b \text{ prog} \)
assumes \( da: \text{determ} (wp a) \)
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.

\textbf{proof} (rule pcorresI, rule contrapos-pp)
from \( wb \) show \( \text{feasible} (wp b) \) by (auto)

note \( ha = \text{well-def-wp-healthy}[OF wa] \)
note \( hb = \text{well-def-wp-healthy}[OF wb] \)
from \( wb \) have \( \text{sublinear} (wp b) \) by (auto)
moreover from \( hb \) have \( \text{feasible} (wp b) \) by (auto)
ultimately have \( \text{sab: sub-add (wp b)} \) by (rule sublinear-subadd)
4.9. THE ALGEBRA OF PGCL

fix \( Q : \alpha \Rightarrow \text{real} \)
assume \( uQ : \text{unitary } Q \)
hence \( uQ\varphi : \text{unitary } (Q \circ \varphi) \) by(auto)
assume \( \text{ne: } \langle G \rangle \& \& \langle \text{wp } a \ Q \circ \varphi \rangle \neq \langle G \rangle \& \& \langle \text{wp } b \ (Q \circ \varphi) \rangle \)
hence \( \text{ne': wp } a \ Q \circ \varphi \neq \text{wp } b \ (Q \circ \varphi) \) by(auto)

From refinement, \( \langle G \rangle \& \& \langle \text{wp } a \ Q \circ \varphi \rangle \) lies below \( \langle G \rangle \& \& \langle \text{wp } b \ (Q \circ \varphi) \rangle \).
from \( \text{ha } uQ \)
have \( \text{gle: } \langle G \rangle \& \& \langle \text{wp } a \ Q \circ \varphi \rangle \vdash \text{wp } b \ (Q \circ \varphi) \) by(blast intro:drefinesD[OF dr])
have \( \text{le: } \langle G \rangle \& \& \langle \text{wp } a \ Q \circ \varphi \rangle \leq \langle G \rangle \& \& \langle \text{wp } b \ (Q \circ \varphi) \rangle \) by(auto)
ultimately
show \( \langle G \rangle \& \& \langle \text{wp } a \ Q \circ \varphi \rangle \leq \langle G \rangle \& \& \langle \text{wp } b \ (Q \circ \varphi) \rangle \) 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 \( \text{rule: } \exists s. \langle G \rangle \& \& \langle \text{wp } a \ Q \circ \varphi \rangle \ s < \langle G \rangle \& \& \langle \text{wp } a \ Q \circ \varphi \rangle \ s \) by(auto)
proof(rule contrapos-np[OF ne], rule ext, rule antisym)
fix \( s \)
from \( \text{le show } \langle G \rangle \& \& \langle \text{wp } a \ Q \circ \varphi \rangle \ s \leq \langle G \rangle \& \& \langle \text{wp } a \ Q \circ \varphi \rangle \ s \) by(blast)
next
fix \( s \)
assume \( \neg (\exists s. \langle G \rangle \& \& \langle \text{wp } a \ Q \circ \varphi \rangle \ s < \langle G \rangle \& \& \langle \text{wp } a \ Q \circ \varphi \rangle \ s) \)
thus \( \langle G \rangle \& \& \langle \text{wp } b \ (Q \circ \varphi) \rangle \ s \leq \langle G \rangle \& \& \langle \text{wp } a \ Q \circ \varphi \rangle \ s \) by(simp add:not-less)
qed
from \( \text{this obtain } s \) where \( \text{less-s: } \langle G \rangle \& \& \langle \text{wp } a \ Q \circ \varphi \rangle \ s < \langle G \rangle \& \& \langle \text{wp } b \ (Q \circ \varphi) \rangle \ s \) by(blast)
The transformers themselves must differ at this point:

hence \( \text{larger: wp } a \ Q \ (\varphi \ s) < \text{wp } b \ (Q \circ \varphi) \ s \) by(auto)
proof(cases \( G \ s \))
case True
moreover from \( h a uQ \) have \( 0 \leq \text{wp } a \ Q \ (\varphi \ s) \) 
by(blast)
moreover from \( h b uQ \varphi \) have \( 0 \leq \text{wp } b \ (Q \circ \varphi) \ s \) 
by(blast)
moreover note \( \text{less-s} \)
ultimately show \( \text{?thesis by(simp add:exp-conj-def)} \)
next
case False
moreover from \( h a uQ \) have \( \text{wp } a \ Q \ (\varphi \ s) \leq 1 \) 
by(blast)
moreover {
  from \( uQ \) have \( \text{bounded-by } 1 \ (Q \circ \varphi) \) 
  by(blast)
  moreover from \( \text{unitary-sound}[OF \ uQ] \) 
  have \( \text{sound } (Q \circ \varphi) \) by(auto)
  ultimately have \( \text{wp } b \ (Q \circ \varphi) \ s \leq 1 \)
  using \( hb \) by(auto)
}
moreover note \( \text{less-s} \)
ultimately show \( \text{?thesis by(simp add:exp-conj-def)} \)
qed
from \( \text{less-s} \) have \( (\langle G \rangle \ \&\& \ (\text{wp } a \ Q \circ \varphi)) \ s \neq (\langle G \rangle \ \&\& \ \text{wp } b \ (Q \circ \varphi)) \ s \)
by(force)

\( G \) must also hold, as otherwise both would be zero.

hence \( G-s: \ G \ s \)
proof(rule contrapos-np)
  assume \( \neg G \ s \)
moreover from \( h a uQ \) have \( \text{wp } a \ Q \ (\varphi \ s) \leq 1 \) 
by(blast)
moreover {
  from \( uQ \) have \( \text{bounded-by } 1 \ (Q \circ \varphi) \) 
  by(blast)
  moreover from \( \text{unitary-sound}[OF \ uQ] \) 
  have \( \text{sound } (Q \circ \varphi) \) by(auto)
  ultimately have \( \text{wp } b \ (Q \circ \varphi) \ s \leq 1 \)
  using \( hb \) by(auto)
}
ultimately
show \( (\langle G \rangle \ \&\& \ (\text{wp } a \ Q \circ \varphi)) \ s = (\langle G \rangle \ \&\& \ \text{wp } b \ (Q \circ \varphi)) \ s \)
by(simp add:exp-conj-def)
qed

Take a carefully constructed expectation:

let \( Qc = \lambda s. \text{bound-of } Q - Q \ s \)
have \( bQc: \text{bounded-by } 1 \ ?Qc \)
proof(rule bounded-byI)
  fix \( s \)
from \( uQ \) have \( \text{bound-of } Q \leq 1 \) and \( 0 \leq Q \ s \) by(auto)
4.9. THE ALGEBRA OF PGCL

thus \( \text{bound-of } Q - Q s \leq 1 \) by(auto)

qed

have \( sQc : \text{sound } ?Qc \)

proof (rule soundI)

from \( bQc \) show bounded \( ?Qc \) by(auto)

show \( \text{neg } ?Qc \)

proof (rule negI)

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 \( \text{wp } a, \text{wp } b \) 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 \( \text{bound-of } Q = \text{wp } a (\lambda s. \text{bound-of } Q) (\varphi s) \)

by (simp add: maximalD determ-maximalD)

also have \( \text{wp } a (\lambda s. \text{bound-of } Q) (\varphi s) = \text{wp } a (\lambda s. Q \ s + \ ?Qc \ s) (\varphi s) \)

by (simp)

also \{

from \( da \) have additive \( (\text{wp } a) \) by(blast)

with \( uQ \ sQc \)

have \( \text{wp } a (\lambda s. Q \ s + \ ?Qc \ s) (\varphi s) = \text{wp } a Q (\varphi s) + \text{wp } a ?Qc (\varphi s) \) by (subst additiveD, blast+)

\}

also \{

from \( ha \) and \( sQc \) and \( bQc \)

have \( \langle \text{G} \rangle & (\text{wp } a \ ?Qc \circ \varphi) \vdash \text{wp } b (\ ?Qc \circ \varphi) \)

by (blast intro: drefinesD[OF dr])

hence \( \langle \text{G} \rangle & (\text{wp } a \ ?Qc \circ \varphi) \ s \leq \text{wp } b (\ ?Qc \circ \varphi) \ s \)

by (blast)

moreover from \( sQc \) and \( ha \)

have \( 0 \leq \text{wp } a (\lambda s. \text{bound-of } Q - Q s) (\varphi s) \)

by (blast)

ultimately

have \( \text{wp } a \ ?Qc (\varphi s) \leq \text{wp } b (\ ?Qc \circ \varphi) \ s \)

using \( \text{G-s} \) by (simp add: exp-conj-def)

hence \( \text{wp } a Q (\varphi s) + \text{wp } a \ ?Qc (\varphi s) \leq \text{wp } a Q (\varphi s) + \text{wp } b (\ ?Qc o \varphi) \ s \)

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 \)

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 \( \text{unitary-sound} \) \( \text{OF} \) \( \text{uQ} \) and \( \text{sQc} \)

have \( \text{wp b} (Q \circ \varphi) s + \text{wp b} (\text{?Qc} \circ \varphi) s \leq \text{wp b} (\lambda s. (Q \circ \varphi) s + (\text{?Qc} \circ \varphi) s) s \)

by\((\text{blast})\)

also have \( ... = \text{wp b} (\lambda s. \text{bound-of} \ Q) s \)

