pGCL for Isabelle

David Cock

February 23, 2021
Contents

1 Overview 1

2 Introduction to pGCL 3
   2.1 Language Primitives 3
      2.1.1 The Basics 3
      2.1.2 Assertion and Annotation 4
      2.1.3 Probability 4
      2.1.4 Nondeterminism 5
      2.1.5 Properties of Expectations 5
   2.2 Loops 6
      2.2.1 Guaranteed Termination 6
      2.2.2 Probabilistic Termination 8
   2.3 The Monty Hall Problem 9
      2.3.1 The State Space 9
      2.3.2 The Game 10
      2.3.3 A Brute Force Solution 11
      2.3.4 A Modular Approach 11

3 Semantic Structures 17
   3.1 Expectations 17
      3.1.1 Bounded Functions 18
      3.1.2 Non-Negative Functions 20
      3.1.3 Sound Expectations 21
      3.1.4 Unitary expectations 25
      3.1.5 Standard Expectations 25
      3.1.6 Entailment 27
      3.1.7 Expectation Conjunction 28
      3.1.8 Rules Involving Conjunction 29
      3.1.9 Rules Involving Entailment and Conjunction Together 31
   3.2 Expectation Transformers 33
      3.2.1 Comparing Transformers 36
      3.2.2 Healthy Transformers 39
      3.2.3 Sublinearity 47
3.2.4 Determinism ........................................ 57
3.2.5 Modular Reasoning ................................ 60
3.2.6 Transforming Standard Expectations ............ 61
3.3 Induction ............................................. 62
3.3.1 The Lattice of Expectations ..................... 62
3.3.2 The Lattice of Transformers ..................... 66
3.3.3 Tail Recursion .................................... 72

4 The pGCL Language ..................................... 75
4.1 A Shallow Embedding of pGCL in HOL ............... 75
4.1.1 Core Primitives and Syntax ..................... 75
4.1.2 Unfolding rules for non-recursive primitives ...... 79
4.2 Healthiness .......................................... 81
4.2.1 The Healthiness of the Embedding .............. 81
4.2.2 Healthiness for Loops ........................... 94
4.3 Continuity ........................................... 104
4.3.1 Continuity of Primitives ....................... 105
4.3.2 Continuity of a Single Loop Step .............. 123
4.4 Continuity and Induction for Loops ................. 126
4.4.1 The Limit of Iterates .......................... 134
4.5 Sublinearity .......................................... 144
4.5.1 Nonrecursive Primitives ....................... 144
4.5.2 Sublinearity for Loops ......................... 150
4.6 Determinism ......................................... 154
4.6.1 Additivity ....................................... 154
4.6.2 Maximality ...................................... 155
4.6.3 Determinism .................................... 157
4.7 Well-Defined Programs. ............................. 158
4.7.1 Strict Implies Liberal .......................... 158
4.7.2 Sub-Distributivity of Conjunction ............. 163
4.7.3 The Well-Defined Predicate .................... 172
4.8 The Loop Rules ...................................... 175
4.8.1 Liberal and Strict Invariants .................. 175
4.8.2 Partial Correctness ............................. 176
4.8.3 Total Correctness ............................... 177
4.8.4 Unfolding ....................................... 177
4.9 The Algebra of pGCL ................................. 178
4.9.1 Program Refinement ............................. 178
4.9.2 Simple Identities ............................... 179
4.9.3 Deterministic Programs are Maximal ............ 184
4.9.4 The Algebraic Structure of Refinement .......... 186
4.9.5 Data Refinement ................................ 190
4.9.6 The Algebra of Data Refinement ............... 193
4.9.7 Structural Rules for Correspondence .......... 200
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.9.8</td>
<td>Structural Rules for Data Refinement</td>
<td>202</td>
</tr>
<tr>
<td>4.10</td>
<td>Structured Reasoning</td>
<td>204</td>
</tr>
<tr>
<td>4.10.1</td>
<td>Syntactic Decomposition</td>
<td>204</td>
</tr>
<tr>
<td>4.10.2</td>
<td>Algebraic Decomposition</td>
<td>209</td>
</tr>
<tr>
<td>4.10.3</td>
<td>Hoare triples</td>
<td>209</td>
</tr>
<tr>
<td>4.11</td>
<td>Loop Termination</td>
<td>209</td>
</tr>
<tr>
<td>4.11.1</td>
<td>Trivial Termination</td>
<td>210</td>
</tr>
<tr>
<td>4.11.2</td>
<td>Classical Termination</td>
<td>210</td>
</tr>
<tr>
<td>4.11.3</td>
<td>Probabilistic Termination</td>
<td>213</td>
</tr>
<tr>
<td>4.12</td>
<td>Automated Reasoning</td>
<td>217</td>
</tr>
<tr>
<td>4.12.1</td>
<td>Miscellaneous Mathematics</td>
<td>219</td>
</tr>
<tr>
<td>4.13.1</td>
<td>Truncated Subtraction</td>
<td>225</td>
</tr>
</tbody>
</table>

### Additional Material

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

```plaintext
datatype coin = Heads | Tails
record coins =
  a :: coin
  b :: coin
```

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

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

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

2.1.2 Assertion and Annotation

Assertions in pGCL are real-valued functions of the state, which are often 
interpreted as a probability distribution over possible outcomes. These func-
tions 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 \( «P» \supseteq «Q» \), \( [\llbracket ?P \rrbracket \supseteq \llbracket ?Q \rrbracket ; ?P ?s] \rightarrow ?Q \). Thus a valid specification of \( \text{Apply} (\lambda s. s(a := b s)) \) is:

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

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:

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

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

**definition**

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

**where**

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

**lemma**

\[ (\lambda s. 1/2) = \wp (\text{flip-a } (1/2)) \subseteq (\lambda s. a = \text{Heads}) \]

**unfolding** flip-a-def

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

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

### 2.1.4 Nondeterminism

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

**lemma**

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

**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**

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

**where**

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

**lemma**

\[ (\lambda s. 1/4) = \wp (\text{flip-a } (1/2) \sqcup \text{flip-b } (1/2)) \subseteq (\lambda s. a = \text{Heads} \land b = \text{Heads}) \]

**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 \texttt{Puniform} :: 'a set \Rightarrow ('a \Rightarrow \texttt{real})
where \texttt{Puniform} S = (\lambda x. \text{if } x \in S \text{ then } 1 / \text{card } S \text{ else } 0)

lemma \texttt{Puniform-in}:
  \( x \in S \Rightarrow \texttt{Puniform} S x = 1 / \text{card } S \)
  by (simp add: \texttt{Puniform-def})

lemma \texttt{Puniform-out}:
  \( x \notin S \Rightarrow \texttt{Puniform} S x = 0 \)
  by (simp add: \texttt{Puniform-def})

lemma \texttt{supp-Puniform}:
  finite S \Rightarrow \texttt{supp} (\texttt{Puniform} S) = S
  by (auto simp: \texttt{Puniform-def} \texttt{supp-def})

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

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

The expectations of independent variables add:

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

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

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

**Definition**

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

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

**Lemma** `wp-inv-count`:

```plaintext
wp-inv (λx. 0 < x) (Apply (λs. s - 1)) «inv-count»
unfolding wp-inv-def inv-count-def wp-eval o-def
proof(clarify, cases)
fix x::int
assume 0 ≤ x
then show (λx. 0 < x) x * (λx. 0 ≤ x) x ≤ (λx. 0 ≤ x) (x - 1)
  by (simp add:embed-bool-def)
next
fix x::int
assume ¬0 ≤ x
then show (λx. 0 < x) x * (λx. 0 ≤ x) x ≤ (λx. 0 ≤ x) (x - 1)
  by (simp add:embed-bool-def)
qed
```

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

**Lemma** `term-countdown`:

```plaintext
«inv-count» ⊢ ⊢ wp countdown (λs. 1)
unfolding countdown-def
proof(intro loop-term-nat-measure[where \(m=λx. nat (max x 0)\)] wp-inv-count)
let ?p = Apply (λs. x - 1::int)
```

As usual, well-definedness is trivial.

**Show** `well-def ?p`

```plaintext
by(rule wd-intros)
```

A measure of 0 implies termination.

**Show** \(∀x. nat (max x 0) = 0 → ¬0 < x\)

```plaintext
by(auto)
```

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

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


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

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

qed

\subsection{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 (\lambda -. Heads) (\lambda s. 1/2) \oplus Apply (\lambda -. 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 ((\neq) Heads) \rightarrow flip od

Nonetheless, we can show termination.

lemma wait-for-heads-term:
\[ \lambda s. 1 \vdash \text{wp} \ \text{wait-for-heads} \ (\lambda s. 1) \]

unfolding wait-for-heads-def

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

proof (rule termination-0-1)

show well-def flip
unfolding flip-def
by (auto intro: wd-intros)

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

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

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

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

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

show $(\theta :: \text{real}) < \frac{1}{2} \text{ by (simp)}$
qed
end

2.3 The Monty Hall Problem

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

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

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

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

2.3.1 The State Space

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

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

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

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

We prove explicitly that only valid doors are ever chosen.

\textbf{definition} \textit{inv-prize} :: \textit{game} \rightarrow \textit{bool}  \\
\textit{where} \textit{inv-prize} \textit{g} ≡ \textit{prize} \textit{g} \in \{1,2,3\}

\textbf{definition} \textit{inv-clue} :: \textit{game} \rightarrow \textit{bool}  \\
\textit{where} \textit{inv-clue} \textit{g} ≡ \textit{clue} \textit{g} \in \{1,2,3\}

\textbf{definition} \textit{inv-guess} :: \textit{game} \rightarrow \textit{bool}  \\
\textit{where} \textit{inv-guess} \textit{g} ≡ \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 prog} \\
\textit{where} \textit{hide-behind} \textit{D} ≡ \textit{Apply} (\textit{prize-update} (\lambda \textit{x}. \textit{D}))

Choose door \(D\).

\textbf{definition} \textit{guess-behind} :: \textit{nat} \rightarrow \textit{game prog}  \\
\textit{where} \textit{guess-behind} \textit{D} ≡ \textit{Apply} (\textit{guess-update} (\lambda \textit{x}. \textit{D}))

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

\textbf{definition} \textit{open-door} :: \textit{nat} \rightarrow \textit{game prog}  \\
\textit{where} \textit{open-door} \textit{D} ≡ \textit{Apply} (\textit{clue-update} (\lambda \textit{x}. \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 prog}  \\
\textit{where} \textit{hide-prize} ≡ \textit{hide-behind} 1 \sqcap \textit{hide-behind} 2 \sqcap \textit{hide-behind} 3

Guess uniformly at random.

\textbf{definition} \textit{make-guess} :: \textit{game prog}  \\
\textit{where} \textit{make-guess} ≡ \textit{guess-behind} 1 (\lambda \textit{s}. \frac{1}{3}) \oplus \textit{guess-behind} 2 (\lambda \textit{s}. \frac{1}{2}) \oplus \textit{guess-behind} 3

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

\textbf{definition} \textit{reveal} :: \textit{game prog}  \\
\textit{where} \textit{reveal} ≡ \prod \textit{d} \in (\lambda \textit{s}. \{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 prog}  \\
\textit{where} \textit{switch-guess} ≡ \prod \textit{d} \in (\lambda \textit{s}. \{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 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]:

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

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

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

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

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

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

**lemma wp-monty-noswitch**:\[
(\lambda s. \frac{1}{3}) = \text{wp} (\text{monty False}) \ \neg \text{player-wins}
\]

**unfolding** monty-def hide-prize-def make-guess-def reveal-def

\[
\text{hide-behind-def guess-behind-def open-door-def switch-guess-def}
\]

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

**lemma swap-upd**:\[
s[prize := p, clue := c, guess := g ] =
\]

\[
s[prize := p, guess := g, clue := c ]
\]

**by** (simp)

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

**lemma wp-monty-switch-brute-force**:\[
(\lambda s. \frac{2}{3}) = \text{wp} (\text{monty True}) \ \neg \text{player-wins}
\]

**unfolding** monty-def hide-prize-def make-guess-def reveal-def

\[
\text{hide-behind-def guess-behind-def open-door-def switch-guess-def}
\]

— Note that this is getting slow

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

2.3.4 A Modular Approach

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

**Healthiness**

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

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

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

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

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

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

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

**Annotations**

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

hide-prize chooses a valid door.

**Lemma wp-hide-prize:**
\[(\lambda s. 1) \vdash \text{wp hide-prize «inv-prize»}\]

**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:**
\[(\lambda s. 2/3 \times (\lambda g. \text{inv-prize } g) s) \vdash \text{wp make-guess «\lambda g. guess } g \neq \text{prize } g \wedge \text{inv-prize } g \wedge \text{inv-guess } g»}\]

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

**Lemma last-one:**
assumes \(a \neq b\) and \(a \in \{1::\text{nat}, 2, 3\}\) 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:**
\[«\lambda g. \text{guess } g \neq \text{prize } g \wedge \text{inv-prize } g \wedge \text{inv-guess } g» \vdash \text{wp reveal «\lambda g. guess } g \neq \text{prize } g \wedge \text{clue } g \neq \text{prize } g \wedge \text{clue } g \neq \text{guess } g \wedge \text{inv-prize } g \wedge \text{inv-guess } g \wedge \text{inv-clue } g»}\]

(is ?X \vdash \text{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: \(\{\text{Suc 0, 2, 3}\} - \{\text{prize } s, \text{guess } s\} = \{c\}\)
and c \neq prize s
and c \neq guess s
and c \in \{\text{Suc 0, 2, 3}\}

**Unfolding** inv-prize-def inv-guess-def
by (force dest:last-one elim!:ex1E)

ultimately show \(1 \leq \text{wp reveal ?Y } s\)
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.
CHAPTER 2. INTRODUCTION TO PGCL

lemma distinct-game:
  \[\begin{array}{l}
  \text{guess } g \neq \text{prize } g; \quad \text{clue } g \neq \text{prize } g; \quad \text{clue } g \neq \text{guess } g; \\
  \quad \text{inv-prize } g; \quad \text{inv-guess } g; \quad \text{inv-clue } g \quad \Rightarrow \\
  \{1, 2, 3\} = \{\text{guess } g, \text{prize } g, \text{clue } g\}
\end{array}\]

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:
«\(\lambda g. \text{guess } g \neq \text{prize } g \land \text{clue } g \neq \text{prize } g \land \text{clue } g \neq \text{guess } g \land \text{inv-prize } g \land \text{inv-guess } g \land \text{inv-clue } g\)» ⊢ wp switch-guess «player-wins»

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

fix s
assume guess s ≠ prize s and clue s ≠ prize s
and clue s ≠ guess s and inv-prize s
and inv-guess s and inv-clue s
note state = this
hence 1 ≤ Inf ((\(\lambda a. \text{«player-wins»} (s[\text{guess} := a])\)) ·
  \{\text{guess } s, \text{prize } s, \text{clue } s\} − \{\text{clue } s, \text{guess } s\})
by(auto simp:insert-Diff-if player-wins-def)
also have «Inf «\(\lambda a. \text{«player-wins»} (s[\text{guess} := a])\)» ·
  \{1, 2, 3\} − \{\text{clue } s, \text{guess } s\})»
by(simp add:distinct-game[symmetric])
also have ... = wp switch-guess «player-wins» s
by(simp add:switch-guess-def guess-behind-def wp-eval o-def)
finally show 1 ≤ 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:
(\(\lambda s. 2/3\) ⊬ 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 {\(\lambda s. 2/3\) monty True} «player-wins» p
unfolding monty-def
by (simp add: wd-intros sound-intros monty-healthy)

qed

Using the VCG

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

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

lemma wp-monty-switch-vcg:
(\alpha. 2/3) ⊢ wp (monty True) «player-wins»
unfolding monty-def
by (simp, pvcg)

end
Chapter 3

Semantic Structures

3.1 Expectations

theory Expectations imports Misc begin type-synonym 's expect = 's ⇒ real

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

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$a \rightarrow b$</th>
<th>$x$</th>
<th>$y$</th>
<th>$x \leq y$</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 :: $\text{real} \Rightarrow (\text{'a} \Rightarrow \text{real}) \Rightarrow \text{bool}$

**where**

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

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

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

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

by (simp add:bounded-by-def)

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

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

by (blast dest:le-funD)

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

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

by (simp add:bounded-by-def)

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

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

by (blast intro:le-funI)

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

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

**where**

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

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

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

**where**

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

**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 bound-of $P \leq b$
unfolding bound-of-def
using $bP$ by(intro cSup-least, auto)

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

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

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

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

This is occasionally easier to apply:

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

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

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

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

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

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

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

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

lemma le-bound-of[intro]:
\( \forall x. \) bounded \( f \) \( \Rightarrow \) \( f x \leq \) 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
\( \text{nneg} :: (a \Rightarrow 'b::{zero,order}) \Rightarrow \text{bool} \)
where
\( \text{nneg} P \leftarrow\leftarrow (\forall x. 0 \leq P x) \)

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

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

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

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

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

**lemma** nneg-const[[iff]]:
nneg (λx. c) ↔ 0 ≤ c
**by** (simp add:nneg-def)

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

**lemma** nneg-bound-nneg[intro]:
[ bounded P; nneg P ] → 0 ≤ bound-of P
**by** (blast intro:order-trans)

**lemma** nneg-bounded-by-nneg[dest]:
[ bounded-by b P; nneg P ] → 0 ≤ (b::real)
**by** (blast intro:order-trans)

**lemma** bounded-by-nneg[dest]:
**fixes** P :: 's ⇒ real
**shows** [ bounded-by b P; nneg P ] → 0 ≤ b
**by** (blast intro:order-trans)

3.1.3 Sound Expectations

**definition** sound :: ('s ⇒ real) ⇒ bool
**where** sound P ≡ bounded P ∧ nneg P

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

**lemma** soundI:
[ bounded P; nneg P ] → sound P
**by** (simp add:sound-def)

**lemma** soundI2[intro]:
[ bounded-by b P; nneg P ] → sound P
**by**(blast intro:soundI)

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

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

**lemma** bound-of-sound[intro]:
**assumes** sP: sound P
shows $0 \leq \text{bound-of } P$

using \textit{assms by(auto)}

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

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

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

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

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

\textbf{lemma} \textit{div-sound}:
\textit{assumes} $sP$: sound $P$ and cpos: $0 < c$
\textit{shows} sound $(\lambda s. P s \mathbin{/} c)$
\textit{proof}
\textit{from} $sP$ and cpos have $\forall s. P s \mathbin{/} c \leq \text{bound-of } P \mathbin{/} c$
\textit{by(blast intro:divide-right-mono less-imp-le)}
\textit{thus} bounded-by $(\text{bound-of } P \mathbin{/} c)$ $(\lambda s. P s \mathbin{/} c)$ \textit{by(blast)}
\textit{from} assms show \textit{nneg } $(\lambda s. P s \mathbin{/} c)$
\textit{by(blast intro:divide-nonneg-pos)}
\textit{qed}