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

finally

show \( \neg \text{feasible} \ (\text{wp b}) \)

proof\((\text{rule contrapos-pn})\)

assume \( \mathbf{fb} : \text{feasible} \ (\text{wp b}) \)

have bounded-by \((\text{bound-of} \ Q) \ (\lambda s. \text{bound-of} \ Q)) \ (\text{wp b} (\lambda s. \text{bound-of} \ Q)) \)

using \( \text{uQ} \) \((\text{blast intro:feasible-boundedD[OF \ \mathbf{fb}]})\)

hence \( \text{wp b} (\lambda s. \text{bound-of} \ Q) s \leq \text{bound-of} \ Q \) \((\text{blast})\)

thus \( \neg \text{bound-of} \ Q < \text{wp b} (\lambda s. \text{bound-of} \ Q) s \) \((\text{simp})\)

qed

qed

4.9.7 Structural Rules for Correspondence

lemma \( \text{pcorres-Skip}: \)

\( \text{pcorres} \ \varphi \ G \ \text{Skip} \ \text{Skip} \)

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

Correspondence composes over sequential composition.

lemma \( \text{pcorres-Seq}: \)

fixes \( A :: \text{'}b \ \text{prog} \) \( \text{and} \ B :: \text{'}c \ \text{prog} \)

and \( C :: \text{'}b \ \text{prog} \) \( \text{and} \ D :: \text{'}c \ \text{prog} \)

and \( \varphi :: \text{'}c \Rightarrow \text{'}b \)

assumes \( \text{pcAB} : \text{pcorres} \ \varphi \ G \ A \ B \)

and \( \text{pcCD} : \text{pcorres} \ \varphi \ H \ C \ D \)

and \( wA : \text{well-def} \ A \) \( \text{and} \ wB : \text{well-def} \ B \)

and \( wC : \text{well-def} \ C \) \( \text{and} \ wD : \text{well-def} \ D \)

and \( \text{p3p2} : \forall Q. \ \text{unitary} \ Q \Rightarrow \ «I» \ \& \& \ \text{wp} \ B \ Q = \ «H» \ \& \& \ Q \)

and \( \text{p1p3} : \forall s. \ G \ s \Rightarrow I \ s \)

shows \( \text{pcorres} \ \varphi \ G \ (A;C) \ (B;D) \)

proof\((\text{rule pcorresI})\)

fix \( Q :: \text{'}b \Rightarrow \text{real} \)

fix \( R :: \text{'}c \Rightarrow \text{real} \)

assume \( uQ : \text{unitary} \ Q \)

with \( \text{well-def-up-healthy}[\text{OF} \ wC] \) have \( uCQ : \text{unitary} \ (\text{wp} \ C \ Q) \) \((\text{by(auto)})\)

from \( uQ \) \( \text{well-def-wp-healthy}[\text{OF} \ wD] \) have \( uDQ : \text{unitary} \ (\text{wp} \ D \ (Q \circ \varphi)) \)

by\((\text{auto dest:unitary-comp})\)

have \( \text{p3p1} : \forall R S. \ [ \text{unitary} \ R ; \ \text{unitary} \ S ; \ «I» \ \& \& \ R = «I» \ \& \& \ S ] \Rightarrow \ «G» \ \& \& \ R = «G» \ \& \& \ S \)

proof\((\text{rule ext})\)

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

assume \( aS : «I» \ \& \& \ R = «I» \ \& \& \ S \)

and \( uR : \text{unitary} \ R \) \( \text{and} \ uS : \text{unitary} \ S \)
4.9. THE ALGEBRA OF PGCL

show \((\langle G \rangle \land R)\ s = (\langle G \rangle \land \langle S \rangle)\ s\)

proof(simp add:exp-conj-def, cases G s)

case False note this

moreover from uR have \(R \ s \leq 1\) by(blast)

moreover from uS have \(S \ s \leq 1\) by(blast)

ultimately show \(\langle G \rangle \ s \land R \ s = \langle G \rangle \ s \land S \ s\)

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)

qed

show \(\langle G \rangle \ and (wp (A;;C) Q o \varphi) = \langle G \rangle \ and (wp (B;;D) (Q o \varphi))\)

proof(simp add:wp-eval

from uCQ pcAB have \(\langle G \rangle \land wp A (wp C Q) o \varphi = \langle G \rangle \land wp B ((wp C Q) o \varphi)\)

by(auto dest:pcorresD)

also have \(\langle G \rangle \land wp B ((wp C Q) o \varphi) = \langle G \rangle \land wp B (wp D (Q o \varphi))\)

proof(rule p3p1)

from uCQ well-def-wp-healthy[OF wB] show unitary (wp B (wp C Q o \varphi))

by(auto intro:unitary-comp

from uDQ well-def-wp-healthy[OF wB] show unitary (wp B (wp D (Q o \varphi)))

by(auto)

from uQ have « I » \& wp B (wp C Q o \varphi) = « I » \& wp B (wp D (Q o \varphi))

by(blast intro:pcorresD[OF pcCD])

thus « I » \& wp B (wp C Q o \varphi) = « I » \& wp B (wp D (Q o \varphi))

by(simp add:p3p2 uCQ uDQ)

qed

finally show \(\langle G \rangle \land (wp A (wp C Q) o \varphi) = \langle G \rangle \land wp B (wp D (Q o \varphi))\)

. qed

4.9.8 Structural Rules for Data Refinement

lemma dr-Skip:

fixes \varphi::c \Rightarrow 'b

shows drefines \varphi G Skip Skip

proof(intro drefinesI2 wd-intros

fix P::'b \Rightarrow real and Q::'b \Rightarrow real and s::'c

assume P ⊨ wp Skip Q

hence \(P \ o \ \varphi\) \ s \leq wp Skip Q (\varphi s) by(simp, blast)
thus \((P \circ \varphi) \leq \text{wp} \text{ Skip} \ (Q \circ \varphi)\) by\(\text{(simp add:wp-evl)}\)

qed

lemma \textit{drefines}\_\textit{Abort}:
fixes \(\varphi : \cdot c \Rightarrow \cdot b\)
shows \(\text{drefines} \ \varphi \ G \text{ Abort} \text{ Abort}\)

proof(\textit{intro drefines}\_\textit{I}2 \textit{wd-intros})
fix \(P : \cdot b \Rightarrow \text{real}\)
and \(Q : \cdot b \Rightarrow \text{real}\)
and \(s : \cdot c\)
assume \(P \vdash \vdash \text{wp} \text{ Abort} \ Q\)
hence \((P \circ \varphi) \leq \text{wp} \text{ Abort} \ Q \ (\varphi \ s)\) by\(\text{(auto)}\)
thus \((P \circ \varphi) \leq \text{wp} \text{ Abort} \ (Q \circ \varphi) \ s\) by\(\text{(simp add:wp-evl)}\)

qed

lemma \textit{drefines}\_\textit{Apply}:
fixes \(\varphi : \cdot c \Rightarrow \cdot b\)
assumes \(\text{commutes}: f \circ \varphi = \varphi \circ g\)
shows \(\text{drefines} \ \varphi \ G \text{ (Apply} f \text{)} \text{ (Apply} g\text{)}\)

proof(\textit{intro drefines}\_\textit{I}2 \textit{wd-intros})
fix \(P : \cdot b \Rightarrow \text{real}\)
and \(Q : \cdot b \Rightarrow \text{real}\)
and \(s : \cdot c\)
assume \(\text{wp}: P \vdash \vdash \text{wp} \text{ (Apply} f \text{)} \ Q\)
hence \(P \vdash (Q \circ f) \text{ by}(\text{simp add:wp-evl})\)
hence \(P \ (\varphi \ s) \leq (Q \circ f) \ (\varphi \ s)\) by\(\text{(blast)}\)
also have \(\ldots = Q ((f \circ \varphi) \ s)\) by\(\text{(simp)}\)
also with \text{commutes}
have \(\ldots = ((Q \circ \varphi) \circ g) \ s\) by\(\text{(simp)}\)
also have \(\ldots = \text{wp} \text{ (Apply} g \text{)} (Q \circ \varphi) \ s\)
by\(\text{(simp add:wp-evl)}\)
finally show \((P \circ \varphi) \ s \leq \text{wp} \text{ (Apply} g \text{)} (Q \circ \varphi) \ s\) by\(\text{(simp)}\)

qed

lemma \textit{drefines}\_\textit{Seq}:
assumes \(\text{drAB}: \text{drefines} \ \varphi \ P \ A \ B\)
and \(\text{drBC}: \text{drefines} \ \varphi \ Q \ C \ D\)
and \(\text{wpB}: «P» \vdash \vdash \text{wp} \ B \ «Q»\)
and \(\text{wB}: \text{well-def} \ B\)
and \(\text{wC}: \text{well-def} \ C\)
and \(\text{wD}: \text{well-def} \ D\)
shows \(\text{drefines} \ \varphi \ P \ (A;;C) \ (B;;D)\)

proof
fix \(R \text{ and} \ S\)
assume \(\text{uR}: \text{unitary} \ R \ \text{and} \ \text{uS}: \text{unitary} \ S\)
and \(\text{wpAC}: R \vdash \vdash \text{wp} \ (A;;C) \ S\)

from \(\text{uR}\)
have \(«P» \ & \ (R \circ \varphi) = «P» \ & \ («P» \ & \ (R \circ \varphi))\)
by\(\text{(simp add:exp-conj-assoc)}\)

also {
from well-def-wp-healthy[OF wC] uR uS
and wpAC[unfolded eval-wp-Seq o-def]
have «P» && (R o φ) ⊢ wp B (wp C S o φ)
by(auto intro:drefinesD[OF drAB])

with wpB well-def-wp-healthy[OF wC] uS
sublinear-sub-conj[OF well-def-wp-sublinear, OF wB]
have «P» && («P» && (R o φ)) ⊢ wp B («Q» && (wp C S o φ))
by(auto intro!:entails-combine dest!:unitary-sound)
}

also {
from uS well-def-wp-healthy[OF wC]
have «Q» && (wp C S o φ) ⊢ wp D (S o φ)
by(auto intro!:drefinesD[OF drBC])

with well-def-wp-healthy[OF wB] well-def-wp-healthy[OF wC]
well-def-wp-healthy[OF wD] and unitary-sound[OF uS]
have wp B («Q» && (wp C S o φ)) ⊢ wp B (wp D (S o φ))
by(blast intro!:mono-transD)
}

finally
show «P» && (R o φ) ⊢ wp (B;;D) (S o φ)
unfolding wp-eval o-def .
qed

lemma dr-repeat:
  fixes φ :: 'a ⇒ 'b
  assumes dr-ab: drefines φ G a b
    and Gpr: «G» ⊢ wp b «G»
    and wa: well-def a
    and wb: well-def b
  shows drefines φ G (repeat n a) (repeat n b) (is ?X n)
proof(induct n)
  show ?X 0 by(simp add:dr-Skip)

    fix n
  assume IH: ?X n
  thus ?X (Suc n) by(auto intro!:dr-Seq Gpr assms wd-intros)
qed

end

4.10 Structured Reasoning

theory StructuredReasoning imports Algebra begin

By linking the algebraic, the syntactic, and the semantic views of computation, we derive a set of rules for decomposing expectation entailment proofs, firstly over the syntactic structure of a program, and secondly over the re-
finement relation. These rules also form the basis for automated reasoning.

### 4.10.1 Syntactic Decomposition

**Lemma wp-Abort:**
\[(\lambda s. 0) \vdash wp \text{ Abort } Q\]
*unfolding* `wp-eval` by `(simp)`

**Lemma wlp-Abort:**
\[(\lambda s. 1) \vdash wlp \text{ Abort } Q\]
*unfolding* `wp-eval` by `(simp)`

**Lemma wp-Skip:**
P \vdash wp Skip P
*unfolding* `wp-eval` by `(blast)`

**Lemma wlp-Skip:**
P \vdash wlp Skip P
*unfolding* `wp-eval` by `(blast)`

**Lemma wp-Apply:**
Q o f \vdash wp (Apply f) Q
*unfolding* `wp-eval` by `(simp)`

**Lemma wlp-Apply:**
Q o f \vdash wlp (Apply f) Q
*unfolding* `wp-eval` by `(simp)`

**Lemma wp-Seq:**
assumes ent-a: P \vdash wp a Q
and ent-b: Q \vdash wp b R
and wa: well-def a
and wb: well-def b
and s-Q: sound Q
and s-R: sound R
shows P \vdash wp (a ;; b) R
*proof*
- *note* ha = well-def-wp-healthy[OF wa]
- *note* hb = well-def-wp-healthy[OF wb]
- *note* ent-a
  - *also from* ent-b ha hb s-Q s-R *have* wp a Q \vdash wp a (wp b R)
    by `(blast intro:healthy-monoD2)`
- *finally show* ?thesis by `(simp add:wp-eval)`
  qed

**Lemma wlp-Seq:**
assumes ent-a: P \vdash wlp a Q
and ent-b: Q \vdash wlp b R
and wa: well-def a
and \( wb \): well-def \( b \)  
and \( u-Q \): unitary \( Q \)  
and \( u-R \): unitary \( R \)  
shows \( P \vdash \text{lwp} \, (a \or b) \, R \)

**proof**

- **note** \( ha = \text{well-def-lwp-nearly-healthy}[OF wa] \)  
- **note** \( hb = \text{well-def-lwp-nearly-healthy}[OF wb] \)  
- **also from** \( \text{ent-b} \, ha \, hb \, u-Q \, u-R \)  
  have \( \text{lwp} \, a \, Q \vdash \text{lwp} \, a \, (\text{lwp} \, b \, R) \)

**finally show** \( \text{thesis} \) **by** \( \text{simp add:wp-eval} \)

**qed**

**lemma** \( \text{wp-PC} \):

\[
(\lambda s. P s \ast \text{wp} a Q s + (1 - P s) \ast \text{wp} b Q s) \vdash \text{wp} \, (a \, (\lambda s. p) \oplus b) \, R
\]

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

**lemma** \( \text{wlp-PC} \):

\[
(\lambda s. P s \ast \text{wlp} a Q s + (1 - P s) \ast \text{wlp} b Q s) \vdash \text{wlp} \, (a \, (\lambda s. p) \oplus b) \, R
\]

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

A simpler rule for when the probability does not depend on the state.

**lemma** \( \text{PC-fixed}\):

**assumes** \( \text{wp}: P \vdash a \, ab \, R \)  
and \( \text{wpb}: Q \vdash b \, ab \, R \)
and \( \text{np}: 0 \leq p \) and \( \text{bp}: p \leq 1 \)
shows \( (\lambda s. p \ast P s + (1 - p) \ast Q s) \vdash (a \, (\lambda s. p) \oplus b) \, ab \, R \)

**unfolding** \( \text{PC-def} \)

**proof** \( \text{rule le-funI} \)

**fix** \( s \)

**from** \( \text{wp} \) and \( \text{np} \) have \( p \ast P s \leq p \ast a \, ab \, R \, s \)

**by** \( \text{auto intro:mult-left-mono} \)

**moreover** \{  
  **from** \( \text{bp} \) have \( 0 \leq 1 - p \) **by** \( \text{simp} \)  
  with \( \text{wpb} \) have \( (1 - p) \ast Q s \leq (1 - p) \ast b \, ab \, R \, s \)
  **by** \( \text{auto intro:mult-left-mono} \)  
\}

**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** \( \text{rule add-mono} \)

**qed**

**lemma** \( \text{wp-PC-fixed}\):

\[
[ P \vdash \text{wp} \, a \, R; \, Q \vdash \text{wp} \, b \, R; \, 0 \leq p; \, p \leq 1 ] \implies \\
(\lambda s. p \ast P s + (1 - p) \ast Q s) \vdash \text{wp} \, (a \, (\lambda s. p) \oplus b) \, R
\]

**by** \( \text{simp add:wp-def PC-fixed} \)

**lemma** \( \text{wlp-PC-fixed}\):

\[
[ P \vdash \text{wlp} \, a \, R; \, Q \vdash \text{wlp} \, b \, R; \, 0 \leq p; \, p \leq 1 ] \implies \\
\]
\[(\lambda s. p * P s + (1 - p) * Q s) \vdash \text{wlp} \ (a (\lambda s. p) \oplus b) \ R\]

by (simp add: wlp-def PC-fixed)

**Lemma wp-DC:**
\[(\lambda s. \text{min} \ (w p a Q s) \ (w p b Q s)) \vdash \text{wp} \ (a \sqcap b) \ Q\]

unfolding wp-eval by (simp)

**Lemma wlp-DC:**
\[(\lambda s. \text{min} \ (w lp a Q s) \ (w lp b Q s)) \vdash \text{wlp} \ (a \sqcap b) \ Q\]

unfolding wp-eval by (simp)

Combining annotations for both branches:

**Lemma DC-split:**
fixes \(a::s\ prog\ and \ b\)
assumes wpa: \(P \vdash a \ ab \ R\)
and wpb: \(Q \vdash b \ ab \ R\)
shows \((\lambda s. \text{min} \ (P s) \ (Q s)) \vdash (a \sqcap b) \ ab \ R\)
unfolding DC-def
proof (rule le-funI)
fix \(s\)
from wpa wpb have \(P s \leq a \ ab \ R s\) and \(Q s \leq b \ ab \ R s\) by (auto)
thus \(\text{min} \ (P s) \ (Q s) \leq \text{min} \ (a \ ab \ R s) \ (b \ ab \ R s)\) by (auto)
qed

**Lemma wp-DC-split:**
\[[ P \vdash wp \ prog \ R; Q \vdash wp \ prog' \ R \] \implies \n(\lambda s. \text{min} \ (P s) \ (Q s)) \vdash \text{wp} \ (prog \sqcap prog') \ R\]
by (simp add: wp-def DC-split)

**Lemma wlp-DC-split:**
\[[ P \vdash wlp \ prog \ R; Q \vdash wlp \ prog' \ R \] \implies \n(\lambda s. \text{min} \ (P s) \ (Q s)) \vdash \text{wlp} \ (prog \sqcap prog') \ R\]
by (simp add: wlp-def DC-split)

**Lemma wp-DC-split-same:**
\[[ P \vdash wp \ prog \ Q; P \vdash wp \ prog' \ Q \] \implies \n\vdash \text{wp} \ (prog \sqcap prog') \ Q\]
unfolding wp-eval by (blast intro:min.boundedI)

**Lemma wlp-DC-split-same:**
\[[ P \vdash wlp \ prog \ Q; P \vdash wlp \ prog' \ Q \] \implies \n\vdash \text{wlp} \ (prog \sqcap prog') \ Q\]
unfolding wp-eval by (blast intro:min.boundedI)