\textbf{lemma} \textit{tminus-sound}:
\textit{assumes} $sP$: sound $P$ and nnc: $0 \leq c$
3.1. EXPECTATIONS

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

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

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

unfolding o-def by (blast)

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

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

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

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

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

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

lemma sum-sound:
assumes sP: \forall x \in S. sound (P x)
shows sound (\lambda s. \sum x \in S. P x s)
proof (rule soundI2)
from sP show bounded-by (\sum x \in S. bound-of (P x)) (\lambda s. \sum x \in S. P x s)
  by (auto intro!:sum-mono)
from sP show 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:
«P» s * «P» s = «P» s
by (simp add:embed-bool-def)

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

lemma eval-embed-false[simp]:
¬P s ⇒ «P» s = 0
by (simp add:embed-bool-def)

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

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

lemma embed-le-1-alt[simp,intro]:
0 ≤ 1 − «G» s
by(subst add-le-cancel-right[where e=«G» s, symmetric], simp)

lemma expect-1-I:
P x ⇒ 1 ≤ «P» x
by(simp)

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

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

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

definition negate :: ('s ⇒ bool) ⇒ 's ⇒ bool (N)
where negate P = (λs. ¬ P s)

lemma negateI:
¬ P s ⇒ N P s
by (simp add:negate-def)

lemma embed-split:
f s = «P» s * f s + «N P» s * f s
by (simp add:negate-def embed-bool-def)
3.1. EXPECTATIONS

**lemma** negate-embed:

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

by \((\text{simp add:embed-bool-def negate-def})\)

**lemma** eval-nembed-true[\(simp\)]:

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

by \((\text{simp add:embed-bool-def negate-def})\)

**lemma** eval-nembed-false[\(simp\)]:

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

by \((\text{simp add:embed-bool-def negate-def})\)

**lemma** negate-Not[\(simp\)]:

\[ N \text{ Not} = (\lambda x. x) \]

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

**lemma** negate-negate[\(simp\)]:

\[ N (N P) = P \]

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

**lemma** embed-bool-cancel:

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

by \((\text{cases G s, simp-all})\)

3.1.6 Entailment

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

**abbreviation** entails :: \((s \Rightarrow \text{real}) \Rightarrow (s \Rightarrow \text{real}) \Rightarrow \text{bool}\)

where \(P \vdash Q \equiv P \leq Q\)

**lemma** entailsI[\(intro\)]:

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

by \((\text{simp add:le-funI})\)

**lemma** entailsD[\(dest\)]:

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

by \((\text{simp add:le-funD})\)

**lemma** eq-entails[\(intro\)]:

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

by \((\text{blast})\)

**lemma** entails-trans[\(trans\)]:

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

by \((\text{blast intro:order-trans})\)

For standard expectations, both notions of entailment coincide. This result justifies the above claim that our definition generalises predicate entailment:
lemma implies-entails:
\[ \forall s. \; P \implies Q \quad \implies \langle P \rangle \vdash \langle Q \rangle \]
by (rule entailsI, case-tac P s, simp-all)

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

3.1.7 Expectation Conjunction

declaration
\[ \text{pconj} :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real} \quad \quad \text{infixl} \& 71 \]
where
\[ p \& q \equiv p + q - 1 \]

declaration
\[ \text{exp-conj} :: (\forall s. \text{real} \Rightarrow \text{real}) \Rightarrow (\forall s. \text{real} \Rightarrow \text{real}) \Rightarrow (\forall s. \text{real}) \quad \text{infixl} \& 71 \]
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 pconj-lzero[intro,simp]:
\[ b \leq 1 \implies 0 \& b = 0 \]
by (simp add: pconj-def tminus-def)

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

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

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

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

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

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

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

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

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

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

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

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

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

3.1.8 Rules Involving Conjunction.

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

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

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

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

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

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

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

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

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

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

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

show \( \text{nneg } (\lambda s. P s \& Q s) \)
unfolding \( \text{pconj-def tminus-def} \) by\((\text{force})\)
qed

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

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

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

3.1.9 Rules Involving Entailment and Conjunction Together

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

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

Thus $(P \&\& Q) \leq (R \&\& S)$

unfolding exp-conj-def pconj-def.

Qed.

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

Lemma pentails-cases:
assumes $PQe: \forall x. P x \vdash Q x$
and exhaust: $\forall s. \exists x. P (x s) s = 1$
and framed: $\forall x. P x \&\& R \vdash Q x \&\& S$
and $sR$: sound $R$ and $sS$: sound $S$
and $bQ$: $\forall x$. bounded-by 1 ($Q x$)

shows $R \vdash S$

Proof (rule le-funI)

fix $s$

from exhaust obtain $x$ where $P-x$s: $P x s = 1$ by (blast)
moreover {
    hence $1 = P x s$ by (simp)
    also from $PQe$ have $P x s \leq Q x s$ by (blast dest: le-funD)
    finally have $Q x s = 1$
        using $bQ$ by (blast intro: antisym)
}

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

Qed.

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

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

Lemma unitary-embed[iff]:
unitary $(P)$
by (auto)

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

Lemma unitary-mult:
3.2. EXPECTATION TRANSFORMERS

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

lemma exp-conj-unitary:
  $[\text{unitary } P; \text{unitary } Q ] \implies \text{unitary } (P \&\& Q)$
  by (intro unitaryI2 nnegI2, auto)

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

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

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

3.2 Expectation Transformers

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

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

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

\[
P_{\text{prior}}(a) = 0.7 \times P_{\text{post}}(b) + 0.3 \times P_{\text{post}}(c),
\]

but this need not be the case.

Consider the automaton of Figure 3.2. Here, we have extended that of Figure 3.1 with two additional states, d and e, and a pair of silent (unlabelled) transitions. From the initial state, e, this automaton is free to transition either to the original starting state (a), and thence behave exactly as the previous automaton did, or to d, which has the same set of available transitions, now with different probabilities. Where previously we could state that the automaton would terminate in state b with probability 0.7 (and in c with probability 0.3), this now depends on the outcome of the nondeterministic transition from e to either a or d. The most we can now say is that we must reach b with probability at least 0.5 (the minimum from either a or d) and c with at least probability 0.3. Note that these probabilities do not sum to one (although the sum will still always be less than one). The associated expectation transformer is now sub-linear: 

\[
P_{\text{prior}}(e) = 0.5 \times P_{\text{post}}(b) + 0.3 \times P_{\text{post}}(c).
\]

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

arguments in Section 4.11.

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

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

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

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

**Definition**
le-trans :: 's trans ⇒ 's trans ⇒ bool

where
le-trans t u ≡ ∀ P. sound P → t P ≤ u P

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

**Definition**
le-utrans :: 's trans ⇒ 's trans ⇒ bool

where
le-utrans t u ≡→ (∀ P. unitary P → t P ≤ u P)

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

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

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

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

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

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

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

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

**Lemma** le-trans-le-utrans[dest]:
le-trans t u \rightarrow le-utrans t u

**Unfolding** le-trans-def le-utrans-def **by**(auto)

**Definition**
l-trans :: 's trans \Rightarrow 's trans \Rightarrow bool

**Where**
l-trans t u \longleftrightarrow le-trans t u \wedge \neg le-trans u t

Transformer equivalence is induced by comparison:

**Definition**
equiv-trans :: 's trans \Rightarrow 's trans \Rightarrow bool

**Where**
equiv-trans t u \longleftrightarrow le-trans t u \wedge le-trans u t

**Definition**
equiv-utrans :: 's trans \Rightarrow 's trans \Rightarrow bool

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

**Lemma** equiv-transI[intro]:
[ \forall P. sound P \Rightarrow t P = u P ] \Rightarrow equiv-trans t u

**Unfolding** equiv-trans-def **by**(force)

**Lemma** equiv-utransI[intro]:
[ \forall P. sound P \Rightarrow t P = u P ] \Rightarrow equiv-utrans t u

**Unfolding** equiv-utrans-def **by**(force)

**Lemma** equiv-transD[dest]:
[ equiv-trans t u; sound P ] \Rightarrow t P = u P

**Unfolding** equiv-trans-def **by**(blast intro:antisym)

**Lemma** equiv-utransD[dest]:
[ equiv-utrans t u; unitary P ] \Rightarrow t P = u P

**Unfolding** equiv-utrans-def **by**(blast intro:antisym)

**Lemma** equiv-trans-refl[iff]:
equiv-trans t t

**By**(blast)

**Lemma** equiv-utrans-refl[iff]:
equiv-utrans t t

**By**(blast)

**Lemma** le-trans-antisym:
[ le-trans x y; le-trans y x ] \Rightarrow equiv-trans x y

**Unfolding** equiv-trans-def **by**(simp)

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

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

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

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

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

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

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

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

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

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

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

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

\textbf{lemma equiv-uimp-le-alt}:
\[ \text{equiv-utrans } t \ u \ \Rightarrow \text{le-utrans } u \ t \]

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

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

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

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

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

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

\textbf{by (simp)}

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

\textbf{by (simp)}

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

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

\textbf{lemma le-utrans-equiv-rsp-right[simp]}:
\[ \text{equiv-utrans } t \ u \ \Rightarrow \text{le-utrans } v \ t \ \leftrightarrow \text{le-utrans } v \ u \]
3.2. EXPECTATION TRANSFORMERS

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

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

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

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

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

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

3.2.2 Healthy Transformers

Feasibility

definition feasible :: (('a ⇒ real) ⇒ ('a ⇒ real)) ⇒ bool
where feasible t ⨸ (\forall P \ h. \ bounded-by \ h \ P \ ∧ \ nneg \ P \ →
\ bounded-by \ h \ (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]:**
\[
\phi \land \phi \Rightarrow \phi' \Rightarrow \phi'' \Rightarrow \phi'''
\]
by (force simp feasible-def)

**Lemma feasible-boundedD[dest]:**
\[
[\text{feasible } t; \text{bounded-by } b \ P; \text{nneg } P] \Rightarrow \text{bounded-by } b \ (t \ P)
\]
by (simp add feasible-def)

**Lemma feasible-nnegD[dest]:**
\[
[\text{feasible } t; \text{bounded-by } b \ P; \text{nneg } P] \Rightarrow \text{nneg } (t \ P)
\]
by (simp add feasible-def)

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

**Lemma feasible-pr-0[simp]:**
\[
\text{fixes } t::('s \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real}
\]
assumes \(ft\) feasible \(t\)
shows \(t \ (\lambda s. 0) = (\lambda s. 0)\)
proof (rule ext, rule antisym)
fix \(s\)

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

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

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

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

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

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

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

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

Monotonicity

definition
  mono-trans :: (('s ⇒ real) ⇒ ('s ⇒ real)) ⇒ bool
where
  mono-trans t ≡ ∀ P Q. (sound P ∧ sound Q ∧ P ≤ Q) −→ t P ≤ 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 o 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]:
  [ [ ∀ P Q. [ sound P; sound Q; P ≤ Q ] −→ t P ≤ t Q ] −→ mono-trans t
by (simp add: mono-trans-def)

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

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

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

Healthiness

Healthy transformers are feasible and monotonic, and respect scaling

definition
healthy :: (('s ⇒ real) ⇒ ('s ⇒ real)) ⇒ bool

where
healthy t ↔ feasible t ∧ mono-trans t ∧ scaling t

lemma healthyI[intro]:
[ feasible t; mono-trans t; scaling t ] healthy t
by(simp add:healthy-def)
3.2. EXPECTATION TRANSFORMERS

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

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

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

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

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

lemma healthy-bounded-byD[intro]:
    [ 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, 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 \textit{healthy-fixes-bot} = \textit{feasible-fixes-bot}[OF \textit{healthy-feasibleD}]

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

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

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

\textbf{Lemma} \textit{healthy-pr-bot[\textit{simp}]}:
healthy \(t\) \(\Rightarrow\) \textit{t} (\(\lambda s\). 0) \(=\) (\(\lambda s\). 0)
\textbf{by}(\text{blast intro:feasible-pr-0})

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

\textbf{Lemma} \textit{healthy-equivI}:
\textit{fixes} \(t::'(s \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real}\) and \(u\)
\textbf{assumes} \textit{equiv: equiv-trans} \(t\) \(u\)
\textbf{shows} \textit{healthy} \(u\)
\textbf{proof}
\textbf{have} \textit{le-t-u}: \textit{le-trans} \(t\) \(u\) by(\text{blast intro:equiv})
\textbf{have} \textit{le-u-t}: \textit{le-trans} \(u\) \(t\) by(\text{simp add:equiv-imp-le ac-simps equiv})
\textbf{from} \textit{equiv} \textbf{have} eq-u-t: \textit{equiv-trans} \(u\) \(t\) by(\text{simp add:ac-simps})

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

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

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

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-eqI 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 mnP: nneg P and bP: bounded-by b P

from ht mnP bP have \A s. t P s ≤ b by (blast)
with nc have \( \forall s. \, c \cdot t \, P \, s \leq c \cdot b \) by (\text{blast intro:mult-left-mono})

also \{
  from \textsf{hnP and bP} have \( \theta \leq b \) by (auto)
  with bc have \( c \cdot b \leq 1 \cdot b \) by (\text{blast intro:mult-right-mono})
  hence \( c \cdot b \leq b \) by (simp)
\}

finally show \( \text{bounded-by} \, b \) (\( \lambda s. \, c \cdot t \, P \, s \)) by (\text{blast intro:mult-nonneg-nonneg})

thus \( \text{nneg} \) (\( \lambda s. \, c \cdot t \, P \, s \)) by (\text{blast intro:mult-lin})

qed

show mono-trans (\( \lambda P \, s. \, c \cdot t \, P \, s \))
proof
fix \( P :: \mathcal{S} \Rightarrow \mathtt{real} \) and \( Q \)
assume \( sP: \text{sound} \, P \) and \( sQ: \text{sound} \, Q \) and \( le: \, P \vdash \top \Rightarrow Q \)
with \( \text{ht} \) have \( \forall s. \, t \, P \, s \leq t \, Q \, s \) by (\text{auto intro:le-funD})
with nc have \( \forall s. \, c \cdot t \, P \, s \leq c \cdot t \, Q \, s \) by (\text{blast intro:mult-left-mono})
thus \( \lambda s. \, c \cdot t \, P \, s \vdash \top \Rightarrow \lambda s. \, c \cdot t \, Q \, s \) by (\text{blast})
qed
from \( \text{ht} \) show scaling (\( \lambda P \, s. \, c \cdot t \, P \, s \)) by (\text{auto simp:scalingD healthy-scalingD \text{ht}})
qed

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

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

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

\textbf{definition}
\begin{align*}
nearly-healthy :: \quad & (('s \Rightarrow \mathtt{real}) \Rightarrow ('s \Rightarrow \mathtt{real})) \Rightarrow \mathtt{bool}\\
\text{where} \\
& \text{nearly-healthy} \, t \iff \quad (\forall P. \, \text{unitary} \, P \Rightarrow \text{unitary} \, (t \, P)) \land \\
& \quad (\forall \, P, \, Q. \, \text{unitary} \, P \Rightarrow \text{unitary} \, Q \Rightarrow P \vdash Q \Rightarrow t \, P \vdash t \, Q)
\end{align*}

lemma nearly-healthy\[intro\]:
\begin{align*}
& [ \forall P, \, \text{unitary} \, P \Rightarrow \text{unitary} \, (t \, P); \\
& \forall P, \, Q, \, [ \text{unitary} \, P; \, \text{unitary} \, Q; \, P \vdash Q ] \Rightarrow t \, P \vdash t \, Q ] \Rightarrow \text{nearly-healthy} \, t\\
& \text{by (simp add:nearly-healthy-def)}
\end{align*}

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

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

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

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

\text{lemmas nearly-healthy-id}[\text{iff} ] =
\text{healthy-nearly-healthy}[\text{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 \textit{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 \( x \odot y \) represents \textit{truncated subtraction} i.e. \( \max(x - y) (0::'a) \) (see \textit{Section 4.13.1}).