**Lemma SetPC-split:**
fixes \(f::'x \Rightarrow 'y\ prog\)
and \(p::'y \Rightarrow 'x \Rightarrow \text{real}\)
assumes rec: \(\forall x \ s. \ x \in \supp \ (p s) \Rightarrow P x \vdash f x \ ab \ Q\)
and nnp: \(\forall x. \ \text{nnp} \ (p s)\)
shows \((\lambda s. \sum x \in \supp \ (p s). \ p s x \ * \ P x s) \vdash \text{SetPC} \ f p \ ab \ Q\)
4.10. STRUCTURED REASONING

unfolding SetPC-def
proof (rule le-funI)
  fix s
  from rec have \( \forall x. x \in \text{supp} (p s) \implies P \ x s \leq f x \ ab Q s \) by (blast)
  moreover from mnp have \( \forall x. 0 \leq p s x \) by (blast)
  ultimately have \( \forall x. x \in \text{supp} (p s) \implies p s x * P \ x s \leq p s x * f x \ ab Q s \)
    by (blast intro: mult-left-mono)
  thus \( (\sum x \in \text{supp} (p s). p s x * P \ x s) \leq (\sum x \in \text{supp} (p s). p s x * f x \ ab Q s) \)
    by (rule sum-mono)
qed

lemma wp-SetPC-split:
  \[ \forall x. x \in \text{supp} (p s) \implies P \ x s \vdash \ wp (f x) \ Q; \forall s. \text{nneg} (p s) \] \implies
  \( (\lambda s. \sum x \in \text{supp} (p s). p s x * P \ x s) \vdash \ wp (\text{SetPC f p}) \ Q \)
  by (simp add: wp-def SetPC-split)

lemma wlp-SetPC-split:
  \[ \forall x. x \in \text{supp} (p s) \implies P \ x s \vdash \ wlp (f x) \ Q; \forall s. \text{nneg} (p s) \] \implies
  \( (\lambda s. \sum x \in \text{supp} (p s). p s x * P \ x s) \vdash \ wlp (\text{SetPC f p}) \ Q \)
  by (simp add: wp-def SetPC-split)

lemma wp-SetDC-split:
  \[ \forall s. x \in S s \implies P \vdash \ wp (f x) \ Q; \forall s. S s \neq \emptyset \] \implies
  \( P \vdash \ wp (\text{SetDC f s}) \ Q \)
  by (rule le-funI, unfold wp-eval, blast intro!: cInf-greatest)

lemma wlp-SetDC-split:
  \[ \forall s. x \in S s \implies P \vdash \ wlp (f x) \ Q; \forall s. S s \neq \emptyset \] \implies
  \( P \vdash \ wlp (\text{SetDC f s}) \ Q \)
  by (rule le-funI, unfold wp-eval, blast intro!: cInf-greatest)

lemma wp-SetDC:
  assumes \( wp: \forall s. x \in S s \implies P \vdash \ wp (f x) \ Q \)
    and \( nc: \forall s. S s \neq \emptyset \)
    and \( sp: \forall x. \text{sound} (P x) \)
  shows \( (\lambda s. \text{Inf} ((\lambda x. P x s) \cdot S s)) \vdash \ wp (\text{SetDC f s}) \ Q \)
  using assms by (intro le-funI, simp add: wp-eval, blast intro!: cInf-monotone)

lemma wlp-SetDC:
  assumes \( wp: \forall s. x \in S s \implies P \vdash \ wlp (f x) \ Q \)
    and \( nc: \forall s. S s \neq \emptyset \)
    and \( sp: \forall x. \text{sound} (P x) \)
  shows \( (\lambda s. \text{Inf} ((\lambda x. P x s) \cdot S s)) \vdash \ wlp (\text{SetDC f s}) \ Q \)
  using assms by (intro le-funI, simp add: wp-eval, blast intro!: cInf-monotone)

lemma wp-Embed:
  \( P \vdash t Q \implies P \vdash \ wp (\text{Embed t}) \ Q \)
  by (simp add: wp-def Embed-def)
**Lemma wlp-Embed:**

\[ P \vdash t \ Q \implies P \vdash \text{wlp} (\text{Embed} \ t) \ Q \]

by(simp add:wlp-def Embed-def)

**Lemma wp-Bind:**

\[
\left[ \forall s. \ P s \leq \text{wp} (a \ (f \ s)) \ Q \ s \right] \implies P \vdash \text{wp} (\text{Bind} \ f \ a) \ Q
\]

by(auto simp:wp-def Bind-def)

**Lemma wlp-Bind:**

\[
\left[ \forall s. \ P s \leq \text{wlp} (a \ (f \ s)) \ Q \ s \right] \implies P \vdash \text{wlp} (\text{Bind} \ f \ a) \ Q
\]

by(auto simp:wlp-def Bind-def)

**Lemma wp-repeat:**

\[
\left[ P \vdash \text{wp} \ a \ Q ; \ Q \vdash \text{wp} (\text{repeat} \ n \ a) \ R; \ well-def \ a; \ sound \ Q; \ sound \ R \right] \implies P \vdash \text{wp} (\text{repeat} \ (\text{Suc} \ n) \ a) \ R
\]

by(auto intro!:wp-Seq wd-intros)

**Lemma wlp-repeat:**

\[
\left[ P \vdash \text{wlp} \ a \ Q ; \ Q \vdash \text{wlp} (\text{repeat} \ n \ a) \ R; \ well-def \ a; \ unitary \ Q; \ unitary \ R \right] \implies P \vdash \text{wlp} (\text{repeat} \ (\text{Suc} \ n) \ a) \ R
\]

by(auto intro!:wlp-Seq wd-intros)

Note that the loop rules presented in section Section 4.8 are of the same form, and would belong here, had they not already been stated.

The following rules are specialisations of those for general transformers, and are easier for the unifier to match.

**Lemmas wp-strengthen-post=**

\[ \text{entails-strengthen-post}[\text{where } t=\text{wp} \ a \ for \ a] \]

**Lemma wlp-strengthen-post:**

\[ P \vdash \text{wlp} \ a \ Q \implies \text{nearly-healthy} (\text{wlp} \ a) \implies \text{unitary} \ R \implies Q \vdash R \implies \text{unitary} \]

\[ P \vdash \text{wlp} \ a \ R \]

by(blast intro:entails-trans)

**Lemmas wp-weaken-pre=**

\[ \text{entails-weaken-pre}[\text{where } t=\text{wp} \ a \ for \ a] \]

**Lemmas wlp-weaken-pre=**

\[ \text{entails-weaken-pre}[\text{where } t=\text{wlp} \ a \ for \ a] \]

**Lemmas wp-scale=**

\[ \text{entails-scale}[\text{where } t=\text{wp} \ a \ for \ a, \ OF - \text{well-def-wp-healthy}] \]

### 4.10.2 Algebraic Decomposition

Refinement is a powerful tool for decomposition, belied by the simplicity of the rule. This is an axiomatic formulation of refinement (all annotations of
the $a$ are annotations of $b$), rather than an operational version (all traces of $b$ are traces of $a$).

**lemma** wp-refines:

\[
\{ a \sqsubseteq b; P \vdash wp a Q \}: P \vdash wp b Q
\]

\[
\text{by}(\text{auto intro: entails-trans})
\]

**lemmas** wp-drefines = drefinesD

### 4.10.3 Hoare triples

The Hoare triple, or validity predicate, is logically equivalent to the weakest-precondition entailment form. The benefit is that it allows us to define transitivity rules for computational (also/finally) reasoning.

**definition**

wp-valid :: $(a \Rightarrow \text{real}) \Rightarrow (a \Rightarrow \text{real}) \Rightarrow \text{bool} (\{\_\} - \{\_\} p)$

**where**

wp-valid $P$ $\text{prog}$ $Q$ $\equiv$ $P \vdash wp$ $\text{prog}$ $Q$

**lemma** wp-validI:

\[
P \vdash wp \text{prog} Q \Rightarrow \{P\} \text{prog} \{Q\} p
\]

\[
\text{unfolding} \quad \text{wp-valid-def} \quad \text{by}(\text{assumption})
\]

**lemma** wp-validD:

\[
\{P\} \text{prog} \{Q\} p \Rightarrow P \vdash wp \text{prog} Q
\]

\[
\text{unfolding} \quad \text{wp-valid-def} \quad \text{by}(\text{assumption})
\]

**lemma** valid-Seq:

\[
\{P\} \; a \; \{Q\} p; \; b \; \{R\} p \; ; \; \text{well-def } a; \; \text{well-def } b; \; \text{sound } Q; \; \text{sound } R \Rightarrow \{P\} \; a \; ;; \; b \; \{R\} p
\]

\[
\text{unfolding} \quad \text{wp-valid-def} \quad \text{by}(\text{rule wp-Seq})
\]

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 \( ws :: well-def s \)
- **shows** \((\lambda s. 1) \vdash wp (r ;; s) (\lambda s. 1)\)

**proof** –

- **note** \( hs = well-def-wp-healthy[OF ws]\)
- **have** \( wp s (\lambda s. 1) = (\lambda s. 1)\)

**proof (rule antisym)**

- **show** \((\lambda s. 1) \vdash wp s (\lambda s. 1)\) **by (rule ts)**
- **have** bounded-by \( I (wp s (\lambda s. 1))\)
  **by (auto intro!:healthy-bounded-byD[OF hs])**
- **thus** \( wp s (\lambda s. 1) \vdash (\lambda s. 1)\) **by (auto intro!:le-funI)**

**qed**

**with** \( mr \) **show** \(?thesis**

- **by (simp add:wp-eval embed-bool-def maximalD)**

**qed**

From any state where the guard does not hold, a loop terminates in a single step.

**lemma term-onestep:**

- **assumes** \( wb :: well-def body \)
- **shows** \( \langle N G \rangle \vdash wp do G \dashrightarrow body od (\lambda s. 1)\)

**proof (rule le-funI)**

- **note** \( hb = well-def-wp-healthy[OF wb]\)
- **fix** \( s\)
- **show** \( \langle N G \rangle s \le wp do G \dashrightarrow body od (\lambda s. 1) s\)

**proof (cases G s, simp-all add:wp-loop-nguard hb)**

- **from** \( hb \) **have** sound \((wp do G \dashrightarrow body od (\lambda s. 1))\)
  **by (auto intro!:healthy-sound[OF healthy-wp-loop])**
- **thus** \( 0 \le wp do G \dashrightarrow body od (\lambda s. 1) s\) **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** \( m :: s \Rightarrow nat \) and \( body :: s, prog \)
- **assumes** \( wb :: well-def body \)
- and \( guard :: \forall s. m s = 0 \dashrightarrow \neg G s\)
and variant: \( \forall n. \langle \lambda s. m s = \text{Suc } n \rangle \vdash wp \text{ body } \langle \lambda s. m s = n \rangle \)
shows \( \lambda s. 1 \vdash wp \text{ do } G \rightarrow \text{ body od } (\lambda s. 1) \)
proof –

note \( hb = \text{well-def-wp-healthy[OF wb]} \)
have \( \forall n. (\forall s. m s = n \rightarrow 1 \leq wp \text{ do } G \rightarrow \text{ body od } (\lambda s. 1) s) \)
proof (induct-tac n)

fix \( n \)
show \( \forall s. m s = 0 \rightarrow 1 \leq wp \text{ do } G \rightarrow \text{ body od } (\lambda s. 1) s \)
proof (clarify)
fix \( s \)
assume \( m s = 0 \)
with guard have \( \neg G s \) by (blast)
with \( hb \) show \( 1 \leq wp \text{ do } G \rightarrow \text{ body od } (\lambda s. 1) s \)
by (simp add:wp-loop-nguard)

qed

 assume \( IH: \forall s. m s = n \rightarrow 1 \leq wp \text{ do } G \rightarrow \text{ body od } (\lambda s. 1) s \)
hence \( IH': \forall s. m s = n \rightarrow 1 \leq wp \text{ do } G \rightarrow \text{ body od } \langle \lambda s. \text{True} \rangle s \)
by (simp add:embed-bool-def)

have \( \forall s. m s = \text{Suc } n \rightarrow 1 \leq wp \text{ do } G \rightarrow \text{ body od } \langle \lambda s. \text{True} \rangle s \)
proof (intro fold-premise healthy-intros hb, rule le-funI)

fix \( s \)
show \( \langle \lambda s. m s = \text{Suc } n \rangle s \leq wp \text{ do } G \rightarrow \text{ body od } \langle \lambda s. \text{True} \rangle s \)
proof (cases G s)

case False
hence \( 1 = \langle \text{N } G \rangle s \) by (auto)
also from \( wb \) have \( ... \leq wp \text{ do } G \rightarrow \text{ body od } (\lambda s. 1) s \)
by (rule le-funD[OF term-onestep])
finally show ?thesis by (simp add:embed-bool-def)

next
case True note \( G = \text{this} \)
from \( IH' \) have \( \langle \lambda s. m s = n \rangle \vdash wp \text{ do } G \rightarrow \text{ body od } \langle \lambda s. \text{True} \rangle \)
by (blast intro:use-premise healthy-intros hb)
with variant \( wb \)
have \( \langle \lambda s. m s = \text{Suc } n \rangle \vdash wp \text{ (body ;; do G } \rightarrow \text{ body od) } \langle \lambda s. \text{True} \rangle \)
by (blast intro:wp-Seq ad-intros)
hence \( \langle \lambda s. m s = \text{Suc } n \rangle s \leq wp \text{ (body ;; do G } \rightarrow \text{ body od) } \langle \lambda s. \text{True} \rangle s \)
by (auto)
also from \( hb \) \( G \) have \( ... = wp \text{ do } G \rightarrow \text{ body od } \langle \lambda s. \text{True} \rangle s \)
by (simp add:wp-loop-guard)
finally show ?thesis .

qed

qed

thus \( \forall s. m s = \text{Suc } n \rightarrow 1 \leq wp \text{ do } G \rightarrow \text{ body od } (\lambda s. 1) s \)
by (simp add:embed-bool-def)

qed

thus ?thesis by (auto)

qed

This version allows progress to depend on an invariant. Termination is then
determined by the invariant’s value in the initial state.

**Lemma loop-term-nat-measure:**

**Fixes** \( m :: 's \Rightarrow \text{nat} \) and \( \text{body} :: 's \text{ prog} \)

**Assumes** \( \text{wb} :: \text{well-def body} \)

and guard: \( \forall s. m \, s = 0 \rightarrow G \, s \)

and variant: \( \forall n. \:\lambdas. \, m \, s = \text{Suc} \, n \) \&\& \( \langle I \rangle \vdash \text{wp body} \:\lambdas. \, m \, s = n \)

and \( \text{inv} :: \text{wp-inv G body} \langle I \rangle \)

shows \( \langle I \rangle \vdash \text{wp do G} \rightarrow \text{body od} \:\lambdas. \, \text{True} \)

**Proof**

*note* \( \text{hb} = \text{well-def-wp-healthy}[\text{OF wb}] \)

*note* \( \text{scb} = \text{sublinear-sub-conj}(\text{OF well-def-wp-sublinear}, \text{OF wb}) \)

**Have** \( \langle I \rangle \vdash \text{wp do G} \rightarrow \text{body od} \:\lambdas. \, \text{True} \)

**Proof** (rule use-premise, intro healthy-intros \( \text{hb} \))

**Fix** \( s \)

**Have** \( \forall n. (\forall s. m \, s = n \wedge I \, s \rightarrow 1 \leq \text{wp do G} \rightarrow \text{body od} \:\lambdas. \, \text{True} \) \( s \)

**Proof** (induct-tac \( n \))

**Fix** \( n \)

**Show** \( \forall s. m \, s = 0 \wedge I \, s \rightarrow 1 \leq \text{wp do G} \rightarrow \text{body od} \:\lambdas. \, \text{True} \) \( s \)

**Proof** (clarify)

**Fix** \( s \)

**Assume** \( m \, s = 0 \)

with guard **Have** \( \neg \, G \, s \) **by(blast)**

with \( \text{hb} \) **Show** \( I \) \( \leq \) **by** (simp add: wp-loop-nguard)

**QED**

**Assume** \( \forall s. m \, s = n \wedge I \, s \rightarrow 1 \leq \text{wp do G} \rightarrow \text{body od} \:\lambdas. \, \text{True} \) \( s \)

**Show** \( \forall s. m \, s = \text{Suc} \, n \wedge I \, s \rightarrow 1 \leq \text{wp do G} \rightarrow \text{body od} \:\lambdas. \, \text{True} \) \( s \)

**Proof** (intro fold-premise healthy-intros \( \text{hb} \) le-funI)

**Fix** \( s \)

**Show** \( \:\lambdas. \, m \, s = \text{Suc} \, n \wedge I \, s \) \( \leq \) **by** (simp add: wp-loop-nguard)

**Next**

**Case** \( \text{False} \) **with** \( \text{hb} \) **Show** \( \) \?

**By** (simp add: wp-loop-nguard)

**Case** \( \text{True} \) **Note** \( G = \) **this**

**Have** \( \:\lambdas. \, m \, s = \text{Suc} \, n \) \&\& \( \langle I \rangle \) \&\& \( \langle G \rangle = \)

\( \:\lambdas. \, m \, s = \text{Suc} \, n \) \&\& \( \langle I \rangle \) \&\& \( \langle I \rangle \) \&\& \( \langle G \rangle \)

**By** (simp)

**Also Have** \( \ldots = \langle \:\lambdas. \, m \, s = \text{Suc} \, n \rangle \) \&\& \( \langle I \rangle \rangle \&\& \( \langle G \rangle \rangle \)

**By** (simp add: exp-conj-assoc exp-conj-unitary del: exp-conj-idem)

**Also Have** \( \ldots = \langle \:\lambdas. \, m \, s = \text{Suc} \, n \rangle \) \&\& \( \langle I \rangle \rangle \&\& \( \langle G \rangle \) \&\& \( \langle I \rangle \)

**By** (simp only: exp-conj-comm)

**Also** \{ \}

from inv \( \text{hb} \) **Have** \( \langle G \rangle \) \&\& \( \langle I \rangle \vdash \text{wp body} \langle I \rangle \)

**By** (rule wp-inv-stdD)

with variant

**Have** \( \langle \:\lambdas. \, m \, s = \text{Suc} \, n \rangle \) \&\& \( \langle I \rangle \rangle \&\& \( \langle G \rangle \rangle \&\& \( \langle I \rangle \)

**By** (rule entails-frame)
4.11.1 LOOP TERMINATION

{ }
also from scb
have wp body «λs. m s = n» & & wp body «I» ⊢
wp body («λs. m s = n» & & «I»)
by (blast)
finally have «λs. m s = Suc n» & & «I» & & «G» ⊢
wp body («λs. m s = n» & & «I»).
moreover {
from IH have «λs. m s = n» & & «I» ⊢ wp do G → body od «λs. True»
by (blast intro:use-premise healthy-intros hb)
hence «λs. m s = n» & & «I» ⊢ wp do G → body od «λs. True»
by (simp add: exp-conj-std-split)
}
ultimately have «λs. m s = Suc n» & & «I» & & «G» ⊢
wp (body ;; do G → body od) «λs. True»
using wb by (blast intro: wp-Seq wd-intros)
hence («λs. m s = Suc n ∧ I s» & & «G») s ≤
wp (body ;; do G → body od) «λs. True» s
by (auto simp: exp-conj-std-split)
with G have «λs. m s = Suc n ∧ I s» s ≤
wp (body ;; do G → body od) «λs. True» s
by (simp add: exp-conj-def)
also from hb G have «... = wp do G → body od «λs. True» s
by (simp add: wp-loop-guard)
finally show ?thesis.
qed
qed
qed
moreover assume I s
ultimately show I ≤ wp do G → body od «λs. True» s
by (auto)
qed
thus ?thesis by (simp add: embed-bool-def)
qed