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

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

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

\((\forall t_p \Rightarrow \text{real} \Rightarrow \text{real} \Rightarrow \text{bool})\)

**where**

\[
\text{sub-add } t \leftarrow (\forall P Q s. (\text{sound } P \land \text{sound } Q) \rightarrow t P s + t Q s \leq t (\lambda s'. P s' + Q s') s)
\]

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

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

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

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

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

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

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

Can be rewritten, using scaling, to:

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

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

\[
vR + vS \leq v(R + S)
\]

---

**lemma** sub-addI[intro]:

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

by(simp add:sub-add-def)

**lemma** sub-addI2:

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

by(auto)

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

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

by(simp add:sub-add-def)

**lemma** equiv-sub-add:

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

assumes eq: equiv-trans \(t\ \ u\)

and sa: sub-add \(t\)

shows sub-add \(u\)

proof

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

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 eq have \( t (\lambda s. P s + Q s) s = u (\lambda s. P s + Q s) s \)
by(simp add:equiv-transD)
\}
finally show \( u P s + u Q s \leq u (\lambda s. P s + Q s) s \).
qed

Sublinearity and feasibility imply sub-additivity.

lemma sublinear-subadd:
fixes \( t :: (\prime s \Rightarrow \text{real}) \Rightarrow \prime s \Rightarrow \text{real} \)
assumes slt: sublinear \( t \)
and ft: feasible \( t \)
shows sub-add \( t \)
proof
fix \( P :: \prime s \Rightarrow \text{real} \) and \( Q :: \prime s \Rightarrow \text{real} \) and \( s :: \prime s \)
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 \( \ldots = \ldots \odot 0 \) by(simp)
also from sP sQ
have \( \ldots \leq t (\lambda s. P s + Q s \odot 0) s \)
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 \( t \langle P \rangle s + t \langle \neg P \rangle s \leq 1 \)
proof
from sat have \( t \langle P \rangle s + t \langle \neg P \rangle s \leq t (\lambda s. \langle P \rangle s + \langle \neg P \rangle s) s \) by(auto)
also have \( \ldots = t (\lambda s. 1) s \) by(simp add:negate-embed)
also \{
from ht have bounded-by 1 \( t (\lambda s. 1) \) by(auto)
\}
finally show \?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
  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 \(t \ P x \ \sum y \in F. \ t \ P y \ \leq t \ (P x) \ \sum y \in F. \ P y \ x) \)
by(auto intro:add-left-mono)
also from sat sP
have ... \(\leq t \ (\lambda xa. \ P x \ xa \ + \ \sum y \in F. \ P y \ xa)\) s
by(auto intro!:sub-addD[OF sat] sum-sound)
finally
show \(t \ P x \ \sum y \in F. \ t \ P y \ \leq t \ (\lambda xa. \ P x \ xa \ + \ \sum y \in F. \ P y \ xa)\) .
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 \( (\sum y \in \{s. \ G s\}. \ P y \ \cdot \ t \ \lambda z. \ z = y \ \cdot \ s) + \)
  \( (\sum y \in \{s. \ \neg G s\}. \ P y \ \cdot \ t \ \lambda z. \ z = y \ \cdot \ s) \leq t \ P s \)
proof
  have \(\{s. \ G s\} \cap \{s. \ \neg G s\} = \{\}\) by(blast)
  hence \( (\sum y \in \{s. \ G s\}. \ P y \ \cdot \ t \ \lambda z. \ z = y \ \cdot \ s) + \)
  \( (\sum y \in \{s. \ \neg G s\}. \ P y \ \cdot \ t \ \lambda z. \ z = y \ \cdot \ s) = \)
  \( (\sum y \in \{s. \ G s\} \cup \{s. \ \neg G s\}. \ P y \ \cdot \ t \ \lambda z. \ z = y \ \cdot \ s) \)
by(auto intro: sum.union-disjoint[symmetric])
also {\}
  have \(\{s. \ G s\} \cup \{s. \ \neg G s\} = \text{UNIV}\) by(blast)
  hence \( (\sum y \in \{s. \ G s\} \cup \{s. \ \neg G s\}. \ P y \ \cdot \ t \ \lambda z. \ z = y \ \cdot \ s) = \)
  \( (\lambda x. \sum y \in \text{UNIV}. \ P y \ \cdot \ t \ (\lambda x. \ \lambda z. \ z = y \ x) \ x) \ s \)
  by(simp)
}
from $s \wedge P$. have $\forall y. 0 \leq P y$ by(auto)

with {healthy-scalingD[OF ht]}

have $(\lambda x. \sum y \in \text{UNIV}. P y * t (\lambda x. \langle \lambda z. z = y \rangle x) x) s = (\lambda x. \sum y \in \text{UNIV}. t (\lambda x. P y * \langle \lambda z. z = y \rangle x) x) s$

by(simp add:scalingD)

} also { from sat ht \text{sP} have \((\lambda x. \sum y \in \text{UNIV}. t (\lambda x. P y * \langle \lambda z. z = y \rangle x) \leq t (\lambda x. \sum y \in \text{UNIV}. P y * \langle \lambda z. z = y \rangle x)\) by(intro sub-add-sum sound-intros, auto)

hence $(\lambda x. \sum y \in \text{UNIV}. t (\lambda x. P y * \langle \lambda z. z = y \rangle x) x) s \leq t (\lambda x. \sum y \in \text{UNIV}. P y * \langle \lambda z. z = y \rangle x) s$ by(auto)

} also { have ru1: $(\lambda x. \sum y \in \text{UNIV}. P y * \langle \lambda z. z = y \rangle x) = (\lambda x. \sum y \in \text{UNIV}. if y = x then P y else 0)$

by(rule ext [OF sum.cong]) auto

also from $s \wedge P$. have ... $\vdash P$

by(cases finite (\text{UNIV}::\text{s set}), auto simp:sum.delta)

finally have leP: $(\lambda x. \sum y \in \text{UNIV}. P y * \langle \lambda z. z = y \rangle x) \leq P$.

moreover have sound $(\lambda x. \sum y \in \text{UNIV}. P y * \langle \lambda z. z = y \rangle x)$

proof(intro soundI2 bounded-byI nnegI sum-nonneg ballI)

fix $x$

from leP have $(\sum y \in \text{UNIV}. P y * \langle \lambda z. z = y \rangle x) \leq P x$ by(auto)

also from $s \wedge P$. have ... $\leq \text{bound-of} P$ by(auto)

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

fix $y$

from $s \wedge P$. show $0 \leq P y * \langle \lambda z. z = y \rangle x$

by(auto intro:mult-nonneg-nonneg)

qed

ultimately have $(\lambda x. \sum y \in \text{UNIV}. P y * \langle \lambda z. z = y \rangle x) s \leq t P s$

using $s \wedge P$ by(auto intro:funD[OF mono-transD, OF healthy-monoD, OF ht])

} finally show \text{thesis}.

qed

Sub-distributivity

definition sub-distrib ::

\((\langle s \Rightarrow \text{real} \rangle \Rightarrow (\langle s \Rightarrow \text{real} \rangle) \Rightarrow \text{bool})\)

where

sub-distrib \$t \leftrightarrow (\forall P \ w s. \text{sound } P \rightarrow t P s \odot 1 \leq t (\lambda s'. P s' \odot 1) s)\$

lemma sub-distribI[intro]:

\[ \forall P \ w s. \text{sound } P \rightarrow t P s \odot 1 \leq t (\lambda s'. P s' \odot 1) s \] \implies \text{sub-distrib} \$t$

by(simp add:sub-distrib-def)
3.2. EXPECTATION TRANSFORMERS

lemma sub-distrib12:
\[ \bigwedge P. \text{sound } P \Rightarrow \lambda s. t P s \ominus 1 \vdash (\lambda s. P s \ominus 1) \Rightarrow \text{sub-distrib } t \]
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 eqv: equiv-trans t u
and sd: sub-distrib t
shows sub-distrib u
proof
fix P :: (\'s \Rightarrow \text{real})
assume sP: sound P
moreover have sound (\\lambda -. 0) by(auto)
ultimately show \(t P s \ominus 1 \leq u (\lambda s. P s \ominus 1) s\).
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 \text{real})
assume sP: sound P
moreover have sound (\\lambda -. 0) by(auto)
ultimately show \(t P s \ominus 1 \leq u (\lambda s. P s \ominus 1) s\).
by(rule sublinearD[OF 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 \text{real}) \Rightarrow (\'s \Rightarrow \text{real})
assumes sdt: sub-distrib t and sat: sub-add t and ht: healthy t
shows sublinear t
proof
fix P :: (\'s \Rightarrow \text{real}) and Q :: (\'s \Rightarrow \text{real})
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 \(ht \, sP \, sQ \, nna \, nnb\)
have \(saP\): sound \((\lambda s. \, a \ast P \, s)\) \textbf{and} \(staP\): sound \((\lambda s. \, a \ast t \, P \, s)\)
and \(sbQ\): sound \((\lambda s, \, b \ast Q \, s)\) \textbf{and} \(stbQ\): sound \((\lambda s, \, b \ast t \, Q \, s)\)
by\((auto \, intro; \text{sc-sound})\)

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

from \(ht \, sP \, sQ \, nna \, nnb\)
have \(a \ast t \, P \, s + b \ast t \, Q \, s = t \, (\lambda s. \, a \ast P \, s) + t \, (\lambda s. \, b \ast Q \, s)\) \(s\)
by\((\text{simp add; scalingD healthy-scalingD})\)
also from \(saP \, sbQ\) sat
have \(t \, (\lambda s. \, a \ast P \, s) + t \, (\lambda s. \, b \ast Q \, s)\) \(s \leq t \, (\lambda s. \, a \ast P \, s + b \ast Q \, s)\) \(s\) \textbf{by}(\text{blast})
finally
have \(\text{notm}: a \ast t \, P \, s + b \ast t \, Q \, s \leq t \, (\lambda s. \, a \ast P \, s + b \ast Q \, s)\) \(s\).

\textbf{show} \(a \ast t \, P \, s + b \ast t \, Q \, s \mathrel{\ominus} c \leq t \, (\lambda s'. \, a \ast P \, s' + b \ast Q \, s' \mathrel{\ominus} c)\) \(s\)
\textbf{proof}(\text{cases} \(c = 0\))
\begin{itemize}
  \item \textbf{case} \textit{True} \textbf{note} \(z = \text{this}\)
    \begin{itemize}
      \item from \(stabPQ\) have \(\forall s. \, 0 \leq a \ast t \, P \, s + b \ast t \, Q \, s\) \textbf{by}(\text{auto})
    \end{itemize}
  \item moreover from \(sabPQ\)
    \begin{itemize}
      \item have \(\forall s. \, 0 \leq a \ast P \, s + b \ast Q \, s\) \textbf{by}(\text{auto})
    \end{itemize}
  \item ultimately \textbf{show} \(\text{thesis}\) \textbf{by}(\text{simp add; z notm})
\end{itemize}
next
\begin{itemize}
  \item \textbf{case} \textit{False} \textbf{note} \(nz = \text{this}\)
    \begin{itemize}
      \item from \(nz\) \textbf{and} \(nnc\) \textbf{have} \(nni: \, 0 \leq \text{inverse } c\) \textbf{by}(\text{auto})
    \end{itemize}
  \item have \(\forall s. \, (\text{inverse } c \ast a) \ast P \, s + (\text{inverse } c \ast b) \ast Q \, s = \text{inverse } c \ast (a \ast P \, s + b \ast Q \, s)\)
    \textbf{by}(\text{simp add; divide-simps})
  \item with \(sabPQ\) \textbf{and} \(nni\)
    \begin{itemize}
      \item have \(si: \text{ sound } (\lambda s. \, (\text{inverse } c \ast a) \ast P \, s + (\text{inverse } c \ast b) \ast Q \, s)\)
        \textbf{by}(\text{auto intro; sc-sound})
      \item hence \(\text{sim: sound } (\lambda s. \, (\text{inverse } c \ast a) \ast P \, s + (\text{inverse } c \ast b) \ast Q \, s \mathrel{\ominus} 1)\)
        \textbf{by}(\text{auto intro!; tminus-sound})
    \end{itemize}
  \item from \(nz\)
    \begin{itemize}
      \item have \(a \ast t \, P \, s + b \ast t \, Q \, s \mathrel{\ominus} c = (c \ast \text{inverse } c) \ast a \ast t \, P \, s + (c \ast \text{inverse } c) \ast b \ast t \, Q \, s \mathrel{\ominus} c\)
        \textbf{by}(\text{simp})
      \item also
        \begin{itemize}
          \item have \(\ldots = c \ast (\text{inverse } c \ast a \ast t \, P \, s) + c \ast (\text{inverse } c \ast b \ast t \, Q \, s) \mathrel{\ominus} c\)
            \textbf{by}(\text{simp add; field-simps})
        \end{itemize}
    \end{itemize}
  \item also from \(nnc\)
    \begin{itemize}
      \item have \(\ldots = c \ast (\text{inverse } c \ast a \ast t \, P \, s + \text{inverse } c \ast b \ast t \, Q \, s \mathrel{\ominus} 1)\)
        \textbf{by}(\text{simp add; distrib-left tminus-left-distrib})
    \end{itemize}
\end{itemize}
3.2. EXPECTATION TRANSFORMERS

also { have X: \( s, (inverse \ c \ast \ a) \ast t \ P \ s + (inverse \ c \ast \ b) \ast t \ Q \ s = \)

inverse c \ast (a \ast t \ P \ s + b \ast t \ Q \ s) by(simp add: divide-simps)

also from nni and notm
have inverse c \ast (a \ast t \ P \ s + b \ast t \ Q \ s) \leq

inverse c \ast (t \ (\lambda s. a \ast P \ s + b \ast Q \ s) \ s)
by(blast intro:mult-left-mono)

also from nni ht subPQ
have inverse c \ast (a \ast t \ P \ s + (inverse c \ast b) \ast t \ Q \ s) \leq

(t \ (\lambda s. inverse c \ast a) \ast P \ s + (inverse c \ast b) \ast Q \ s) \ s \ominus 1
by(simp add:scalingD(OF healthy-scalingD, OF ht) algebra-simps)

finally
have (inverse c \ast a) \ast t \ P \ s + (inverse c \ast b) \ast t \ Q \ s \ominus 1 \leq

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

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

(t \ (\lambda s. inverse c \ast a) \ast P \ s + (inverse c \ast b) \ast Q \ s \ominus 1) \ s
by(blast)
}

finally
have c \ast (inverse c \ast a \ast t \ P \ s + inverse c \ast b \ast t \ Q \ s \ominus 1) \leq

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

} also from nnc ht sim
have c \ast t \ (\lambda s. inverse c \ast a \ast P \ s + inverse c \ast b \ast Q \ s \ominus 1) \ s

= t \ (\lambda s. c \ast (inverse c \ast a \ast P \ s + inverse c \ast b \ast Q \ s \ominus 1)) \ s
by(simp add:scalingD healthy-scalingD)

also from nnc
have ... = t \ (\lambda s. c \ast inverse c) \ast a \ast P \ s +

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

(c \ast inverse c) \ast b \ast Q \ s \ominus c \ s
by(simp add:field-simps)
finally
show a \ast t \ P \ s + b \ast t \ Q \ s \ominus c \leq t \ (\lambda s'. a \ast P \ s' + b \ast Q \ s' \ominus c) \ s
using nz by(simp)
qed
qed

Sub-conjunctivity

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

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

t \ P \ \\& \& \ t \ Q \vdash \ t \ (P \ \\& \& \ Q)
lemma sub-conjI[intro]:
\[ \forall P Q. [\text{sound } P;\text{ sound } Q] \Rightarrow \mathit{t} P \&\& \mathit{t} Q \vdash \mathit{t} (P \&\& Q) \] \Rightarrow sub-conj t
unfolding sub-conj-def by(simp)

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

lemma sub-conj-wp-twice:
fixes f ::′s ⇒ (′s ⇒ real) ⇒′s ⇒ real
assumes all: ∀ s. sub-conj (f s)
shows sub-conj (λ P 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
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:
\[ [\text{equiv-trans } t u;\text{ sublinear } t;\text{ healthy } t] \Rightarrow \text{sublinear } u \] 
by(iproper intro:sd-sa-sublinear healthy-equivI 
dest:equiv-sub-distrib equiv-sub-add 
sublinear-sub-distrib sublinear-subadd 
healthy-feasibleD)
3.2. EXPECTATION TRANSFORMERS

3.2.4 Determinism

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

Additivity

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

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

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

lemma additiveD:
\[
\frac{\text{additive } t; \text{sound } P; \text{sound } Q}{t (\lambda s. P s + Q s) = (\lambda s. t P s + t Q s)}
\]

by (simp add:additive-def)

lemma additiveI[intro]:
\[
\frac{\forall P \ Q. s. \text{sound } P; \text{sound } Q}{t (\lambda s. P s + Q s) s = t P s + t Q s}
\]

Unfolding additive-def by (blast)

Additivity is strictly stronger than sub-additivity.

lemma additive-sub-add:
\[\text{additive } t \Rightarrow \text{sub-add } t\]

by (simp add:sub-addI additiveD)

The additivity property extends to finite summation.

lemma additive-sum:
\[
\text{fixes } S::'s \text{ set; assumes additive: additive } t \text{ and healthy: healthy } t \text{ and finite: finite } S \text{ and sPz: } \bigwedge z. \text{sound } (P z) \text{ shows } t (\lambda x. \sum y \in S. P y x) = (\lambda x. \sum y \in S. t (P y) x)\]

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

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

assume finT: finite T

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

from additive sPz

have t (\lambda x. P z x + (\sum y \in T. P y x)) =
(\lambda x. t (P z) x + t (\lambda x. \sum y \in T. P y x) x)

by (auto intro!:sum-sound additiveD)

also from IH
Proof

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

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 $ht : \text{healthy} t$
and $sP : \text{sound} P$
shows $t \, P \, x = (\sum y \in \text{UNIV}. \ P \ y \ast t \, \langle \lambda z. \ z = y \rangle \, x)$

proof

have $\forall x. (\sum y \in \text{UNIV}. \ P \ y \ast \langle \lambda z. \ z = y \rangle \, x) =$
  $(\sum y \in \text{UNIV}. \ \text{if} \ y = x \ \text{then} \ P \ y \ \text{else} \ 0)$
by (rule sum.cong) auto
also have $\forall x. \ ... \ x = P \ x$
by (simp add: sum.delta)
finally
have $t \, P \, x = t \, (\lambda x. \ \sum y \in \text{UNIV}. \ P \ y \ast \langle \lambda z. \ z = y \rangle \, x)$
by (simp)
also {
  from $sP$ have $\forall z. \ \text{sound} \ (\lambda a. \ P \ z \ast \langle \lambda z. \ za = z \rangle \, a)$
by (auto intro!: mult.sound)
  hence $t \, \langle \lambda x. \ \sum y \in \text{UNIV}. \ P \ y \ast \langle \lambda z. \ z = y \rangle \, x \rangle \, x =$
  $(\sum y \in \text{UNIV}. \ t \, (\lambda x. \ P \ y \ast \langle \lambda z. \ z = y \rangle \, x))$
by (subst additive-sum, simp-all add:assms)
}
also from $sP$
have $(\sum y \in \text{UNIV}. \ t \, (\lambda x. \ P \ y \ast \langle \lambda z. \ z = y \rangle \, x)) =$
  $(\sum y \in \text{UNIV}. \ P \ y \ast t \, \langle \lambda z. \ z = y \rangle \, x)$
by (subst scalingD)[OF healthy-scalingD, OF ht], auto
finally show $t \, P \, x = (\sum y \in \text{UNIV}. \ P \ y \ast t \, \langle \lambda z. \ z = y \rangle \, x)$.

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 $ht : \text{healthy} t$
and $sP : \text{sound} P$
shows $t \, P \, x = (\sum y \in \{s. \ \text{G} \, s\}. \ P \ y \ast t \, \langle \lambda z. \ z = y \rangle \, x) +$
  $(\sum y \in \{s. \ \neg \text{G} \, s\}. \ P \ y \ast t \, \langle \lambda z. \ z = y \rangle \, x)$

proof --
3.2. EXPECTATION TRANSFORMERS

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

Maximality

definition
  maximal :: (('a ⇒ real) ⇒ 'a ⇒ real) ⇒ bool
where
  maximal \( t \equiv \forall c. 0 \leq c \rightarrow t \langle \lambda-. c \rangle = (\lambda-. c) \)

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

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

A transformer that is both additive and maximal is deterministic:

definition determ :: (('a ⇒ real) ⇒ 'a ⇒ real) ⇒ bool
where
  determ \( t \equiv \text{additive} \ t \land \text{maximal} \ t \)

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

lemma determ-additiveD[intro]:
  \( \text{determ} \ t \implies \text{additive} \ t \)
  by(simp add:determ-def)

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

**Lemma determ-negate:**

- **Assumes:** determ: determ t
- **Shows:** t «P» s + t «N P» s = 1

**Proof**

- **Have:** t «P» s + t «N P» s = t (λs. «P» s + «N P» s) s
  - by (simp add: additiveD determ determ-additiveD)
- **Also:**
  - have (∀ s. «P» s + «N P» s = 1)
    - by (case-tac P s, simp-all)
    - hence t (λs. «P» s + «N P» s) = t (λs. 1)
  - by (simp)
- **Also have:** t (λs. 1) = (λs. 1)
  - by (simp add: maximalD determ determ-maximalD)
- **Finally show:** thesis.