4.11.3 Probabilistic Termination

Any loop that has a non-zero chance of terminating after each step terminates with probability 1.

lemma termination-0-1:
  fixes body :: 's prog
  assumes wb: well-def body
  — The loop terminates in one step with nonzero probability
  and onestep: (λs. p) ⊢ wp body «N G»
  and nzp: 0 < p
  — The body is maximal i.e. it terminates absolutely.
  and mb: maximal (wp body)
  shows λs. 1 ⊢ wp do G → body od (λs. 1)
proof

note \(hb = \text{well-def-wp-healthy}[\text{OF wb}]\)

note \(sh = \text{healthy-scalingD}[\text{OF hb}]\)

note \(sab = \text{sublinear-subadd}[\text{OF well-def-wp-sublinear}, \text{OF wb}, \text{OF healthy-feasibleD}, \text{OF hb}]\)

from \(hb\) have \(hloop: \text{healthy}(\text{wp do } G \rightarrow \text{body od})\)
by (rule healthy-intros)

hence \(swp: \text{sound}(\text{wp do } G \rightarrow \text{body od} (\lambda s. 1))\) by (blast)

\(p\) is no greater than \(1\), by feasibility.

from \(onestep\) have \(onestep\': \forall s. \, p \leq \text{wp body } \langle N \rangle G\ s\) by (auto)
also { from \(hb\) have \(unitary(\text{wp body } \langle N \rangle G)\) by (auto)
  hence \(\forall s. \, \text{wp body } \langle N \rangle G\ s \leq 1\) 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 \(new-bound: \forall k. \, 0 \leq k \implies k \leq 1 \implies (\lambda s. k) \vdash \text{wp do } G \rightarrow \text{body od} (\lambda s. 1)\) 
proof (rule le-funI)
fix \(k\ s\)
assume \(X: \lambda s. k \vdash \text{wp do } G \rightarrow \text{body od} (\lambda s. 1)\)
  and \(k0: 0 \leq k\) and \(k1: k \leq 1\)
from \(k1\) have \(nz1k: 0 \leq 1 - k\) 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 \(p \ast (1 - k) + k \leq \langle N \rangle G\ s + \langle G \rangle s \ast (p \ast (1 - k) + k)\)
by (cases \(G\ s, \text{simp-all}\))

By the one-step termination assumption:
also from \(onestep\' \, nz1k\)
have \(\ldots \leq \langle N \rangle G\ s + \langle G \rangle s \ast (\text{wp body } \langle N \rangle G\ s \ast (1 - k) + k)\)
by (simp add: mult-right-mono ordered-comm-semiring-class.comm-mult-left-mono)

By scaling:
also from \(nz1k\)
have \(\ldots = \langle N \rangle G\ s + \langle G \rangle s \ast (\text{wp body } (\lambda s. \langle N \rangle G\ s \ast (1 - k)) s + k)\)
by (simp add: right-scalingD[\text{OF sb}])

By the maximality (termination) of the loop body:
4.11. LOOP TERMINATION

also from mb k0
have ... = \langle N G \rangle s + \langle G \rangle s * (wp body (\lambda s. \langle N G \rangle s * (1-k)) s + wp body (\lambda s. k) s) 
by(simp add:maximalD)

By sub-additivity of the loop body:

also from k0 nz1k
have ... \leq \langle N G \rangle s + \langle G \rangle s * (wp body (\lambda s. \langle N G \rangle s * (1-k) + k) s) 
by(auto intro!:add-left-mono multi-left-mono sub-addD[OF sub] sound-intros)
also
have ... = \langle N G \rangle s + \langle G \rangle s * (wp body (\lambda s. \langle N G \rangle s + \langle G \rangle s * k) s) 
by(simp add:negate-embed algebra-simps)

By monotonicity of the loop body, and that k is a lower bound:

also from k0 bloop le-funD[OF X]
have ... \leq \langle N G \rangle s + \langle G \rangle s * (wp body (\lambda s. \langle N G \rangle s + \langle G \rangle s * wp do G \rightarrow body od (\lambda s. 1) s) s) 
by(iprover intro:add-left-mono multi-left-mono le-funI embed-ge-0
le-funD[OF mono-transD, OF healthy-monoD, OF hb]
sound-sum standard-sound sound-intros swp)

Unrolling the loop once and simplifying:

also {
have \lambda s. \langle N G \rangle s + \langle G \rangle s * wp body (wp do G \rightarrow body od (\lambda s. 1)) s = 
\langle N G \rangle s + \langle G \rangle s * (\langle N G \rangle s + \langle G \rangle s * wp body (wp do G \rightarrow body od (\lambda s. 1) s) s) 
by(simp only:distrib-left mult.assoc[symmetric] embed-bool-idem embed-bool-cancel)
also have \lambda s. ... s = \langle N G \rangle s + \langle G \rangle s * wp do G \rightarrow body od (\lambda s. 1) s 
by(simp add:fun-cong[OF wp-loop-unfold[symmetric, where P=\lambda s. 1, simplified, OF hb]]]
finally have X: \lambda s. \langle N G \rangle s + \langle G \rangle s * wp body (wp do G \rightarrow body od (\lambda s. 1)) s = 
\langle N G \rangle s + \langle G \rangle s * wp do G \rightarrow body od (\lambda s. 1) s .
have \langle N G \rangle s + \langle G \rangle s * (wp body (\lambda s. \langle N G \rangle s + \langle G \rangle s * wp do G \rightarrow body od (\lambda s. 1) s) s) = 
\langle N G \rangle s + \langle G \rangle s * (wp body (\lambda s. \langle N G \rangle s + \langle G \rangle s * wp body (wp do G \rightarrow body od (\lambda s. 1)) s) s) 
by(simp only:X)
}

Lastly, by folding two loop iterations:

also
have \langle N G \rangle s + \langle G \rangle s * (wp body (\lambda s. \langle N G \rangle s + \langle G \rangle s * wp body (wp do G \rightarrow body od (\lambda s. 1)) s) s) = 
wp do G \rightarrow body od (\lambda s. 1) s 
by(simp add:wp-loop-unfold[OF - hb, where P=\lambda s. 1, simplified, symmetric]
fun-cong[OF wp-loop-unfold[OF - hb, where P=\lambda s. 1, simplified, symmetric]]]}
CHAPTER 4. THE PGCL LANGUAGE

Finally show \( p \ast (1 - k) + k \leq \text{wp do } G \longrightarrow \text{body od } \langle \lambda s. 1 \rangle \) s.

Qed

If the previous bound lay in \([0, 1)\), the new bound is strictly greater. This is where we appeal to the fact that \( p \) is nonzero.

From \( \text{nzp have inc: } \forall k. 0 \leq k \Rightarrow k < 1 \Rightarrow k < p \ast (1 - k) + k \)

By (auto intro: mult-pos-pos)

The result follows by contradiction.

Show \( \text{thesis} \)

Proof (rule ccontr)

If the loop does not terminate everywhere, then there must exist some state from which the probability of termination is strictly less than one.

Assume \( \neg \text{thesis} \)

Hence \( \neg (\forall s. 1 \leq \text{wp do } G \longrightarrow \text{body od } \langle \lambda s. 1 \rangle \) s by (auto)

Then obtain \( s \) where point: \( \neg 1 \leq \text{wp do } G \longrightarrow \text{body od } \langle \lambda s. 1 \rangle \) s by (auto)

Let \( ?k = \text{Inf } \langle \text{range } (\text{wp do } G \longrightarrow \text{body od } \langle \lambda s. 1 \rangle) \rangle \)

From \( \text{hloop} \)

Have \( \text{Inflb: } \forall s. ?k \leq \text{wp do } G \longrightarrow \text{body od } \langle \lambda s. 1 \rangle \) s

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

Also from point have \( \text{wp do } G \longrightarrow \text{body od } \langle \lambda s. 1 \rangle \) s < 1 by (auto)

Thus the least (infimum) probability of termination is strictly less than one.

Finally have \( ?k < 1 \)

Hence \( ?k \leq 1 \) by (auto)

Moreover from \( \text{hloop} \) have \( k0 \): \( 0 \leq ?k \)

By (intro cInf-greatest, auto)

The infimum is, naturally, a lower bound.

Moreover from \( \text{Inflb} \) have \( (\lambda s. ?k) \vdash \text{wp do } G \longrightarrow \text{body od } \langle \lambda s. 1 \rangle \) by (auto)

Ultimately

We can therefore use the previous result to find a new bound, ...

Have \( \forall s. p \ast (1 - ?k) + ?k \leq \text{wp do } G \longrightarrow \text{body od } \langle \lambda s. 1 \rangle \) s

By (blast intro: le-funD[OF new-bound])

... which is lower than the infimum, by minimality, ...