Qed

### 3.2.5 Modular Reasoning

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

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

**Lemma entails-combine:**

- **Assumes:** wp1: P ⊢ t R
  - and wp2: Q ⊢ t S
  - and sc: sub-conj t
  - and sR: sound R
  - and sS: sound S
- **Shows:** P & & Q ⊢ t (R & & S)

**Proof**

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

Qed

These allow mismatched results to be composed

**Lemma entails-strengthen-post:**

\[
\begin{align*}
\{ & P ⊢ t Q; \ \text{healthy } t; \ \text{sound } R; \ Q ⊢ R; \ \text{sound } Q \} \Rightarrow P ⊢ t R \\
& \text{by (blast intro: entails-trans)}
\end{align*}
\]
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 t (\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 \langle Q \rangle s
shows \langle P \rangle \vdash t \langle Q \rangle
proof (rule entailsI)
fix s show \langle P \rangle s \leq t \langle Q \rangle s
proof (cases P s)
  case True with wp show \langle P \rangle s \leq t \langle Q \rangle s
  by (auto)
next
  case False with h show \langle P \rangle s \leq t \langle Q \rangle s
  by (auto)
qed
d qed

The other direction works too.

lemma fold-premise:
assumes ht: healthy t
and wp: \langle P \rangle \vdash t \langle Q \rangle
shows \forall s. P s \implies 1 \leq t \langle Q \rangle s
proof (clarify)
fix s assume P s
hence 1 = \langle P \rangle s by (simp)
also from wp have \ldots \leq t \langle Q \rangle s by (auto)
finally show 1 \leq t \langle Q \rangle s.
qed
Predicate conjunction behaves as expected:

**lemma conj-post:**

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

Similar to \[\mathsf{healthy} \ ?t; \ \exists s. \ ?P s \implies 1 \leq ?t \ ?Q s \implies \ ?P \vdash ?t \ ?Q,\] but more general.

**lemma entails-pconj-assumption:**

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

and \( uQ: \mathsf{unitary} \ Q \) and \( uR: \mathsf{unitary} \ R \)

shows \( \langle P \rangle \land \mathsf{Q} \vdash t \ \mathsf{R} \)

unfolding \( \exp\conjunction \mathsf{def} \)

**proof (rule entailsI)**

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

**proof (cases \( P s \))**

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

- **next**
  
  **case False**
  
  moreover from \( uQ \) have \( Q s \leq 1 \) by (auto)
  
  ultimately show \( \mathsf{thesis} \) using \( \mathsf{assms} \) by (auto)

qed

qmed

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 \( \text{HOL.Inductive} \)), is that we do not have a complete lattice.

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

**definition** \( \text{Inf-exp} :: \langle \text{'s expect set} \rightarrow \text{'s expect} \rangle \)

**where** \( \text{Inf-exp} \, S = (\lambda \, s. \, \text{Inf} \{ f \, s \, | \, f \in S \}) \)

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

\[
[ \, P \in S; \, \forall \, P \in S. \, \text{nneq} \, P \, \] \rightarrow \text{Inf-exp} \, S \leq P
\]

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

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

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

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

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

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

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

**where** \( \text{Sup-exp} \, S = (\langle \text{if} \, S = \{\} \, \text{then} \, \lambda \, s. \, 0 \, \text{else} \, (\lambda \, s. \, \text{Sup} \{ f \, s \, | \, f \in S \} )\rangle) \)

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

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

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

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

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

\[
[ \, \forall \, P \in S. \, P \leq Q; \, \text{nneq} \, Q \, \] \rightarrow \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** \( sS: \forall \, P \in S \rightarrow \text{sound} \, P \)

**and** \( bS: \forall \, P \in S \rightarrow \text{bounded-by} \, b \, P \)

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

**proof**\((\text{cases} \, S = \{\}, \, \text{simp add:Sup-exp-def, blast, intro soundI2 bounded-byI2 nneqI2})\)

**assume** \( nS: \, S \neq \{\} \)

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

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

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

**from** \( bS \, nb \) **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 \( \lambda s. \bot \vdash \supexp s \).
qed

**Definition** \( \text{lfp-exp} :: \text{\textquotesingle \textquotesingle \, \text{\textquotesingle \textquotesingle \trans} \Rightarrow \text{\textquotesingle \textquotesingle \, \text{\textquotesingle \textquotesingle \expect} \) \]

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

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

**Lemma** \( \text{lfp-exp-greatest}: \)
\[
[ \forall P. \, [ t \, P \leq P; \, \text{sound} \, P ] \Rightarrow Q \leq P; \, t \, R \vdash R; \, \text{sound} \, R ] \Rightarrow Q \leq \text{lfp-exp} \, t
\]
\text{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** \( \text{lfp-exp-sound}: \)
\[
\text{assumes} \, fR: \, t \, R \vdash R \, \text{and} \, sR: \, \text{sound} \, R
\]
\text{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 \( R \Rightarrow \text{\lambda s. \, \text{bound-of} \, R} \) 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 \( \text{unitary} \, (\text{\lambda s. \, 1}) \) by (auto)
with \( IH \) have \( \text{unitary} \, (t \, (\text{\lambda s. \, 1})) \) by (auto)
thus \( t \, (\text{\lambda s. \, 1}) \vdash \text{\lambda s. \, 1} \) by (auto)
show \( \text{sound} \, (\text{\lambda s. \, 1}) \) by (auto)
qed

**Lemma** \( \text{lfp-exp-lemma2}: \)
fixes \( t::\text{\textquotesingle \textquotesingle \trans} \)
\[
\text{assumes} \, st: \forall P. \, \text{sound} \, P \Rightarrow \text{sound} \, (t \, P)
\]
\quad and \( mt: \text{mono-trans} \, t \)
3.3. INDUCTION

and $f \mathcal{R}$: $t \vdash R$ and $s \mathcal{R}$: sound $R$

shows $t \ (lfp\text{-}exp \ t) \leq lfp\text{-}exp \ t$

proof (rule $lfp\text{-}exp\text{-}greatest$ \[ \text{lfp} \to \text{OF} \vdash \mathcal{R} \ s \mathcal{R} \])

from $f \mathcal{R} \ s \mathcal{R}$ show sound \((t \ (lfp\text{-}exp \ t))\) by (auto intro $lfp\text{-}exp\text{-}sound \ s \mathcal{R}$)

fix $P$ :: $t$ expect

assume $fP$: $t \vdash P$ and $sP$: sound $P$

hence $lfp\text{-}exp \ t \leq (lfp\text{-}exp \ t)$ by (rule $lfp\text{-}exp\text{-}lowerbound$)

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

also note $fP$

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

qed

lemma $lfp\text{-}exp\text{-}lemma3$:

assumes $st$: $\forall P. \ \text{sound} \ P \Rightarrow \text{sound} \ (t \ P)$

and $mt$: \text{mono-trans} $t$

and $fR$: $t \vdash R$ and $sR$: sound $R$

shows $lfp\text{-}exp \ t \leq (lfp\text{-}exp \ t)$ by (iprover intro $lfp\text{-}exp\text{-}lowerbound \ lfp\text{-}exp\text{-}sound \ lfp\text{-}exp\text{-}lemma2$ \ assms $\text{mono-trans} \ [\text{OF} \ mt]$)

lemma $lfp\text{-}exp\text{-}unfold$:

assumes $nt$: $\forall P. \ \text{sound} \ P \Rightarrow \text{sound} \ (t \ P)$

and $mt$: \text{mono-trans} $t$

and $fR$: $t \vdash R$ and $sR$: sound $R$

shows $lfp\text{-}exp \ t = (lfp\text{-}exp \ t)$ by (iprover intro $\text{antisym} \ lfp\text{-}exp\text{-}lemma2 \ lfp\text{-}exp\text{-}lemma3$ \ assms)

definition $gfp\text{-}exp$ :: $t$expect

where $gfp\text{-}exp \ t = \text{Sup-exp} \ \{P. \ \text{unitary} \ P \wedge P \leq t \ P\}$

lemma $gfp\text{-}exp\text{-}upperbound$:

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

by (auto simp add $gfp\text{-}exp\text{-}def$ intro $\text{Sup-exp-upper}$)

lemma $gfp\text{-}exp\text{-}least$:

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

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

lemma $gfp\text{-}exp\text{-}bound$:

$[\forall P. \ \text{unitary} \ P \Rightarrow \text{unitary} \ (t \ P)) \Rightarrow \text{bounded-by} \ 1 \ (gfp\text{-}exp \ t)$

unfolding $gfp\text{-}exp\text{-}def$ by (rule $\text{bounded-by} \ 1 \ [\text{OF} \ \text{Sup-exp-least}], \ \text{auto}$)

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

$\text{nneg} \ (gfp\text{-}exp \ t)$

proof (intro $\text{nneg} \ 2$, simp add $gfp\text{-}exp\text{-}def$, cases)

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

show $\lambda s. \ 0 \vdash \text{Sup-exp} \ \{P. \ \text{unitary} \ P \wedge P \vdash t \ P\}$
CHAPTER 3. SEMANTIC STRUCTURES

by (simp only: empty Sup-exp-def, auto)

next

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

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

also from \text{Qin} have \text{Q} \vdash \text{Sup-exp} \{P. \ \text{unitary} \ P \land P \vdash t \ P \}

by (auto intro: Sup-exp-upper)

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

qed

lemma gfp-exp-unitary:

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

by (iprover intro: gfp-exp-nneg gfp-exp-bound unitaryI2)

lemma gfp-exp-lemma2:

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

and \text{mt}: \(\forall P \ Q. [\ \text{unitary} \ P; \ \text{unitary} \ Q; P \vdash Q] \Rightarrow t \ P \vdash t \ Q\)

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

proof (rule gfp-exp-least)

show unitary \((t \ (\text{gfp-exp} \ t))\) \ by (auto intro: gfp-exp-unitary \text{ft})

fix \ P

assume \text{fp}: \ P \leq t \ P \ and \ uP: \ \text{unitary} \ P

with \text{ft} have \ P \leq \text{gfp-exp} \ t \ by (auto intro: gfp-exp-upperbound)

with \ uP \ gfp-exp-unitary \text{ft}

have \ t \ P \leq t \ (\text{gfp-exp} \ t) \ by (blast intro: \text{mt})

with \text{fp} show \ P \leq t \ (\text{gfp-exp} \ t) \ by (auto)

qed

lemma gfp-exp-lemma3:

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

and \text{mt}: \(\forall P \ Q. [\ \text{unitary} \ P; \ \text{unitary} \ Q; P \vdash Q] \Rightarrow t \ P \vdash t \ Q\)

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

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

lemma gfp-exp-unfold:

\((\forall P. \ \text{unitary} \ P \Rightarrow \text{unitary} (t \ P)) \Rightarrow (\forall P \ Q. [\ \text{unitary} \ P; \ \text{unitary} \ Q; P \vdash Q]\ \Rightarrow \ t \ P \vdash t \ Q) \Rightarrow t \ (gfp-exp \ t) = t \ (gfp-exp \ t)\)

by (iprover intro: antisym gfp-exp-lemma2 gfp-exp-lemma3)

3.3.2 The Lattice of Transformers

In addition to fixed points on expectations, we also need to reason about fixed points on expectation transformers. The interpretation of a recursive program in pGCL is as a fixed point of a function from transformers to transformers. In contrast to the case of expectations, healthy transformers do form a complete lattice, where the bottom element is \(\lambda s. 0::'c\), and the
top element is the greatest allowed by feasibility: \( \lambda P \cdot \text{bound-of } P \).

**definition** \( \text{Inf-trans} :: 's \text{ trans set } \Rightarrow 's \text{ trans} \)

**where** \( \text{Inf-trans } S = (\lambda P. \text{Inf-exp } \{ t P \mid t. t \in S \}) \)

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

\[ \left[ t \in S; \forall u \in S. \forall P. \text{sound } P \rightarrow \text{sound } (u P) \right] \implies \text{le-trans } (\text{Inf-trans } S) t \]

**unfolding** \( \text{Inf-trans-def} \) by (rule \( \text{le-transI} \)[OF \( \text{Inf-exp-lower} \)], blast+)

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

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

**unfolding** \( \text{Inf-trans-def} \) by (auto intro: \( \text{le-transI} \)[OF \( \text{Inf-exp-greatest} \)])

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

**where** \( \text{Sup-trans } S = (\lambda P. \text{Sup-exp } \{ t P \mid t. t \in S \}) \)

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

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

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

**lemma** \( \text{Sup-trans-upper2} \):

\[ [ t \in S; \forall u \in S. \forall P. (\text{nneg } P \land \text{bounded-by } b P) \rightarrow (\text{nneg } (u P) \land \text{bounded-by } b (u P)); \]

\[ \text{nneg } P; \text{bounded-by } b P ] \implies t P \vdash (\text{Sup-trans } S) P \]

**unfolding** \( \text{Sup-trans-def} \) by (blast intro: \( \text{Sup-exp-upper} \))

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

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

**unfolding** \( \text{Sup-trans-def} \) by (auto intro:sound-nneg[OF \( \text{unitary-soand} \)], le-utransI[OF \( \text{Sup-exp-least} \)])

**lemma** \( \text{Sup-trans-least2} \):

\[ [ \forall t \in S. \forall P. \text{nneg } P \rightarrow \text{bounded-by } b P \rightarrow t P \vdash u P; \]

\[ \forall u \in S. \forall P. (\text{nneg } P \land \text{bounded-by } b P) \rightarrow (\text{nneg } (u P) \land \text{bounded-by } b (u P)); \]

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

\[ \implies \text{Sup-trans } S P \vdash u P \]

**unfolding** \( \text{Sup-trans-def} \) by (blast intro: \( \text{Sup-exp-least} \))

**lemma** \( \text{feasible-Sup-trans} \):

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

**assumes** \( fS: \forall t \in S. \text{feasible } t \)

**shows** \( \text{feasible } (\text{Sup-trans } S) \)

**proof** (cases \( S=\{\} \), simp add: Sup-trans-def Sup-exp-def, blast, intro feasibleI)

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

**assume** \( bP: \text{bounded-by } b P \text{ and } nP: \text{nneg } P \)

and \( \text{neS: } S \neq {} \)
from $\text{ncS}$ obtain $t$ where (in: $t \in S$) by(auto)
with $\text{fs}$ have $f$: feasible $t$ by(auto)
with $bP \; nP$ have $\lambda s. \; 0 \vdash t \; P$ by(auto)
also {
  from $bP \; nP$ have sound $P$ by(auto)
  with (in $\text{fs}$ have $t \; P \vdash \text{Sup-trans} \; S \; P$
      by(auto intro!: Sup-trans-upper2))
}
finally show nneg $(\text{Sup-trans} \; S \; P)$ 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 $\text{lfp-trans} :: \; (\text{'}s \; \text{trans} \Rightarrow \text{'}s \; \text{trans}) \Rightarrow \text{'}s \; \text{trans}$

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

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

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

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

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

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

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

$\text{le-trans} \; u \; (\text{lfp-trans} \; T)$

unfolding $\text{lfp-trans-def}$ by(rule Inf-trans-greatest, auto)

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

fixes $P \; Q::\; \text{'}s \; \text{expect}$

assumes $sP$: sound $P$

and $fw$: le-trans $(T \; v) \; v$

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

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

proof(intro soundI2 bounded-byI2 nnegI2)

from $fw \; sv$ have le-trans $(\text{lfp-trans} \; T) \; v$
  by(prover intro: lfp-trans-lowerbound)
with $sP$ have $(\text{lfp-trans} \; T \; P \vdash v \; P)$ by(auto)
also {
  from $sv \; sP$ have sound $(v \; P)$ by(prover)
  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 le-trans $(\lambda P \; s. \; 0) \; (\text{lfp-trans} \; T)$

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

```
fix t`:s trans
assume \( \land P. \text{sound } P \Imp \text{sound } (t P) \)
thus \( \land P. \text{sound } P \Imp \lambda s. 0 + t P \) by(auto)
next
fix P`:s expect
assume sound P thus sound (v P) by(rule sv)
next
show le-trans (T v) v by(rule fv)
qed
with sP show \( \lambda s. 0 + \text{lfp-trans } T P \) by(auto)
qed

lemma lfp-trans-unitary:
fixes P Q`:s expect
assumes uP: unitary P
and fv: le-trans (T v) v
and sv: \( \land P. \text{sound } P \Imp \text{sound } (v P) \)
and fT: le-trans (T (\( \lambda P s. \text{bound-of } P \)) (\( \lambda P s. \text{bound-of } P \))
shows unitary (lfp-trans T P)
proof (rule unitaryI)
from unitary-sound[OF uP] fv sv show sound (lfp-trans T P) by(rule lfp-trans-sound)
show bounded-by 1 (lfp-trans T P)
proof (rule bounded-byI2)
from fT have le-trans (lfp-trans T) (\( \lambda P s. \text{bound-of } P \))
by (auto intro: lfp-trans-lowerbound)
with uP have lfp-trans T P \( \vdash \lambda s. 1 \) by(auto)
finally show lfp-trans T P \( \vdash \lambda s. 1 \).
qed

qed

lemma lfp-trans-lemma2:
fixes v`:s trans
assumes mono: \( \land t u. \land P. \text{sound } P \Imp \text{sound } (t P); \land P. \text{sound } P \Imp \text{sound } (u P) \Imp \text{le-trans } (T t) (T u) \)
and nT: \( \land t P. \land Q. \text{sound } Q \Imp \text{sound } (t Q); \text{sound } P \Imp \text{sound } (T t P) \)
and fv: le-trans (T v) v
and sv: \( \land P. \text{sound } P \Imp \text{sound } (v P) \)
shows le-trans (T (lfp-trans T)) (lfp-trans T)
proof (rule lfp-trans-greatest[where T=T and v=v], simp-all add:assms)
fix t`:s trans and P`:s expect
assume ft: le-trans (T t) t and st: \( \land P. \text{sound } P \Imp \text{sound } (t P) \)
hence le-trans (lfp-trans T) t by(auto intro:lfp-trans-lowerbound)
with ft st have le-trans (T (lfp-trans T)) (T t)
by (iprover intro:mono lfp-trans-sound fv sv)
```
also note \( ft \)