Hence \( p \ast (1 - ?k) + ?k \leq ?k \)

By (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 \( \text{False} \) by (simp)
4.12 Automated Reasoning

theory Automation imports StructuredReasoning begin

This theory serves as a container for automated reasoning tactics for pGCL, implemented in ML. At present, there is a basic verification condition generator (VCG).

named-theorems wd
  theorems to automatically establish well-definedness

named-theorems pwp-core
  core probabilistic wp rules, for evaluating primitive terms

named-theorems pwp
  user-supplied probabilistic wp rules

named-theorems pwlp
  user-supplied probabilistic wlp rules

ML-file pVCG.ML

method-setup pvcg =
  ⟨⟨ Scan.succeed (fn ctxt => SIMPLE-METHOD' (pVCG.pVCG-tac ctxt)) ⟩⟩
Probabilistic weakest preexpectation tactic

declare wd-intros[wd]

lemmas core-wp-rules =
  wp-Skip     wlp-Skip
  wp-Abort    wlp-Abort
  wp-Apply    wlp-Apply
  wp-Seq      wlp-Seq
  wp-DC-split wlp-DC-split
  wp-PC-fixed wlp-PC-fixed
  wp-SetDC    wlp-SetDC
  wp-SetPC-split wlp-SetPC-split

declare core-wp-rules[pwp-core]

end
Additional Material

4.13 Miscellaneous Mathematics

theory Misc
imports
  HOL-Analysis.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 \( \text{thesis} \).
qed

lemma cInf-mono:
  fixes A::'a::conditionally-complete-lattice set
  assumes lower: \( \forall b. b \in B \implies \exists a \in A. a \leq b \)
    and bounded: \( \forall a. a \in A \implies c \leq a \)
    and ne: \( B \neq \{\} \)
  shows \( \inf A \leq \inf B \)
proof(rule cInf-greatest[OF ne])
  fix b assume bin: \( b \in B \)
  with lower obtain a where ain: \( a \in A \) and le: \( a \leq b \) by(auto)
  from ain bounded have \( \inf A \leq a \) by(intro cInf-lower bdd-belowI, auto)
  also note le
  finally show \( \inf A \leq b \).
qed

lemma max-distrib:
  fixes c::real
  assumes nn: \( 0 \leq c \)
  shows \( c \cdot \max a b = \max (c \cdot a) (c \cdot b) \)
proof(cases a \leq b)
  case True
  moreover with nn have \( c \cdot a \leq c \cdot b \) by(auto intro:mult-left-mono)
  ultimately show \( \text{thesis} \) by(simp add:max.absorb2)

219
next
  case False then have \( b \leq a \) by (auto)
moreover with nn have \( c \cdot b \leq c \cdot a \) by (auto intro: mult-left-mono)
ultimately show ?thesis by (simp add: max.absorb1)
qed

lemma mult-div-mono-left:
  fixes c::real
  assumes nnc: \( 0 \leq c \) and nzc: \( c \neq 0 \)
    and inv: \( a \leq \text{inverse} c \cdot b \)
  shows \( c \cdot a \leq b \)
proof (cases a \leq b)
  case True
    moreover with nnc have \( c \cdot a \leq (c \cdot \text{inverse} c) \cdot b \)
    by (auto simp: mult.assoc intro: mult-left-mono)
also from nzc have \( \ldots = b \) by (simp)
finally show \( c \cdot a \leq b \).
qed

lemma mult-div-mono-right:
  fixes c::real
  assumes nnc: \( 0 \leq c \) and nzc: \( c \neq 0 \)
    and inv: \( \text{inverse} c \cdot a \leq b \)
  shows \( a \leq c \cdot b \)
proof (cases a \leq c)
  case False
    hence \( b \leq a \) by (auto)
moreover with nnc have \( c \cdot b \leq c \cdot a \)
    by (auto intro: mult-left-mono)
ultimately show ?thesis by (simp add: min.absorb2)
qed

lemma min-distrib:
  fixes c::real
  assumes nnc: \( 0 \leq c \)
  shows \( c \cdot \text{min} a b = \text{min} (c \cdot a) (c \cdot b) \)
proof (cases a \leq b)
  case True
    moreover with nnc have \( c \cdot a \leq c \cdot b \)
    by (blast intro: mult-left-mono)
ultimately show ?thesis by (auto)
next
  case False
    hence \( b \leq a \) by (auto)
moreover with nnc have \( c \cdot b \leq c \cdot a \)
    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: \( S \neq \{\} \)
shows $\exists x \in S. \forall y \in S. \ x \leq y$
proof
have $S = {} \lor (\exists x \in S. \forall y \in S. \ x \leq y)$
proof\(\text{rule finite-induct, simp-all add:assms}\)
fix \(a\) and \(S: \text{a set}\)
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\(\text{cases S={}}\)
\(\text{case True then show } \text{thesis by(auto)}\)
next
\(\text{case False with } IH \text{ have } \exists x \in S. \forall y \in S. \ x \leq y \text{ by(auto)}\)
thus \(\text{thesis by(cases } z \leq x,\text{ auto)}\)
qed

lemma \(c\text{Sup-add}\):
fixes \(c::\text{real}\)
assumes \(ne: S \neq {}\)
\(\text{and } bS: \forall x. \ x \in S \implies x \leq b\)
shows \(\text{Sup } S + c = \text{Sup } \{x + c \mid x \in S\}\)
proof\(\text{rule antisym}\)
from \(ne\ bS\) show \(\text{Sup } \{x + c \mid x \in S\} \leq \text{Sup } S + c\)
by\((\text{auto intro: cSup-least add-right-mono cSup-upper bdd-aboveI})\)
have \(\text{Sup } S \leq \text{Sup } \{x + c \mid x \in S\} - c\)
proof\(\text{intro cSup-least ne}\)
fix \(x\) assume \(xin: x \in S\)
from \(bS\) have \(\forall x. \ x \in S \implies x + c \leq b + c\) by\((\text{auto intro:add-right-mono})\)
hence \(\text{bdd-above } \{x + c \mid x \in S\} \text{ by(intro bdd-aboveI, blast)}\)
with \(xin\) have \(x + c \leq \text{Sup } \{x + c \mid x \in S\}\) by\((\text{auto intro:cSup-upper})\)
thus \(x \leq \text{Sup } \{x + c \mid x \in S\} - c\) by\((\text{auto})\)
qed
thus \(\text{Sup } S + c \leq \text{Sup } \{x + c \mid x \in S\}\) by\((\text{auto})\)
qed

lemma \(c\text{Sup-mult}\):
fixes \(c::\text{real}\)
assumes \(ne: S \neq {}\)
\(\text{and } bS: \forall x. \ x \in S \implies x \leq b\)
\(\text{and } nnc: 0 \leq c\)
shows \(c \ast \text{Sup } S = \text{Sup } \{c \ast x \mid x \in S\}\)
proof\(\text{cases}\)
assume \(c = 0\)
moreover from \(ne\) have \(\exists x. \ x \in S\) by\((\text{auto})\)
ultimately show \(\text{thesis by(simp)}\)
next
assume \( cnz: c \neq 0 \)

show \( ?thesis \)

proof (rule antisym)
  from \( bS \) have \( baS: \ bdd-above \ S \) by (intro bdd-aboveI, auto)
  with \( ne \) \( nnc \) show \( \Sup \ \{ c \ast x \mid x \in S \} \leq c \ast \Sup S \)
    by (blast intro: cSup-least mult-left-mono [OF cSup-upper])
  have \( \Sup S \leq \inverse c \ast \Sup \ \{ c \ast x \mid x \in S \} \)
    proof (intro cSup-least ne)
      fix \( x \) assume \( \xin: x \in S \)
      moreover from \( bS \) \( nnc \) have \( \bigwedge x. x \in S = \Rightarrow c \ast x \leq c \ast b \)
        by (auto intro: mult-left-mono)
      ultimately have \( c \ast x \leq \Sup \ \{ c \ast x \mid x \in S \} \)
        by (auto intro: cSup-upper bdd-aboveI)
    qed
  with \( cnz \) show \( x \leq \inverse c \ast \Sup \ \{ c \ast x \mid x \in S \} \)
    by (simp add: mult. assoc [symmetric])
  qed
with \( nnc \) have \( c \ast \Sup S \leq c \ast (\inverse c \ast \Sup \ \{ c \ast x \mid x \in S \}) \)
  by (auto intro: mult-left-mono)
with \( cnz \) show \( c \ast \Sup S \leq \Sup \ \{ c \ast x \mid x \in S \} \)
  by (simp add: mult. assoc [symmetric])
  qed

lemma closure-contains-Sup:
  fixes \( S :: \) real set
  assumes \( neS: S \neq \{} \) and \( bS: \forall x \in S. x \leq B \)
  shows \( \Sup S \in \closure S \)
proof 
  let \( ?T = \uminus ' S \)
  from \( neS \) have \( neT: ?T \neq \{} \) by (auto)
  from \( bS \) have \( bT: \forall x \in ?T. -B \leq x \) by (auto)
  hence \( bbT: \ bdd-below \ ?T \) by (intro bdd-belowI, blast)
  have \( \Sup S = - \Inf ?T \)
  proof (rule antisym)
    from \( neT \) \( bbT \) have \( \bigwedge x. x \in S \Rightarrow \Inf (\uminus ' S) \leq -x \)
      by (blast intro: cInf-lower)
    hence \( \bigwedge x. x \in S \Rightarrow -1 \ast -x \leq -1 \ast \Inf (\uminus ' S) \)
      by (rule mult-left-mono-neg, auto)
    hence \( \lenInf: \bigwedge x. x \in S \Rightarrow x \leq - \Inf (\uminus ' S) \)
      by (simp)
    with \( neS \) \( bS \) show \( \Sup S \leq - \Inf ?T \)
      by (blast intro: cSup-least)
    have \( - \Sup S \leq \Inf ?T \)
proof (rule cInf-greatest [OF neT])
fix \( x \) assume \( x \in \uminus \cdot S \)
then obtain \( y \) where \( \text{yin: } y \in S \) and \( \text{rwx: } x = -y \)
by (auto)
from \( \text{yin bS have } y \leq \text{Sup } S \)
by (intro cSup-upper bdd-belowI, auto)
hence \(-1 * \text{Sup } S \leq -1 * y \)
by (simp add: mult-left-mono-neg)
with \( \text{rwx show } -\text{Sup } S \leq x \)
by (simp)
qed