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

\[ \text{qed} \]

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

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

\textbf{assumes} \( \text{mono}: \forall t \ u. \ [ \text{le-trans} \ t \ u; \ \forall P. \ \text{sound} \ P \implies \text{sound} \ (t \ P); \]
\[ \forall P. \ \text{sound} \ P \iff \text{sound} \ (u \ P) ] \implies \text{le-trans} \ (T \ t) \ (T \ u) \]
\[ \text{and} \ sT: \ \forall t \ P. \ [ \forall Q. \ \text{sound} \ Q \iff \text{sound} \ (t \ Q); \ \text{sound} \ P ] \implies \text{sound} \ (T \ t \ P) \]
\[ \text{and} \ fu: \ \text{le-trans} \ (T \ v) \ v \]
\[ \text{and} \ su: \ \forall P. \ \text{sound} \ P \iff \text{sound} \ (v \ P) \]

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

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

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

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

\textbf{have} \( n1: \forall P. \ \text{sound} \ P \iff \text{sound} \ (\text{lfp-trans} \ T \ P) \)
\[ \text{by (iprover intro: lfp-trans-sound fu sv)} \]

\textbf{with} \( sP \) \textbf{have} \( n2: \text{sound} \ (\text{lfp-trans} \ T \ P) \)
\[ \text{by (iprover intro: lfp-trans-sound fu sv sT)} \]

\textbf{with} \( n1 \ sP \) \textbf{show} \( n3: \text{sound} \ (T \ (\text{lfp-trans} \ T \ P)) \)
\[ \text{by (iprover intro: sT)} \]

\textbf{next}

\textbf{show} \( \text{le-trans} \ (T \ (T \ (\text{lfp-trans} \ T))) \ (T \ (\text{lfp-trans} \ T)) \)
\[ \text{by (rule mono[OF \ lfp-trans-lemma2, OF \ mono]}, \]
\[ \text{(iprover intro:assms \ lfp-trans-sound)}+ \]

\[ \text{qed} \]

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

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

\textbf{assumes} \( \text{mono}: \forall t \ u. \ [ \text{le-trans} \ t \ u; \ \forall P. \ \text{sound} \ P \implies \text{sound} \ (t \ P); \]
\[ \forall P. \ \text{sound} \ P \iff \text{sound} \ (u \ P) ] \implies \text{le-trans} \ (T \ t) \ (T \ u) \]
\[ \text{and} \ sT: \ \forall t \ P. \ [ \forall Q. \ \text{sound} \ Q \iff \text{sound} \ (t \ Q); \ \text{sound} \ P ] \implies \text{sound} \ (T \ t \ P) \]
\[ \text{and} \ fu: \ \text{le-trans} \ (T \ v) \ v \]
\[ \text{and} \ su: \ \forall P. \ \text{sound} \ P \iff \text{sound} \ (v \ P) \]

\textbf{shows} \( \text{equiv-trans} \ (\text{lfp-trans} \ T) \ (T \ (\text{lfp-trans} \ T)) \)
\[ \text{by (rule \ le-trans-antisym,} \]
\[ \text{rule \ lfp-trans-lemma3[OF \ mono], (iprover intro:assms)}+, \]
\[ \text{rule \ lfp-trans-lemma2[OF \ mono], (iprover intro:assms)}+ \]

\[ \text{definition \ gfp-trans :: ('s \ trans \Rightarrow 's \ trans) \Rightarrow 's \ trans} \]

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

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

\textbf{unfolding} \( \text{gfp-trans-def} \) \textbf{by(auto \ intro:Sup-trans-upper)}
3.3. INDUCTION

lemma gfp-trans-least:
\[ \begin{align*}
\forall t. & \text{le-utrans } t (T t) ; \\
\forall P. & \text{unitary } P \implies \text{unitary } (t P) \implies \text{le-utrans } t u ; \\
\forall P. & \text{unitary } P \implies \text{unitary } (u P) \implies \\
\text{le-utrans } (\text{gfp-trans } T) u
\end{align*} \]
unfolding \text{gfp-trans-def} by(\text{auto intro:Sup-trans-least})

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

lemma gfp-trans-lemma2:
\[ \begin{align*}
\text{assumes } \text{mono}: & \begin{align*}
\forall t u. & \text{le-utrans } t u ; \\
\forall P. & \text{unitary } P \implies \text{unitary } (t P) ; \\
\forall P. & \text{unitary } P \implies \text{unitary } (u P) \implies \\
\text{le-utrans } (\text{gfp-trans } T T) u
\end{align*} \\
\text{and } hT: & \begin{align*}
\forall t P. & \begin{align*}
\forall Q. & \text{unitary } Q \implies \text{unitary } (t Q) ; \\
\text{unitary } P \implies \text{unitary } (T t P)
\end{align*}
\end{align*} \]
\( (T t P) \)
shows le-utrans (gfp-trans T) (T (gfp-trans T))

proof (rule gfp-trans-least, simp-all add:hT gfp-trans-unitary)

fix t
assume fp: le-utrans t (T t) and ht: \( \forall P. \) unitary \( P \) \implies \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)).

declare lemma gfp-trans-lemma3:

assumes mono: \( \forall t \ u. \) \( \forall P. \) unitary \( P \) \implies \text{unitary} \( t \ P \);  
\( \forall P. \) unitary \( P \) \implies \text{unitary} \( u \ P \) \( \implies \) le-utrans (T t) (T u)  
and hT: \( \forall t \ P. \) \( \forall Q. \) unitary \( Q \) \implies \text{unitary} \( t \ Q \); \text{unitary} \( P \) \implies \text{unitary} \( T \ P \)  
shows le-utrans (T (gfp-trans T)) (gfp-trans T)  
by (blast intro!: mono gfp-trans-unitary gfp-trans-upperbound gfp-trans-lemma2 mono hT)

declare lemma gfp-trans-unfold:

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

3.3.3 Tail Recursion

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

declare lemma gfp-pullup:

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

proof (rule antisym)
show ?Y P ≤ ?X P
proof (rule gfp-exp-upperbound)
  from mono T lT uP have (gfp-trans T) P ≤ (T (gfp-trans T)) P
  by (auto intro: le-utransD [OF gfp-trans-lemma2])
also from uP have (T (gfp-trans T)) P = t P (gfp-trans T P) by (rule tailcall)
finally show gfp-trans T P ⊢ t P (gfp-trans T P)
from 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 λR. 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 [OF gfp-exp-unitary])
also from uQ have ... = T (λa. gfp-exp (t a)) Q by (rule tailcall symmetric)
finally show gfp-exp (t Q) ≤ T (λa. gfp-exp (t a)) Q by (simp)
qed
fix Q::'s expect assume unitary Q
with ft have λR. unitary R ⇒ unitary (t Q R) by (auto)
thus unitary (gfp-exp (t Q)) by (rule gfp-exp-unitary)
qed
qed

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

have \( \forall P. \text{sound } P \implies t P (v P) = T v P \) by (simp add: tailcall)
also have \( \forall P. \text{sound } P \implies \vdash P \vdash v P \) by (auto intro: le-transD [OF \_])
finally have \( \vdash P. \text{sound } P \implies t P (v P) \vdash v P \). have \( \vdash P. \text{sound } P \implies \text{sound } (v P) \) by (rule sv)
show \( ?X P \leq ?Y P \)
proof (rule le-transD [OF lfp-trans-lowerbound, OF - sP])
  show \( \vdash T (\lambda a. \text{lfp-exp } (t a)) ) (\lambda a. \text{lfp-exp } (t a)) \)
proof (rule le-transI)
  fix \( P::'s \) expect
  assume \( sP: \text{sound P} \)
  from \( sP \) have \( T (\lambda a. \text{lfp-exp } (t a)) P = t P (\text{lfp-exp } (t P)) \) by (rule tailcall)
also have \( t P (\text{lfp-exp } (t P)) = \text{lfp-exp } (t P) \)
  by (iprover intro: lfp-exp-unfold [symmetric] sP st mt fuP svP)
finally show \( T (\lambda a. \text{lfp-exp } (t a)) P \vdash \text{lfp-exp } (t P) \) by (simp)
qed
fix \( P::'s \) expect
assume \( sP: \text{sound P} \)
with \( fuP svP \) 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 \( S = (\text{if } S = \{\} \text{ then } \lambda P. s . t \text{ else } \text{Inf-trans } S) \)

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

end
Chapter 4

The pGCL Language

4.1 A Shallow Embedding of pGCL in HOL

theory Embedding imports Misc Induction begin

4.1.1 Core Primitives and Syntax

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

\[ \text{Abort} \ \text{either always fails, } \lambda P \ s. \ 0::'c, \ \text{or always succeeds, } \lambda P \ s. \ 1::'c. \]

\[ \text{definition} \ \text{Abort} :: \ 's \ \text{prog} \]
\[ \text{where} \ \text{Abort} \equiv \lambda ab P s. \ \text{if} \ ab \ \text{then} \ 0 \ \text{else} \ 1 \]

\[ \text{Skip} \ \text{does nothing at all.} \]

\[ \text{definition} \ \text{Skip} :: \ 's \ \text{prog} \]
\[ \text{where} \ \text{Skip} \equiv \lambda ab P. \ P \]

\[ \text{Apply} \ \text{lifts a state transformer into the space of programs.} \]

\[ \text{definition} \ \text{Apply} :: \ ('s \Rightarrow 's) \Rightarrow \ 's \ \text{prog} \]
\[ \text{where} \ \text{Apply} \ f \equiv \lambda ab P s. \ P (f s) \]

\[ \text{Seq} \ \text{is sequential composition.} \]

\[ \text{definition} \ \text{Seq} :: \ 's \ \text{prog} \Rightarrow \ 's \ \text{prog} \Rightarrow \ 's \ \text{prog} \]
\[ \text{(infixl \ ;\ ; \ 59)} \]
\[ \text{where} \ \text{Seq} \ a \ b \equiv \lambda ab. \ a \ ab \ o \ b \ ab \]

\[ \text{PC} \ \text{is probabilistic choice between programs.} \]

\[ \text{definition} \ \text{PC} :: \ 's \ \text{prog} \Rightarrow \ ('s \Rightarrow \text{real}) \Rightarrow \ 's \ \text{prog} \Rightarrow \ 's \ \text{prog} \]
\[ \text{(\_\oplus - [58,57,57] \ 57)} \]
\[ \text{where} \ \text{PC} \ a \ P \ b \equiv \lambda ab Q s. \ P s * \ a \ ab \ Q s + (1 - P s) * \ b \ ab \ Q s \]
DC is demonic choice between programs.

**Definition** DC :: 's prog ⇒ 's prog ⇒ 's prog (- [58, 57] 57)
where AC a b \equiv \lambda a b Q s. \min (a ab Q s) (b ab Q s)

AC is angelic choice between programs.

**Definition** AC :: 's prog ⇒ 's prog ⇒ 's prog (- [58, 57] 57)
where \( AC(a, b) = \lambda a b Q s. \max (a ab Q s) (b ab Q s) \)

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

**Definition** Embed :: 's trans ⇒ 's prog
where Embed t = (λab. t)

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

**Definition** Mu :: ('s prog ⇒ 's prog) ⇒ 's prog
where \( Mu(T) = \lambda a b P s. \begin{cases} \text{lfp-trans} (\lambda t. T (Embed t) ab) & \text{if } a b \text{ then} \\ \text{gfp-trans} (\lambda t. T (Embed t) ab) & \text{else} \end{cases} \)

repeat expresses finite repetition

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

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

**Definition** SetDC :: ('a ⇒ 's prog) ⇒ ('s ⇒ 'a set) ⇒ 's prog
where \( SetDC(f, S) = \lambda a b P s. \text{Inf} ((\lambda a. f a ab P s) \cdot S s) \)

**Syntax** -SetDC :: pttrn => ('s => 'a set) => 's prog => 's prog
(\prod x \in S. P) = CONST SetDC (%x. P) S

The above syntax allows us to write \( \prod x \in S. \text{Apply} f \)

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

**Definition** SetPC :: ('a ⇒ 's prog) ⇒ ('s ⇒ 'a ⇒ real) ⇒ 's prog
where \( SetPC(f, p) = \lambda a P s. \sum a \in \text{supp} (p s). p s a \cdot f a ab P s \)

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

This gives us something like let syntax

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

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

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

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

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

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

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

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

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

**syntax** -assign \( : \) ident \( \Rightarrow \) 'a \( \Rightarrow \) 's prog \( (\text{- := } [1000,900])900 \)

**ML**

\[
\text{fun assign-tr - [Const (name,-), ary]} = \\
\text{ Const (Embedding.Apply, dummyT) } \& \\
\text{ Abs (s, dummyT, Syntax.const (suffix Record.updateN name) } \& \\
\text{ Abs (Name.\&uu-, dummyT, ary } \& \text{ Bound 1 } \& \text{ Bound 0) } \\
\text{ | assign-tr - ts = raise TERM (assign-tr, ts) }
\]

**parse-translation** :[@{syntax-const -assign, assign-tr}]

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

**ML**

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

**Translations**

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

**Definition**

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

**Definition**

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

If-Then-Else as a degenerate probabilistic choice.

**Abstraction**

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

Syntax for loops

**Abstraction**

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

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

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

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

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

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

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

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

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

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

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

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

lemma eval-wlp-AC:
wlp (a ▷ b) P = (λs. max (wlp a P s) (wlp b P s))
unfolding wlp-def AC-def by(simp)

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

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

lemma eval-wp-SetDC:
wp (SetDC p S) R s = Inf ((λa. wp (p a) R s) ' S s)
unfolding wp-def SetDC-def by(simp)

lemma eval-wlp-SetDC:
wlp (SetDC p S) R s = Inf ((λa. wlp (p a) R s) ' S s)
unfolding wlp-def SetDC-def by(simp)

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

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

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

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

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

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

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

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

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

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

Use these as simp rules to clear out Skips

lemmas skip-simps = Skip-Seq Seq-Skip

end

4.2 Healthiness

theory Healthiness imports Embedding begin

4.2.1 The Healthiness of the Embedding

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

lemma healthy-wp-Abort:
  healthy (wp Abort)
proof (rule healthy-parts)
  fix b and P::'a ⇒ real
  assume nP: nneg P and bP: bounded-by b P
  thus bounded-by b (wp Abort P)
    unfolding wp-eval by(blast)
  show nneg (wp Abort P)
    unfolding wp-eval by(blast)
next
  fix P Q::'a expect
  show wp Abort P ⊢ wp Abort Q
    unfolding wp-eval by(blast)
next
  fix P and c and s::'a
  show c * wp Abort P s = wp Abort (λs. c * P s) s
    unfolding wp-eval by(auto)
qed
lemma nearly-healthy-wlp-Abort:
nearly-healthy (wlp Abort)
proof (rule nearly-healthyI)
fix P :: 's ⇒ real
  show unitary (wlp Abort P)
    by (simp add: wp-eval)
next
fix P Q :: 's expect
  assume P ⊢ Q and unitary P and unitary Q
  thus wlp Abort P ⊢ wlp Abort Q
    unfolding wp-eval by (blast)
qed

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

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

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

proof (rule nearly-healthyI, simp-all add: wp-eval)
fix $b$ and $P$: $s \Rightarrow \text{real}$
assume unitary $P$
with $hu$ have unitary $(\text{wlp } u P)$ by (auto)
with $ht$ show unitary $(\text{wlp } t (\text{wlp } u P))$ by (auto)

next
fix $P Q$: $s \Rightarrow \text{real}$
assume unitary $P$ and unitary $Q$ and $P \vdash Q$
with $hu$ have unitary $(\text{wlp } u P)$ and unitary $(\text{wlp } u Q)$
and $\text{wlp } u P \vdash \text{wlp } u Q$ by (auto)
with $ht$ show $\text{wlp } t (\text{wlp } u P) \vdash \text{wlp } t (\text{wlp } u Q)$ by (auto)

qed

lemma healthy-wp-PC:
fixes $f$: $\text{prs}$
assumes $hf$: healthy $(\text{wp } f)$ and $hg$: healthy $(\text{wp } g)$
and $uP$: unitary $P$
shows healthy $(\text{wp } (f \oplus g))$

proof (intro healthy-parts bounded-byI nnegI le-funI, simp-all add: wp-eval)
fix $b$ and $Q$: $s \Rightarrow \text{real}$ and $s$: $s$
assume $nQ$: nneg $Q$ and $bQ$: bounded-by $b$ $Q$

Non-negative:

from $nQ$ and $bQ$ and $hf$ have $0 \leq \text{wp } f Q s$ by (auto)
with $uP$ have $0 \leq P s * ...$ by (auto intro: mult-nonneg-nonneg)

moreover {
  from $uP$ have $0 \leq 1 - P s$
  by (auto)
  with $nQ$ and $bQ$ and $hg$ have $0 \leq ... * \text{wp } g Q s$
  by (metis healthy-nnegD2 mult-nonneg-nonneg nneg-def)
}

ultimately show $0 \leq P s * \text{wp } f Q s + (1 - P s) * \text{wp } g Q s$
by (auto intro: mult-nonneg-nonneg)

Bounded:

from $nQ$ $bQ$ $hf$ have $\text{wp } f Q s \leq b$ by (auto)
with $uP$ $nQ$ $bQ$ $hf$ have $P s * \text{wp } f Q s \leq P s * b$
by (blast intro!: mult-mono)

moreover {
  from $nQ$ $bQ$ $hg$ $uP$
  have $\text{wp } g Q s \leq b$ and $0 \leq 1 - P s$
  by (auto)
  with $nQ$ $bQ$ $hg$ have $(1 - P s) * \text{wp } g Q s \leq (1 - P s) * b$
  by (blast intro!: mult-mono)
}

ultimately have $P s * \text{wp } f Q s + (1 - P s) * \text{wp } g Q s \leq P s * b + (1 - P s) * b$
by (blast intro:add-mono)
also have \( \cdots = b \) by (auto simp: algebra-simps)
finally show \( P \ s \ast wp f \ Q \ s + (1 - P \ s) \ast wp g \ Q \ s \leq b \).
next

Monotonic:

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

Scaling:

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

lemma nearly-healthy-wlp-PC:
fixes \( f ::'s \text{ prog} \)
assumes \( hf : \text{nearly-healthy (wp f)} \)
  and \( hg : \text{nearly-healthy (wp g)} \)
  and \( uP : \text{unitary P} \)
shows \( \text{nearly-healthy (wp (f \oplus g))} \)
proof (intro nearly-healthyI unitaryI2 nnegI bounded-byI le-funI, 
  simp-all add: wp-eval)
fix \( Q ::'s \text{ expect} \) and \( s ::'s \)
assume \( uQ : \text{unitary Q} \)
from \( uQ \) \( hf \) \( hg \) have \( uQ : \text{unitary (wp f Q)} \ \text{unitary (wp g Q)} \) by (auto)
from \(uP\) have \(nP\): \(0 \leq P \cdot s \leq 1 - P \cdot s\)
by \(auto\)
moreover from \(uQ\) have \(0 \leq \text{wlp} f \cdot Q \cdot s \leq \text{wlp} g \cdot Q \cdot s\) by(\(auto\))
ultimately show \(0 \leq P \cdot s \cdot \text{wlp} f \cdot Q \cdot s + (1 - P \cdot s) \cdot \text{wlp} g \cdot Q \cdot s\)
by(\(auto\) intro; \(add\)-\(nonneg\)-\(nonneg\) \(mult\)-\(nonneg\)-\(nonneg\))

from \(uQ\) have \(\text{wlp} f \cdot Q \cdot s \leq 1\) \(\text{wlp} g \cdot Q \cdot s \leq 1\) by(\(auto\))
with \(nP\) have \(P \cdot s \cdot \text{wlp} f \cdot Q \cdot s \leq P \cdot s \cdot \text{wlp} f \cdot R \cdot s\)
by(\(auto\) intro; \(mult\)-\(left\)-\(mono\))
moreover \{
from \(uQ\) \(uR\) \(le\) have \(\text{wlp} g \cdot Q \cdot s \leq \text{wlp} g \cdot R \cdot s\)
by(\(auto\) intro; \(le\)-\(fun\)-\(D\) [OF \(nearly\)-\(healthy\)-\(mono\)-\(D\), \(OF\) \(h\f\)])
with \(nP\) have \((1 - P \cdot s) \cdot \text{wlp} g \cdot Q \cdot s \leq (1 - P \cdot s) \cdot \text{wlp} g \cdot R \cdot s\)
by(\(auto\) intro; \(mult\)-\(left\)-\(mono\))
}\ultimately show \(P \cdot s \cdot \text{wlp} f \cdot Q \cdot s + (1 - P \cdot s) \cdot \text{wlp} g \cdot Q \cdot s \leq P \cdot s \cdot \text{wlp} f \cdot R \cdot s + (1 - P \cdot s) \cdot \text{wlp} g \cdot R \cdot s\)
by(\(auto\))

fix \(R\):’s expect
assume \(uR\): unitary \(R\) and \(le\): \(Q \vdash R\)
with \(uQ\) have \(\text{wlp} f \cdot Q \cdot s \leq \text{wlp} f \cdot R \cdot s\)
by(\(auto\) intro; \(le\)-\(fun\)-\(D\) [OF nearly-\(healthy\)-\(mono\)-\(D\), \(OF\) \(h\g\)])
with \(nP\) have \((1 - P \cdot s) \cdot \text{wlp} g \cdot Q \cdot s \leq (1 - P \cdot s) \cdot \text{wlp} g \cdot R \cdot s\)
by(\(auto\) intro; \(mult\)-\(left\)-\(mono\))

next
fix \(P\):’s \(\Rightarrow\) real and \(Q\) and \(s\):’s
from \(assms\) show \(0 \leq \min(\text{wp} f \cdot P \cdot s) \cdot (\text{wp} g \cdot P \cdot s)\) by(\(auto\))
thus \( \min (wp f P s) (wp g P s) \leq \min (wp f Q s) (wp g Q s) \) by\(\text{auto}\)

next

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

qed

lemma \(\text{nearly-healthy-wlp-DC}:\)
fixes \(f::'s \text{ prog}\)
assumes \(hf: \text{nearly-healthy } (wp f) \) and \(hg: \text{nearly-healthy } (wp g)\)
shows \(\text{nearly-healthy } (wp (f \sqcup g))\)
proof\(\text{(intro nearly-healthyI bounded-byI nnegI le-funI unitaryI2, simp-all add: wp-eval, safe)}\)
fix \(P::'s \Rightarrow \text{real} \) and \(s::'s\)
assume \(uP: \text{unitary } P\)
with \(hf \) \(hg\) have \(atP: \text{unitary } (wp f P) \text{ unitary } (wp g P)\) by\(\text{auto}\)
thus \(0 \leq wp f P s \) \(0 \leq wp g P s\) by\(\text{auto}\)
have \(\min (wp f P s) (wp g P s) \leq wp f P s\) by\(\text{auto}\)
also from \(atP\) have \(\ldots \leq 1\) by\(\text{auto}\)
finally show \(\min (wp f P s) (wp g P s) \leq 1\)
\(\text{.}\)

fix \(Q::'s \Rightarrow \text{real} \)
assume \(uQ: \text{unitary } Q \) and \(le: P \vdash Q\)
have \(\min (wp f P s) (wp g P s) \leq wp f P s\) by\(\text{auto}\)
also from \(uP\) \(uQ\) \(le\) have \(\ldots \leq wp f Q s\)
by\(\text{auto intro: le-funD[of nearly-healthy-monoD, OF hf]}\)
finally show \(\min (wp f P s) (wp g P s) \leq wp f Q s\)
\(\text{.}\)

have \(\min (wp f P s) (wp g P s) \leq wp g P s\) by\(\text{auto}\)
also from \(uP\) \(uQ\) \(le\) have \(\ldots \leq wp g Q s\)
by\(\text{auto intro: le-funD[of nearly-healthy-monoD, OF hg]}\)
finally show \(\min (wp f P s) (wp g P s) \leq wp g Q s\)
\(\text{.}\)
qed

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

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

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

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

next 
fix \( P::'s \Rightarrow \text{real} \) and \( c::\text{real} \) and \( s::'s \)
assume \( sP:: \text{sound} P \) and \( \text{pos}:: 0 \leq c \)
from \( \text{assms} \) have \( sf:: \text{scaling} (wp f) \) and \( sg:: \text{scaling} (wp g) \) by\((auto)\)
from \( \text{pos} \) have \( c \ast \max (wp f P s)\) \((wp g P s) \)
by\((\text{simp add: max-distrib})\)
also from \( sP \) and \( \text{pos} \)
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 \( f::'s \) \( \text{prog} \)
assumes \( hf:: \text{nearly-healthy} (wp f) \)
and \( hg:: \text{nearly-healthy} (wp g) \)
shows nearly-healthy \((wp (f \cup g))\)
proof\((\text{intro nearly-healthyI bounded-byI nnegI unitaryI2 le-funI}, \text{simp-all only:wp-eval})\)
fix \( b \) and \( P::'s \Rightarrow \text{real} \) and \( s::'s \)
assume \( uP:: \text{unitary} P \)

with \( bf \) have \( wp f P s \leq 1 \) by\((auto)\)
moreover from \( uP hg \) have \( \text{unitary} (wp g P) \) by\((auto)\)
hence \( wp g P s \leq 1 \) by\((auto)\)
ultimately show \( \max (wp f P s) \) \((wp g P s) \leq 1 \) by\((auto)\)
from uP hf have unitary (wlp f P) by(auto)
  hence 0 ≤ wlp f P s by(auto)
  thus 0 ≤ max (wlp f P s) (wlp g P s) by(auto)
next
fix P::'s ⇒ real and Q and s::'s
assume uP: unitary P and uQ: unitary Q and le: P ⊢ Q
  hence wlp f P s ≤ wlp f Q s and wlp g P s ≤ wlp g Q s
  by(auto intro:le-funD[OF nearly-healthy-monoD, OF hf])
  thus max (wlp f P s) (wlp g P s) ≤ max (wlp f Q s) (wlp g Q s) by(auto)
qed

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

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

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

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

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

from nonempty obtain \( x \) where \( x \in (\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s\) by(blast)
moreover from \( sP\) and healthy
have \( \forall x \in (\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s, \ 0 \leq x\) by(auto)
ultimately have \( \text{Inf} \ ((\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s) \leq x\)
\quad by(intro cInf-lower bdd-belowI, auto)
also from \( x\in\) and healthy and \( sP\) and \( bP\) have \( x \leq b\) by(blast)
finally show \( \text{Inf} \ ((\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s) \leq b\).

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

from nonempty obtain \( x\) where \( x \in (\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s\) by(blast)
moreover from \( sP\) and healthy
have \( \forall x \in (\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s, \ 0 \leq x\) by(auto)
moreover
have \( \forall x \in (\lambda a. \ wp\ (\text{prog } a)\ Q\ s) \ \because \ S\ s, \ \exists y \in (\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s, \ y \leq x\)
proof(rule ballI, clarify, rule bexI)
fix \( x\) and \( a\) assume \( a\in\) and \( S\ s\)
with healthy and \( sP\) and \( sQ\) and \( \text{le}\) show \( wp\ (\text{prog } a)\ P\ s \leq wp\ (\text{prog } a)\ Q\ s\)
\quad by(auto dest:mono-transD[OF healthy-monoD])
from \( a\in\) show \( wp\ (\text{prog } a)\ P\ s \in (\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s\) by(simp)
qed
ultimately
show \( \text{Inf} \ ((\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s) \leq \text{Inf} \ ((\lambda a. \ wp\ (\text{prog } a)\ Q\ s) \ \because \ S\ s)\)
\quad by(intro cInf-mono, blast+)
next
fix \( P::a \Rightarrow \text{real}\) and \( c::\text{real}\) and \( s::a\)
assume \( sP::\text{sound}\) and \( \text{pos}\) c: \( \emptyset \leq c\)
from nonempty obtain \( x\) where \( x \in (\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s\) by(blast)
have \( c \in \text{Inf} \ ((\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s) = \text{Inf} \ ((\star ) c \ (\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s)\) (is \( ?U = {?V}\))
proof(rule antisym)
show \( ?U \leq {?V}\)
\quad proof(rule cInf-greatest)
\quad \quad from nonempty show \( \star ) c \ (\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s \neq \emptyset\) by(auto)
\quad fix \( x\) assume \( x \in \star ) c \ (\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s\)
\quad then obtain \( y\) where \( \text{yin}::y \in (\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s\) and \( \text{rwx}\) c: \( x = c\)
\quad by(auto)
\quad have \( \text{Inf} \ ((\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s) \leq y\)
\quad proof(intro cInf-lower[OF yin] bdd-belowI)
\quad \quad fix \( z\) assume \( z \in (\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s\)
\quad then obtain \( a\) where \( a \in S\ s\) and \( z = wp\ (\text{prog } a)\ P\ s\) by(auto)
\quad with \( sP\) show \( 0 \leq z\) by(auto dest:healthy)
\quad qed
\quad \quad \quad with \( \text{pos}\) \( \text{rwx}\) \( \text{show}\ c \in \text{Inf} \ ((\lambda a. \ wp\ (\text{prog } a)\ P\ s) \ \because \ S\ s) \leq x\) by(auto)
lemma nearly-healthy-wlp-SetDC:
  fixes prog : 'b ⇒ 'a prog and S : 'a ⇒ 'b set

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

lemma nearly-healthy-wlp-SetDC:
4.2. HEALTHINESS

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

proof

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

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

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

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

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

lemma healthy-wp-SetPC:
    fixes $p::'a \Rightarrow 'a \Rightarrow \text{real$
and \( f \cdot 'a \Rightarrow 's \) prog

assumes healthy: \( \forall a. a \in \text{supp} (p s) \Rightarrow \text{healthy} (wp (f a)) \)

and sound: \( \forall s. \text{sound} (p s) \)

and sub-dist: \( \forall s. (\sum a \in \text{supp} (p s). p s a) \leq 1 \)

shows healthy \( (wp (\text{SetPC} f p)) (\text{is} \text{healthy} \ ?X) \)

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

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

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

**hence** \( sP : \text{sound} P \)** **by(auto)**

from \( sP \) and \( bP \) and healthy have \( \forall a. a \in \text{supp} (p s) \Rightarrow wp (f a) P s \leq b \)

**by(blast dest:healthy-bounded-byD)**

with sound have \( (\sum a \in \text{supp} (p s). p s a \ast wp (f a) P s) \leq (\sum a \in \text{supp} (p s). p s a \ast b) \)

**by(blast intro:sum-mono mult-left-mono)**

also have \( ... = (\sum a \in \text{supp} (p s). p s a) \ast b \)

by(simp add:sum-distrib-right)

also {\]

from \( bP \) and \( nP \) have \( 0 \leq b \) **by(blast)**

with sub-dist have \( (\sum a \in \text{supp} (p s). p s a) \ast b \leq 1 \ast b \)

**by(rule mult-right-mono)**

}

also have \( 1 \ast b = b \) **by(simp)**

finally show \( (\sum a \in \text{supp} (p s). p s a \ast wp (f a) P s) \leq b \).

show \( 0 \leq (\sum a \in \text{supp} (p s). p s a \ast wp (f a) P s) \)

**proof** (**rule** nonneg [OF mult-nonneg nonneg])

fix \( x \)

from sound show \( 0 \leq p s x \) **by(blast)**

assume \( x \in \text{supp} (p s) \)** with \( sP \) and healthy

show \( 0 \leq wp (f x) P s \)** **by(blast)**

**qed**

**next**

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

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

with healthy have \( \forall a. a \in \text{supp} (p s) \Rightarrow wp (f a) P s \leq wp (f a) Q s \)

**by(blast)**

with sound show \( (\sum a \in \text{supp} (p s). p s a \ast wp (f a) P s) \leq (\sum a \in \text{supp} (p s). p s a \ast wp (f a) Q s) \)

**by(blast intro:sum-mono mult-left-mono)**

**next**

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

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

have \( c \ast (\sum a \in \text{supp} (p s). p s a \ast wp (f a) P s) = (\sum a \in \text{supp} (p s). p s a \ast (c \ast wp (f a) P s)) \)

(by simp add:sum-distrib-left ac-simps)

also from sound and pos and healthy

have \( ... = (\sum a \in \text{supp} (p s). p s a \ast wp (f a) (\lambda s. c \ast P s) s) \)
4.2. HEALTHINESS

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

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

next
fix P::'s expect and Q::'s expect and s
assume uP: unitary P and uQ: unitary Q and le: P ⊢ Q
hence \( \forall a. a \in \text{supp} \ (p \ s) \implies \text{wlp} \ (f \ a) \leq \text{wlp} \ (f \ a) \)
by(blast intro:le-funD[OF nearly-healthy-monoD, OF healthy])
with sound show \( (\sum a\in\text{supp} \ (p \ s). \ p \ s \ a \ast \text{wlp} \ (f \ a)) \leq (\sum a\in\text{supp} \ (p \ s). \ p \ s \ a \ast \text{wlp} \ (f \ a)) \)
    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-healthyI 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 (wlp (p (f s)))
  shows nearly-healthy (wlp (Bind f p))
proof (intro nearly-healthyI unitaryI2 nnegI bounded-byI le-funI, simp-all only:wp-eval)
  fix P::'s expect and s::'s assume uP: unitary P
  with hsub have unitary (wlp (p (f s))) P by(auto)
  thus 0 ≤ wp (p (f s)) P s ≤ wp (p (f s)) P s ≤ 1 by(auto)
next
  fix Q::'s expect
  assume unitary Q P ⊢ Q
  with uP show wp (p (f s)) P s ≤ wp (p (f s)) Q s
     by (blast intro:le-funD[OF nearly-healthy-monoD,OF hsub])
qed

4.2.2 Healthiness for Loops

lemma wp-loop-step-mono:
  fixes t::'s trans
  assumes hb: healthy (wp body)
  and le: le-trans t u
  and ht: ∃ P. sound P =⇒ sound (t P)
  and hu: ∃ P. sound P =⇒ sound (u P)
  shows le-trans (wp (body :: Embed t ∈ G op Skip))
          (wp (body :: Embed u ∈ G op Skip))
proof (intro le-transI le-funI, simp add:wp-eval)
4.2. HEALTHINESS

\[\text{fix } P.s \text{ expect and } s.\text{'}s\]
\[\text{assume } sP \text{: sound } P\]
\[\text{with le have } t P \vdash u P \text{ by(auto)}\]
\[\text{moreover from } sP \text{ ht hu have sound } (t P) \text{ sound } (u P) \text{ by(auto)}\]
\[\text{ultimately have } wp \text{ body } (t P) s \leq wp \text{ body } (u P) s\]
\[\text{by(auto intro:le-funD[OF mono-transD, OF healthy-monoD, OF hb])}\]
\[\text{thus } «G» s * wp \text{ body } (t P) s \leq «G» s * wp \text{ body } (u P) s\]
\[\text{by(auto intro:mult-left-mono)}\]
\[\text{qed}\]

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

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

\[\text{lemma lfp-loop-fp:}\]
\[\text{assumes } hb \text{: healthy } (wp \text{ body})\]
\[\text{and } sP \text{: sound } P\]
\[\text{shows } \lambda s. «G» s * wp \text{ body } (\lambda s. \text{ bound-of } P) s + «N» G s \leq \lambda s. \text{ bound-of } P\]
\[\text{proof(rule le-funI)}\]
\[\text{fix } s\]
\[\text{from } sP \text{ have sound } (\lambda s. \text{ bound-of } P) \text{ by(auto)}\]
\[\text{moreover hence bounded-by } (\text{bound-of } P) (\lambda s. \text{ bound-of } P) \text{ by(auto)}\]
\[\text{ultimately have bounded-by } (\text{bound-of } P) (wp \text{ body } (\lambda s. \text{ bound-of } P))\]
\[\text{using } hb \text{ by(auto)}\]
\[\text{hence wp \text{ body } (\lambda s. \text{ bound-of } P) s \leq \text{ bound-of } P \text{ by(auto)}\]
\[\text{moreover from } sP \text{ have } P s \leq \text{ bound-of } P \text{ by(auto)}\]
\[\text{ultimately have } «G» s * wp \text{ body } (\lambda a. \text{ bound-of } P) s + (1 - «G» s) \leq \text{ bound-of } P\]
\[\text{by(blast intro:add-mono mult-left-mono)}\]
\[\text{thus } «G» s * wp \text{ body } (\lambda a. \text{ bound-of } P) s + «N» G s \leq \text{ bound-of } P\]
\[\text{by(simp add:algebra-simps negate-embed)}\]
qed

**Lemma lfp-loop-greatest:**

**Fixes** $P$ :: $′$s expect

**Assumes** $lb$: $\forall R. $ $\lambda_s. ($ $G$ $s$ $*$ wp body $R$ $s$ $+$ $\langle N \rangle$ $G$ $s$ $*$ $P$ $s$ $\vdash R$ $\implies$ sound $R$