hence \(-1 * \text{Inf } T \leq -1 * (- \text{Sup } S) \)
by (simp add: mult-left-mono-neg)
thus \(-\text{Inf } T \leq \text{Sup } S \)
by (simp)
qed

also {
from neT bbT have \( \text{Inf } T \in \text{closure } T \)
by (rule closure-contains-Inf)
hence \(-\text{Inf } T \in \uminus \cdot \text{closure } T \)
by (auto)
}
also {
have linear \( \uminus \cdot (\text{Inf }) \cdot T \)
by (auto intro: linearI)
hence \( \uminus \cdot \text{closure } T \subseteq \text{closure } (\uminus \cdot T) \)
by (rule closure-linear-image-subset)
}
also {
have \( \uminus \cdot T \subseteq S \)
by (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 tendsto-min:
fixes \( x \) \( y :: \text{real} \)
assumes ta: \( a \longrightarrow x \)
and tb: \( b \longrightarrow y \)
shows \( (\lambda i. \text{min } (a \cdot i) (b \cdot i)) \longrightarrow \text{min } x \cdot y \)
proof (rule LIMSEQ-I, simp)
fix \( e :: \text{real} \) assume pe: \( 0 < e \)
from ta pe obtain noa where balla: \( \forall n \geq \text{noa}. \text{abs } (a \cdot n - x) < e \)
by (auto dest: LIMSEQ-D)
from tb pe obtain nob where ballb: \( \forall n \geq \text{nob}. \text{abs } (b \cdot n - y) < e \)
by (auto dest: LIMSEQ-D)

fix \( n \)
assume ge: \( \text{max } \text{noa } \text{nob} \leq n \)
hence gea: \( \text{noa} \leq n \) and geb: \( \text{nob} \leq n \)
by (auto)
have \( \text{abs } (\text{min } (a \cdot n) (b \cdot n) - \text{min } x \cdot y) < e \)
proof cases
assume le: \( \text{min } (a \cdot n) (b \cdot n) \leq \text{min } x \cdot y \)
show \( ?\)thesis
proof cases
  assume \( a\ n \leq b\ n \)
  hence rwmin: \( \min (a\ n) (b\ n) = a\ n \) by(auto)
  with le have \( a\ n \leq \min x\ y \) by(simp)
  moreover from gea balla have \( \text{abs} (a\ n - x) < e \) by(simp)
  moreover have \( \min x\ y \leq x \) by(auto)
  ultimately have \( \text{abs} (a\ n - \min x\ y) < e \) by(auto)
  with rwmin show \( \text{abs} (\min (a\ n) (b\ n) - \min x\ y) < e \) by(simp)
next
  assume \( \neg a\ n \leq b\ n \)
  hence \( b\ n \leq a\ n \) by(auto)
  with le have \( b\ n \leq \min x\ y \) by(simp)
  moreover from geb ballb have \( \text{abs} (b\ n - y) < e \) by(simp)
  moreover have \( \min x\ y \leq y \) by(auto)
  ultimately have \( \text{abs} (b\ n - \min x\ y) < e \) by(auto)
  with rwmin show \( \text{abs} (\min (a\ n) (b\ n) - \min x\ y) < e \) by(simp)
qed
next
  assume \( \neg \min (a\ n) (b\ n) \leq \min x\ y \)
  hence le: \( \min x\ y \leq \min (a\ n) (b\ n) \) by(auto)
  show \( ?\)thesis
proof cases
  assume \( x \leq y \)
  hence \( \text{rwmin} : \min x\ y = x \) by(auto)
  with le have \( x \leq \min (a\ n) (b\ n) \) by(simp)
  moreover from gea balla have \( \text{abs} (a\ n - x) < e \) by(simp)
  moreover have \( \min (a\ n) (b\ n) \leq a\ n \) by(auto)
  ultimately have \( \text{abs} (\min (a\ n) (b\ n) - x) < e \) by(auto)
  with rwmin show \( \text{abs} (\min (a\ n) (b\ n) - \min x\ y) < e \) by(simp)
next
  assume \( \neg x \leq y \)
  hence \( y \leq x \) by(auto)
  hence \( \text{rwmin} : \min x\ y = y \) by(auto)
  with le have \( y \leq \min (a\ n) (b\ n) \) by(simp)
  moreover from geb ballb have \( \text{abs} (b\ n - y) < e \) by(simp)
  moreover have \( \min (a\ n) (b\ n) \leq b\ n \) by(auto)
  ultimately have \( \text{abs} (\min (a\ n) (b\ n) - y) < e \) by(auto)
  with rwmin show \( \text{abs} (\min (a\ n) (b\ n) - \min x\ y) < e \) by(simp)
qed
qed

thus \( \exists\ no. \forall n \geq\ no. |\min (a\ n) (b\ n) - \min x\ y| < e \) by(blast)
qed

definition supp :: \( 's \Rightarrow \text{real} \Rightarrow 's\ set \)
where \( \text{supp } f = \{ x. f\ x \neq 0 \} \)
4.13. MISCELLANEOUS MATHEMATICS

**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:
  \[ \begin{array}{c}
  0 \leq a; 0 \leq b
  \end{array} \] \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 \( \ominus \) c) \( \ominus \) 0 = 0
    by (simp add: max.absorb2)
  also {
    from nna le have a \( \ast \) b \leq a \( \ast \) c
      by (blast intro: mult-left-mono)
    hence 0 = max (a \( \ast \) b \( \ominus \) a \( \ast \) c) 0
      by (simp add: max.absorb1 field-simps)
  } finally show ?thesis by (simp add: tminus-def)
next
  case False
  hence le \( \ominus \) c \leq b by (auto)
  hence a \( \ast \) max (b \( \ominus \) c) \( \ominus \) 0 = a \( \ast \) (b \( \ominus \) 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 a \( \ast \) (b \( \ominus \) c) = max (a \( \ast \) b \( \ominus \) a \( \ast \) c)
      0
      by (simp add: max.absorb1)
  } finally show ?thesis by (simp add: tminus-def)
qed

lemma tminus-le[simp]:
  \( b \leq a \implies a \ominus b = a \ominus b \)
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 \( 0 \leq a \ominus c \))
case True
  note pac = this
show \texttt{?thesis}
proof (cases \(0 \leq b - d\))
\begin{itemize}
  \item case \texttt{True} note \texttt{pbd = this}
    from \texttt{pac and pbd} have \((c + d) \leq (a + b)\) by (simp)
  \item with \texttt{pac and pbd} show \texttt{?thesis} by (simp)
\end{itemize}
next
\begin{itemize}
  \item case \texttt{False} with \texttt{pac} show \texttt{?thesis}
    by (cases \(c + d \leq a + b, auto\))
\end{itemize}
qed

next
\begin{itemize}
  \item case \texttt{False} note \texttt{nac = this}
  \item show \texttt{?thesis}
  \begin{itemize}
    \item proof (cases \(0 \leq b - d\))
      \begin{itemize}
        \item case \texttt{True} with \texttt{nac} show \texttt{?thesis}
          by (cases \(c + d \leq a + b, auto\))
      \end{itemize}
    \end{itemize}
  \end{itemize}
  \begin{itemize}
    \item with \texttt{nac} and \texttt{nbd} show \texttt{?thesis}
    by (simp)
  \end{itemize}
  qed
 \end{itemize}


\begin{lemma}
\texttt{tminus-sum-mono}:
\end{lemma}
\begin{proof}
assumes \texttt{fS: finite S}
shows \texttt{sum f S \oplus sum g S} \leq \texttt{sum (\lambda x. f x \oplus g x) S}
(is \texttt{?X S})
\end{proof}

\begin{proof}
(rule \texttt{finite-induct})
from \texttt{fS} show \texttt{finite S}.

show \texttt{?X \{\}} by (simp)
\end{proof}

fix \texttt{x and F}
assume \texttt{fF: finite F} and \texttt{xniF: x \notin F}
and \texttt{IH: \?X F}
have \texttt{f x + sum f F \oplus g x + sum g F} \leq
\texttt{(f x \oplus g x) + (sum f F \oplus sum g F)}
by (rule \texttt{tminus-add-mono})
also from \texttt{IH} have \texttt{...} \leq \texttt{(f x \oplus g x) + (\sum x \in F. f x \oplus g x)}
by (rule \texttt{add-left-mono})
finally show \texttt{?X (insert x F)}
by (simp add: sum.insert[OF \texttt{fF xniF}])
qed

\begin{lemma}
\texttt{tminus-nneg}\texttt{[simp, intro]}:
\end{lemma}
\begin{proof}
\(0 \leq a \oplus b\)
by (cases \(b \leq a, auto\))
\end{proof}

\begin{lemma}
\texttt{tminus-right-antimono}:
\end{lemma}
\begin{proof}
assumes \texttt{clb: c \leq b}
\end{proof}
shows $a \ominus b \leq a \ominus c$

proof (cases $b \leq a$)
  case True
    moreover with $clb$ have $c \leq a$ by (auto)
  moreover note $clb$
  ultimately show $?thesis$ by (simp)
next
  case False then show $?thesis$ by (simp)
qed

lemma min-tminus-distrib:
  $\min a b \ominus c = \min (a \ominus c) (b \ominus c)$
  unfolding tminus-def by (auto)
end
Bibliography