$\implies Q \vdash R$

and $hb$: healthy (wp body)

and $sP$: sound $P$

and $sQ$: sound $Q$

**Shows** $Q \vdash lfp-exp (\lambda Q s. ($ $G$ $s$ $*$ wp body $Q$ $s$ $+$ $\langle N \rangle G$ $s$ $*$ $P$ $s$))

**Using** $sP$ by ($auto$ intro!:$lfp-exp-greatest (OF lb sQ)$ $sP$ lfp-loop-fp $hb$)

**Lemma lfp-loop-sound:**

**Fixes** $P$ :: $′$s expect

**Assumes** $hb$: healthy (wp body)

and $sP$: sound $P$

**Shows** sound ($lfp-exp (\lambda Q s. ($ $G$ $s$ $*$ wp body $Q$ $s$ $+$ $\langle N \rangle G$ $s$ $*$ $P$ $s$))

**Using** $assms$ by ($auto$ intro!:$lfp-exp-sound lfp-loop-fp$)

**Lemma wlp-loop-step-unitary:**

**Fixes** $t$ $u$ :: $′$s trans

**Assumes** $hb$: nearly-healthy (wlp body)

and $ht$: $\forall P. $ unitary $P \implies$ unitary ($t P$)

and $uP$: unitary $P$

**Shows** unitary ($wlp$ (body $;;$ Embed $t$ $G$ $s$ $\oplus$ Skip) $P$)

**Proof** ($intro$ unitaryI2 $nnegI$ bounded-byI, simp-all add:wp-eval)

**Fix** $s$ :: $′$s

**From** $ht uP$ have $utP$: unitary ($t P$) by ($auto$)

with $hb$ have unitary ($wlp$ body ($t P$)) by ($auto$)

**Hence** $0 \leq wlp$ body ($t P$) $s$ by ($auto$)

with $uP$ show $0 \leq$ ($G$ $s$ $*$ $wlp$ body ($t P$) $s$ $+$ ($1 - $ ($G$ $s$) $*$ $P$ $s$)

by ($auto$ intro!:add-nonneg-nonneg mult-nonneg-nonneg)

**From** $ht uP$ have bounded-by 1 ($t P$) by ($auto$)

with $utP$ $hb$ have bounded-by 1 ($wlp$ body ($t P$)) by ($auto$)

**Hence** $wlp$ body ($t P$) $s$ $\leq$ 1 by ($auto$)

with $uP$ have $G$ $s$ $*$ $wlp$ body ($t P$) $s$ $+$ ($1 - $ ($G$ $s$) $*$ $P$ $s$ $\leq$ ($G$ $s$ $*$ 1 $+$ ($1 - $ ($G$ $s$) $*$ 1

by ($blast$ intro!:add-mono mult-left-mono)

also have ... $= 1$ by ($simp$)

finally show $G$ $s$ $*$ $wlp$ body ($t P$) $s$ $+$ ($1 - $ ($G$ $s$) $*$ $P$ $s$ $\leq$ 1 .

qed

**Lemma wp-loop-step-sound:**

**Fixes** $t$ $u$ :: $′$s trans

**Assumes** $hb$: healthy (wp body)

and $ht$: $\forall P. $ sound $P \implies$ sound ($t P$)

and $sP$: sound $P$

**Shows** sound ($wp$ (body $;;$ Embed $t$ $G$ $s$ $\oplus$ Skip) $P$)

**Proof** ($intro$ soundI2 $nnegI$ bounded-byI, simp-all add:wp-eval)
4.2. HEALTHINESS

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

from ht sP have sound (t P) by(auto)
moreover hence bounded-by (bound-of (t P)) (t P) by(auto)
ultimately have wp body (t P) s ≤ bound-of (t P) using hh 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 ut: ∃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)
CHAPTER 4. THE PGCL LANGUAGE

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

from healthy uQ have bounded-by 1 (wlp body Q) by(auto)
with uP have «G» s * wlp body Q s + (1 − «G» s) * P s ≤ «G» s * 1 +
(1 − «G» s) * 1
by(blast intro:add-mono mult-left-mono)
also have ... = 1 by(simp)
finally show «G» s * wlp body Q s + «N G» s * P s ≤ 1
by(simp add:negate-embed)
qed

fix t u::'s trans
assume le-utrans t u
\land P. unitary P \implies unitary (t P)
\land P. unitary P \implies unitary (u P)
thus le-utrans (wlp (?Z t)) (wlp (?Z u))
by(blast intro!:wlp-loop-step-mono[OF healthy])
qed

lemma wp-loop-sound:
assumes sP: sound P
and hb: healthy (wp body)
shows sound (wp do G → body od P)
proof(simp only: wp-eval, intro lfp-trans-sound sP)
let ?v = \lambda P s. bound-of P
show le-trans (wp (body :: Embed ?v s G ⊕ Skip)) ?v
by(intro le-transI, simp add:wp-eval lfp-loop-fp[unfolded negate-embed] hb)
show \land P. sound P \implies 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 * wp body Q s + \langle N \rangle s * P s)
(is \langle X \rangle = lfp-exp (\langle Y \rangle P))

proof –
let \langle Z \rangle u = (body ;; Embed u \langle G \rangle \oplus Skip)
show \langle thesis \rangle
proof(simp only: wp-eval, intro lfp-pulldown assms le-funI sP mono-transI)
fix u P
show wp (\langle Z \rangle u) P = \langle Y \rangle P (u P) by(simp add: wp-eval negate-embed)
next
fix t::'s trans and P::'s expect
assume at: \forall Q. sound Q \Rightarrow sound (t Q) and uP: sound P
with healthy show sound (wp (\langle Z \rangle t) P) by(rule wp-loop-step-sound)
next
fix P Q::'s expect
assume sP: sound P and sQ: sound Q
show sound (\lambda a. \langle G \rangle a * wp body Q a + \langle N \rangle \langle G \rangle a * P a)
proof(intro soundI2 nnegI bounded-byI)
fix s::'
from sQ have nneg Q bounded-by (bound-of Q) Q by(auto)
with healthy have bounded-by (bound-of Q) (wp body Q) by(auto)
hence wp body Q s \leq bound-of Q by(auto)
hence wp body Q s \leq max (bound-of P) (bound-of Q) by(auto)
moreover {
from sP have P s \leq bound-of P by(auto)
hence P s \leq max (bound-of P) (bound-of Q) by(auto)
}
ultimately have \langle G \rangle s * wp body Q s + \langle N \rangle \langle G \rangle s * P s \leq
\langle G \rangle s * max (bound-of P) (bound-of Q) +
\langle N \rangle \langle G \rangle s * max (bound-of P) (bound-of Q)
by(auto intro!:add-mono mult-left-mono)
also have ... = max (bound-of P) (bound-of Q) by(simp add: algebra-simps negate-embed)
finally show \langle G \rangle s * wp body Q s + \langle N \rangle \langle G \rangle s * P s \leq max (bound-of P) (bound-of Q).

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

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

next
from healthy show le-trans (wp (?Z (\( \lambda P. \text{sound } P \) bound-of P))) (\( \lambda P. \text{bound-of } P \))
by(intro le-transI, simp add:wp-eval lfp-loop-fp[unfolded negat-embed])

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

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

show \( \lambda s. 0 \vdash gfp-exp (?X P) \)
proof(rule gfp-exp-upperbound)
show unitary (\( \lambda s. 0::real \)) by(auto)
with 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 gfp-exp (?X P) \( \vdash \lambda s. 1 \)
proof(rule gfp-exp-least)
show unitary (\( \lambda s. 1::real \)) by(auto)
fix Q::'s expect
assume unitary Q
thus \( Q \vdash \lambda s. 1 \) by(auto)
qed

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

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

**Lemma** healthy-wp-loop:

*C. expect assumption *R: unitary *R

*assume *fp: *R ⊢ ?X *P *R

*also from *le have ... ⊢ ?X *Q *R

*by(blast intro:add-mono mult-left-mono *le-funI)

*finally show *R ⊢ gfp-exp (?X *Q)

*using *uR by(auto intro:gfp-exp-upperbound)

*next

*show unitary (gfp-exp (?X *Q))

*proof(rule gfp-exp-unitary, intro unitaryI2 nneqI bounded-byI)

*fix *R::'s expect and *s::'s assume *uR: unitary *R

*with *hb have *uP: unitary (wp body *R) by(auto)

*with *uQ show 0 ≤ « *G » *s * wp body *R *s + « *N *G » *s * *Q *s

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

*from *uP *uQ have wp body *R *s ≤ 1 *Q *s ≤ 1 by(auto)

*hence « *G » *s * wp body *R *s + « *N *G » *s * *Q *s ≤ « *G » *s * 1 + « *N *G » *s

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

*thus « *G » *s * wp body *R *s + « *N *G » *s * *Q *s ≤ 1

*by(simp add:negate-embed)

qed

qed

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

have onesided: \( \exists P \text{ c. c} \neq 0 \rightarrow 0 \leq c \Rightarrow \text{sound } P \Rightarrow \)
\( \lambda a. c \cdot \text{lfp-exp (} \lambda a b. \langle G \rangle b \cdot \text{wp body } a b + \langle N \rangle G \cdot b \cdot P b \rangle a \Rightarrow \)
\( \text{lfp-exp (} \lambda a b. \langle G \rangle b \cdot \text{wp body } a b + \langle N \rangle G \cdot b \cdot (c \cdot P b \rangle) \)

proof –

fix \( P, c\) s expect and \( c::\text{real} \)
assume \( \text{cnz: } c \neq 0 \text{ and } \text{nnic: } 0 \leq c \text{ and } s P \text{: sound } P \)
with \( \text{nnic have } \text{cpso: } 0 < c \text{ by(auto) } \)

hence \( \text{nnic: } 0 \leq \text{inverse } c \text{ by(auto) } \)
show \( \lambda a. c \cdot \text{lfp-exp (} \lambda a b. \langle G \rangle b \cdot \text{wp body } a b + \langle N \rangle G \cdot b \cdot P b \rangle a \Rightarrow \)
\( \text{lfp-exp (} \lambda a b. \langle G \rangle b \cdot \text{wp body } a b + \langle N \rangle G \cdot b \cdot (c \cdot P b \rangle) \)

proof(rule lfp-exp-greatest)

fix \( Q\)’s expect
assume \( s Q \text{: sound } Q \)
and \( \text{fp: } \lambda b. \langle G \rangle b \cdot \text{wp body } Q b + \langle N \rangle G \cdot b \cdot (c \cdot P b \rangle \Rightarrow Q \)
hence \( \land s. \langle G \rangle s \cdot \text{wp body } Q s + \langle N \rangle G \cdot s \cdot (c \cdot P s \rangle \leq Q s \text{ by(auto) } \)
with \( \text{nnic have } \langle G \rangle s \cdot \text{inverse } c \cdot (\langle G \rangle s \cdot \text{wp body } Q s + \langle N \rangle G \cdot s \cdot (c \cdot P s \rangle \leq \)
\( \text{inverse } c \cdot Q s \)
by(auto intro:mult-left-mono)

hence \( \land s. \langle G \rangle s \cdot (\text{inverse } c \cdot \text{wp body } Q s) + (\text{inverse } c \cdot c) \cdot \langle N \rangle G \cdot s \cdot P s \leq \)
\( \text{inverse } c \cdot Q s \)
by(simp add:algebra-simps)

hence \( \land s. \langle G \rangle s \cdot \text{wp body } (\lambda s. \text{inverse } c \cdot Q s) s + \langle N \rangle G \cdot s \cdot P s \leq \)
\( \text{inverse } c \cdot Q s \)
by(simp add:cnz scalingD[OF healthy-scalingD, OF hb sQ nnic])

hence \( \lambda s. \langle G \rangle s \cdot \text{wp body } (\lambda s. \text{inverse } c \cdot Q s) s + \langle N \rangle G \cdot s \cdot P s \Rightarrow \)
\( \lambda s. \text{inverse } c \cdot Q s \text{ by(rule le-funI) } \)

moreover from \( \text{nnic sQ have } \text{sound } (\lambda s. \text{inverse } c \cdot Q s) \)
by(proper intro:sound-intros)

ultimately have \( \text{lfp-exp (} \lambda a b. \langle G \rangle b \cdot \text{wp body } a b + \langle N \rangle G \cdot b \cdot P b \rangle \Rightarrow \)
\( \lambda s. \text{inverse } c \cdot Q s \)
by(rule lfp-exp-lowerbound)

hence \( \land s. \text{lfp-exp (} \lambda a b. \langle G \rangle b \cdot \text{wp body } a b + \langle N \rangle G \cdot b \cdot P b \rangle s \leq \)
\( \text{inverse } c \cdot Q s \)
by(rule le-funD)

with \( \text{nnic have } \land s. ... s = Q s \text{ by(simp) } \)
finally show \( \lambda a. c \cdot \text{lfp-exp (} \lambda a b. \langle G \rangle b \cdot \text{wp body } a b + \langle N \rangle G \cdot b \cdot P b \rangle a \Rightarrow \)
by(rule le-funI)

next

from \( s P \) have \( \text{sound (} \lambda s. \text{bound-of } P \) by(auto)
with \( \text{hb sP have } \text{sound (} \text{lfp-exp (} ?X P \rangle) } \)
by(blast intro:lfp-exp-sound lfp-loop-fp)
4.2. HEALTHINESS

with nnc show sound (\(\lambda s. \ c * \text{lfp-exp}(?X P)\) s)
  by(auto intro!:sound-intros)

from hb sP nnc
show \(\lambda s. \ «G» s * \text{wp body}(\lambda s. \text{bound-of} (\lambda s. c * P s))\) s +
  \(\«N\ G» s * (c * P s)\) \(\not\vdash \lambda s. \text{bound-of} (\lambda s. c * P s)\)
  by(auto intro!:lfp-loop-fp sound-intros)

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

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

from nnc have nzc: inverse \(c \neq 0\) by(auto)
moreover with nnc have nnic: \(0 \leq \text{inverse} c\) by(auto)
moreover from nnc sP have scP: sound (\(\lambda s. c * P s\)) by(auto intro!:sound-intros)
ultimately have \(?X (\lambda s. c * P s) \ (\text{inverse} c)\) \(\vdash \ ?Y (\lambda s. c * P s) \ (\text{inverse} c)\)
  by(rule onecuted)
with nnc have \(\lambda s. c * \ ?X (\lambda s. c * P s) \ (\text{inverse} c)\) \(\vdash \)
  \(\lambda s. c * \ ?Y (\lambda s. c * P s) \ (\text{inverse} c)\)
  by(blast intro:mult-left-mono)
with nze show \(?Y P c \vdash \ ?X P c\) by(simp add:mult.assoc[symmetric])
qed
qed
next
fix \(P::\text{'s expect}\) and \(b::\text{real}\)
assume bP: bounded-by b P and nP: nneg P
show lfp-exp (\(\lambda Q s. \ «G» s * \text{wp body} Q\) s + «N G» s * P s) \(\vdash \lambda s. b\)
proof(intro lfp-exp-lowerbound le-funI)
fix s::'
from bP nP hb have bounded-by b (wp body (\(\lambda s. b\))) by(auto)
  hence wp body (\(\lambda s. b\)) s \leq b by(auto)
moreover from bP have P s \leq b by(auto)
ultimately have «G» s * wp body (\(\lambda s. b\)) s + «N G» s * P s \leq «G» s * b + «N G» s * b
  by(auto intro!:add-mono mult-left-mono)
also have ... = b by(simp add:negate-embed field-simps)
finally show «G» s * wp body (\(\lambda s. b\)) s + «N G» s * P s \leq b .
from bP nP have \(0 \leq b\) by(auto)
  thus sound (\(\lambda s. b\)) by(auto)
qed
from hb bP nP show \(\lambda s. 0 \vdash \text{lfp-exp} (\lambda Q s. \ «G» s * \text{wp body} Q\) s + «N G»)
s * P s
by(auto dest!:sound-nneg intro!:lfp-loop-greatest)

next
fix P Q::′s expect
assume sP: sound P and sQ: sound Q and le: P ⊢ Q
show lfp-exp (?X P) ⊢ lfp-exp (?X Q)
proof(rule lfp-exp-greatest)
fix R::′s expect
assume sR: sound R
and fp: λs. «G» s * wp body R s + «N G» s * Q s ⊢ R
from le have λs. «G» s * wp body R s + «N G» s * P s ⊢
   λs. «G» s * wp body R s + «N G» s * Q s
   by(auto intro:le-fun1 add-left-mono mult-left-mono)
also note fp
finally show lfp-exp (λR s. «G» s * wp body R s + «N G» s * P s) ⊢ R
   using sR by(auto intro:lfp-exp-lowerbound)
next
from hb sP show sound (lfp-exp (λR s. «G» s * wp body R s + «N G» s * P s))
   by(rule lfp-loop-sound)
from hb sQ show λs. «G» s * wp body (λs. bound-of Q) s + «N G» s * Q
   s ⊢ λs. bound-of Q
   by(rule lfp-loop-fp)
from sQ show sound (λs. bound-of Q) by(auto)
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.
4.3. CONTINUITY

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

**definition** `bd-cts :: 's trans ⇒ bool`

**where** `bd-cts t = (∀ M. (∀ i. M i ⊢ M (Suc i)) ∧ sound (M i)) →
(∃ b. ∀ i. bounded-by b (M i)) →
t (Sup-exp (range M)) = Sup-exp (range (t o M)))`

**lemma** `bd-ctsD`:

```
[ bd-cts t; ∀ i. M i ⊢ M (Suc i); ∀ i. sound (M i); ∀ i. bounded-by b (M i) ] →
t (Sup-exp (range M)) = Sup-exp (range (t o M))
```

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

**lemma** `bd-ctsI`:

```
(∀ M. (∀ i. M i ⊢ M (Suc i)) → (∀ i. sound (M i)) → (∀ i. bounded-by b (M i)) →
t (Sup-exp (range M)) = Sup-exp (range (t o M))) → 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 = (∀ M. (∀ i. le-trans (M i) (M Succ i) ∧ feasible (M i)) →
equiv-trans (T (Sup-trans (M ′ UNIV))) (Sup-trans ((T o M) ′ UNIV)))`

**lemma** `bd-cts-trD`:

```
[ bd-cts-tr T; ∀ i. le-trans (M i) (M (Suc i)); ∀ i. feasible (M i) ] →
equiv-trans (T (Sup-trans (M ′ UNIV))) (Sup-trans ((T o M) ′ UNIV))
```

**by(simp add:bd-cts-tr-def)**

**lemma** `bd-cts-trI`:

```
∀ M. (∀ i. le-trans (M i) (M (Suc i))) → (∀ i. feasible (M i)) →
equiv-trans (T (Sup-trans (M ′ UNIV))) (Sup-trans ((T o M) ′ UNIV))
```

**implies** `bd-cts-tr T`

**by(simp add:bd-cts-tr-def)**

---

4.3.1 Continuity of Primitives

**lemma** `cts-wp-Abort`:

```
bd-cts (wp (Abort::′s prog))
```

**proof**

```
have X: range (λ i::nat (s::′s). 0) = {λ 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: \( \{P \; | \; P \in \text{range} \; M\} \) by(auto)
show ?thesis by(intro bd-ctsI ext, simp add:wp-eval o-def Sup-exp-def X)
qed

lemma cts-wp-Bind:
fixes a::'a ⇒ 's prog
assumes ca: \( \land s. \) bd-cts (wp (a (f s)))
shows bd-cts (wp (Bind f a))
proof (rule bd-ctsI)
fix M::nat ⇒ 's expect and c::real
assume chain: \( \land i. \) M i ⊨ M (Suc i) and sM: \( \land i. \) sound (M i)
and bM: \( \land i. \) bounded-by c (M i)
with bd-ctsD[OF ca]
have \( \land s. \) wp (a (f s)) (Sup-exp (range M)) = Sup-exp (range (wp (a (f s)) ◦ M))
by(auto)
moreover have \( \land s. \) \{fa s | fa ∈ range (λx. wp (a (f s)) (M x))\} = \{fa s | fa ∈ range (λx s. 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) ◦ 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 ⇒ 's expect and c::real
assume chain: \( \land i. \) M i ⊨ M (Suc i) and sM: \( \land i. \) sound (M i)
and bM: \( \land i. \) bounded-by c (M i)
from ha hb have hab: healthy (wp (a ⋃ b)) by(rule healthy-intros)
from bM have leSup: \( \land i. \) M i ⊨ 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) ◦ M)) ⊨ wp (a ⋃ b) (Sup-exp (range M))
proof (rule Sup-exp-least, clarsimp, rule le-funI)
fix i s
4.3. CONTINUITY

from mono-transD[OF healthy-monoD[OF healthy-monoD[OF hab]]] leSup sM sSup
have wp (a ∩ b) (M i) ⪰ wp (a ∩ b) (Sup-exp (range M)) by(auto)
thus wp (a ∩ b) (M i) ≤ wp (a ∩ b) (Sup-exp (range M)) s by(auto)

from hab sSup have sound (wp (a ∩ b) (Sup-exp (range M))) by(auto)
thus nneg (wp (a ∩ b) (Sup-exp (range M))) by(auto)
qed

from sM bM ha have ∀i. bounded-by c (wp a (M i)) by(auto)
  hence baM: ∀i s. wp a (M i) ≤ c by(auto)
from sM bM hb have ∀i. bounded-by c (wp b (M i)) by(auto)
  hence bbM: ∀i s. wp b (M i) ≤ c by(auto)

show wp (a ∩ b) (Sup-exp (range M)) ⪰ Sup-exp (range (wp (a ∩ b) o M))
proof(simp add:wp-eval o-def, rule le-funD)
  fix s:’s
from bd-ctsD[OF ca, of M, OF chain sM bM] bd-ctsD[OF cb, of M, OF chain sM bM]
  have min (wp (a ∩ b) (Sup-exp (range M))) (wp (a ∩ b) o M)) s =
      min (Sup-exp (range (wp a o M))) (Sup-exp (range (wp b o M))) s
    by(simp)
  also { have {f s | f ∈ range (λx. wp a (M x))}) = range (λi. wp a (M i) s)
          {f s | f ∈ range (λx. wp b (M x))}) = range (λi. wp b (M i) s)
        by(auto)
    hence min (Sup-exp (range (wp a o M))) (Sup-exp (range (wp b o M))) s =
          min (Sup (range (λi. wp a (M i) s))) (Sup (range (λi. wp b (M i) s)))
        by(simp add:Sup-exp-def o-def)
  }
also { have (λi. wp a (M i) s) ⟷ Sup (range (λi. wp a (M i) s))
proof(rule increasing-LIMSEQ)
  fix n
from mono-transD[OF healthy-monoD, OF ha] sM chain
  show wp a (M n) ≤ wp a (M (Suc n)) s by(auto intro:le-funD)
from baM show wp a (M n) ≤ Sup (range (λi. wp a (M i) s))
  by(intro cSup-upper bdd-aboveI, auto)

  fix c:real assume pe: 0 < e
from baM have cSup: Sup (range (λi. wp a (M i) s)) ∈ closure (range (λi. wp a (M i) s))
  by(blast intro:closure-contains-Sup)
  with pe obtain y where yin: y ∈ (range (λi. wp a (M i) s))
      and dy: dist y (Sup (range (λi. wp a (M i) s))) < e
    by(blast dest:iffD[OF closure-approachable])
  from yin obtain i where y = wp a (M i) s by(auto)
  with dy have dist (wp a (M i) s) (Sup (range (λi. wp a (M i) s))) < e
    by(simp)
moreover from \( \text{baM have \ wp \ a \ (M \ i) \ s \leq \ Sup \ (range \ (\lambda i. \ wp \ a \ (M \ i) \ s))} \)
by(intro cSup-upper bdd-aboveI, auto)
ultimately have \( \text{Sup \ (range \ (\lambda i. \ wp \ a \ (M \ i) \ s)) \leq \ wp \ a \ (M \ i) \ s + e} \)
by(simp add:dist-real-def)
thus \( \exists i. \ Sup \ (range \ (\lambda i. \ wp \ a \ (M \ i) \ s)) \leq \ wp \ a \ (M \ i) \ s + e \) by(auto)
qed

moreover

have \( (\lambda i. \ wp \ b \ (M \ i) \ s) \longrightarrow \ Sup \ (range \ (\lambda i. \ wp \ b \ (M \ i) \ s)) \)
proof(rule increasing-LIMSEQ)

fix \( n \)
from mono-transD[OF healthy-monoD, OF hb] sM chain
show \( \wp \ b \ (M \ n) \ s \leq \ wp \ b \ (M \ (Suc \ n)) \) s by(auto intro:le-fanD)
from bbM show \( \wp \ b \ (M \ n) \ s \leq \ Sup \ (range \ (\lambda i. \ wp \ b \ (M \ i) \ s)) \)
by(intro cSup-upper bdd-aboveI, auto)

fix \( c :: \text{real} \) assume \( \text{pc: \( 0 < e \)} \)
from bbM have \( \text{cSup:} \ Sup \ (range \ (\lambda i. \ wp \ b \ (M \ i) \ s)) \in \text{closure} \ (range \ (\lambda i. \ wp \ b \ (M \ i) \ s)) \)
by(blast intro:closure-contains-Sup)
with \( \text{pe obtain \( y \ where \ yin: \ y \in \ (range \ (\lambda i. \ wp \ b \ (M \ i) \ s)) \) \) \)
and \( \text{dy:} \ dist \ y \ (Sup \ (range \ (\lambda i. \ wp \ b \ (M \ i) \ s))) < e \)
by(blast dest:iffD[OF closure-approachable])
from \( yin \) obtain \( i \ where \ y = wp \ b \ (M \ i) \ s \) by(auto)
with \( \text{dy have \} dist \ wp \ b \ (M \ i) \ s \ (Sup \ (range \ (\lambda i. \ wp \ b \ (M \ i) \ s))) < e \) \)
by(simp)
moreover from bbM have \( \wp \ b \ (M \ i) \ s \leq \ Sup \ (range \ (\lambda i. \ wp \ b \ (M \ i) \ s)) \)
by(intro cSup-upper bdd-aboveI, auto)
ultimately have \( \text{Sup \ (range \ (\lambda i. \ wp \ b \ (M \ i) \ s)) \leq \ wp \ b \ (M \ i) \ s + e} \)
by(simp add:dist-real-def)
thus \( \exists i. \ Sup \ (range \ (\lambda i. \ wp \ b \ (M \ i) \ s)) \leq \ wp \ b \ (M \ i) \ s + e \) by(auto)
qed

ultimately have \( (\lambda i. \ min \ (wp \ a \ (M \ i) \ s) \ (wp \ b \ (M \ i) \ s)) \longrightarrow \ min \ (Sup \ (range \ (\lambda i. \ wp \ a \ (M \ i) \ s))) \ (Sup \ (range \ (\lambda i. \ wp \ b \ (M \ i) \ s))) \)
by(rule tendsto-min)

moreover have \( \text{bdd-above \ (range \ (\lambda i. \ min \ (wp \ a \ (M \ i) \ s) \ (wp \ b \ (M \ i) \ s)))} \)
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 \( \text{ha sM bM have \ bounded-by \ c \ (wp \ a \ (M \ i))} \) by(auto)

hence \( \text{wp \ a \ (M \ i) \ s \leq \ c} \) by(auto)
}
finally show \( \text{min \ (wp \ a \ (M \ i) \ s) \ (wp \ b \ (M \ i) \ s) \leq \ c} \).
qed

ultimately

have \( \text{min \ (Sup \ (range \ (\lambda i. \ wp \ a \ (M \ i) \ s))) \ (Sup \ (range \ (\lambda i. \ wp \ b \ (M \ i) \ s)))} \)
\leq 
\( \text{Sup \ (range \ (\lambda i. \ min \ (wp \ a \ (M \ i) \ s) \ (wp \ b \ (M \ i) \ s)))} \)
4.3. CONTINUITY

by (blast intro: LIMSEQ_le_const2 cSup_upper min_mono [OF baM bbM])
}
also {
  have range (λi. min (wp a (M i) s) (wp b (M i) s)) =
    { f : s | f ∈ range (λi. 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 (range (λi. min (wp a (M i) s) (wp b (M i) s)))
    by (simp add: SUP_cong simp)
}
finally show min (wp a (Sup-exp (range M) s) (wp b (Sup-exp (range M) s)) ≤
    Sup-exp (range (λi. 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)
\textbf{CHAPTER 4. THE PGCL LANGUAGE}

\begin{quote}
and \( ha: \text{healthy} (wp a) \)
and \( bb: \text{healthy} (wp b) \)
and \( wp: \text{unitary} p \)
shows \( \text{bd-cts} (wp (\text{PC a p b})) \)
\end{quote}

\textbf{proof} (rule \text{bd-ctsI, rule ext, simp add:o-def wp-eval})

fix \( M::\text{nat} \Rightarrow 's \text{ expect} \ \text{and} \ c::\text{real} \ \text{and} \ s:'s \)
assume chain: \( \bigwedge. M i \vdash M (\text{Suc i}) \ \text{and} \ sM: \bigwedge. \text{sound} (M i) \)
and \( bM: \bigwedge. \text{bounded-by c} (M i) \)

from \( sM \) have \( \bigwedge. \text{nnc} (M i) \) \textbf{by(auto)}
with \( bM \) have \( \text{nc: } 0 \leq c \) \textbf{by(auto)}

from chain \( sM \) \( bM \) have \( wp a (\text{Sup-exp} (\text{range } M)) = \text{Sup-exp} (\text{range } (wp a o M)) \)
by(rule \text{bd-ctsD}[OF ca])
hence \( wp a (\text{Sup-exp} (\text{range } M)) s = \text{Sup-exp} (\text{range } (wp a o M)) s \)
by(simp)
also \{ 
have \( \{f s \mid f. f \in \text{range } (\lambda x. wp a (M x))\} = \text{range } (\lambda i. wp a (M i) s) \)
by(auto)
hence \( \text{Sup-exp} (\text{range } (wp a o M)) s = \text{Sup} (\text{range } (\lambda i. wp a (M i) s)) \)
by(simp add:Sup-exp-def o-def)
\}
finally have \( p s * wp a (\text{Sup-exp} (\text{range } M)) s = \)
\( p s * \text{Sup} (\text{range } (\lambda i. wp a (M i) s)) \) \textbf{by(simp)}
also have \( ... = \text{Sup} \{p s * x \mid x. x \in \text{range } (\lambda i. wp a (M i) s)\} \)
proof (rule \text{cSup-mult}, blast, clarsimp)
from wp show \( \text{0} \leq p s \) \textbf{by(auto)}
fix \( i \)
from \( sM \) \( bM \) \( ha \) have \( \text{bounded-by c} (wp a (M i)) \) \textbf{by(auto)}
thus \( wp a (M i) s \leq c \) \textbf{by(auto)}
qed
also \{ 
have \( \{p s * x \mid x. x \in \text{range } (\lambda i. wp a (M i) s)\} = \text{range } (\lambda i. p s * wp a (M i) s) \)
by(auto)
hence \( \text{Sup} \{p s * x \mid x. x \in \text{range } (\lambda i. wp a (M i) s)\} = \)
\( \text{Sup} (\text{range } (\lambda i. p s * wp a (M i) s)) \) \textbf{by(simp)}
\}
finally have \( p s * wp a (\text{Sup-exp} (\text{range } M)) s = \text{Sup} (\text{range } (\lambda i. p s * wp a (M i) s)) \).
moreover \{ 
from chain \( sM \) \( bM \) have \( wp b (\text{Sup-exp} (\text{range } M)) = \text{Sup-exp} (\text{range } (wp b o M)) \)
by(rule \text{bd-ctsD}[OF cb])
hence \( wp b (\text{Sup-exp} (\text{range } M)) s = \text{Sup-exp} (\text{range } (wp b o M)) s \)
by(simp)
also \{ 
have \( \{f s \mid f. f \in \text{range } (\lambda x. wp b (M x))\} = \text{range } (\lambda i. wp b (M i) s) \)
\end{quote}
4.3. CONTINUITY

\[
\text{by (auto)}
\]
\[
\text{hence Sup-exp (range (wp b o M)) s = Sup (range (λi. wp b (M i) s))}
\]
\[
\text{by (simp add: Sup-exp-def a-def)}
\]

} finally have \((1 - p s) * wp b (\text{Sup-exp (range } M)) s =
\]
\[
(1 - p s) * \text{Sup (range (λi. wp b (M i) s)) by (simp)}
\]
also have \(\ldots = \text{Sup } \{((1 - p s) * x | x \in \text{range (λi. wp b (M i) s)}\}
\]
proof (rule cSup-mult, blast, clarsimp)
from up show \(0 \leq 1 - p s\)
by auto
fix i
from sM bM hb have bounded-by c (wp b (M i)) by (auto)
thus wp b (M i) s \(\leq c\) by (auto)
qed

also { have \(\{((1 - p s) * x | x \in \text{range (λi. wp b (M i) s)}\} =
\]
\[
\text{range (λi. (1 - p s) * wp b (M i) s)}
\]
by (auto)
\[
\text{hence Sup } \{((1 - p s) * x | x \in \text{range (λi. wp b (M i) s)}\} =
\]
\[
\text{Sup (range (λi. (1 - p s) * wp b (M i) s)) by (simp)}
\]
} finally have \((1 - p s) * wp b (\text{Sup-exp (range } M)) s =
\]
\[
\text{Sup (range (λi. (1 - p s) * wp b (M i) s))}.
\]

ultimately have \(p s * wp a (\text{Sup-exp (range } M)) s + (1 - p s) * wp b (\text{Sup-exp (range } M)) s =
\]
\[
\text{Sup (range (λi. p s * wp a (M i) s)) + Sup (range (λi. (1 - p s) * wp b (M i) s))}
\]
by (simp)
also { from bM sM ha have \(\bigwedge i. \text{bounded-by } c (wp a (M i))\) by (auto)

hence \(\bigwedge i. wp a (M i) s \leq c\) by (auto)

moreover from up have \(0 \leq p s\) by (auto)

ultimately have \(\bigwedge i. p s * wp a (M i) s \leq p s * c\) by (auto intro: mult-left-mono)

also from up nc have \(p s * c \leq 1 * c\) by (blast intro: mult-right-mono)

also have \(\ldots = c\) by (simp)

finally have baM: \(\bigwedge i. p s * wp a (M i) s \leq c\).

have lima: \(\lambda i. p s * wp a (M i) s \longrightarrow \text{Sup (range (λi. p s * wp a (M i) s))}\)

proof (rule increasing-LIMSEQ)
fix n
from sM chain healthy-monoD [OF ha] have \(wp a (M n) \vdash wp a (M (Suc n))\)
by (auto)
with up show \(p s * wp a (M n) s \leq p s * wp a (M (Suc n)) s\)
by (blast intro: mult-left-mono)
from baM show \(p s * wp a (M n) s \leq \text{Sup (range (λi. p s * wp a (M i) s))}\)
by (intro cSup-upperbdd-above1, auto)
next
fix e::real
assume pe: 0 < e
from baM have Sup (range (λi. p s * wp a (M i) s)) ∈
closure (range (λi. p s * wp a (M i) s))
by(blast intro:closure-contains-Sup)

thm closure-approachable
with pe obtain y where yin: y ∈ range (λi. p s * wp a (M i) s)
and dy: dist y (Sup (range (λi. p s * wp a (M i) s))) < e
by(blast dest:iffD1[OF closure-approachable])
from yin obtain i where y = p s * wp a (M i) s by(auto)
with dy have dist (p s * wp a (M i) s) (Sup (range (λi. p s * wp a (M i) s))) < e
by(simp)
moreover from baM have p s * wp a (M i) s ≤ Sup (range (λi. p s * wp a (M i) s))
by(intro cSup-upper bdd-aboveI, auto)
ultimately have Sup (range (λi. p s * wp a (M i) s)) ≤ p s * wp a (M i) s + e
by(simp add:dist-real-def)
thus ∃i. Sup (range (λi. p s * wp a (M i) s)) ≤ p s * wp a (M i) s + e
by(auto)

qed

from baM baM have λi. bounded-by c (wp b (M i)) by(auto)

hence λi. wp b (M i) s ≤ c by(auto)
moreover from wp have θ ≤ (1 − p s)
by auto
ultimately have λi. (1 − p s) * wp b (M i) s ≤ (1 − p s) * c
by(auto intro:mult-left-mono)
also {
from wp have 1 − p s ≤ 1 by(auto)
with nc have (1 − p s) * c ≤ 1 * c
by(blast intro:mult-right-mono)
}
also have 1 * c = c by(simp)
finally have bbM: λi. (1 − p s) * wp b (M i) s ≤ c

have limb: (λi. (1 − p s) * wp b (M i) s) ----→ Sup (range (λi. (1 − p s) * wp b (M i) s))
proof(rule increasing-LIMSEQ)
fix n
from sM chain healthy-monoD[OF hb] have wp b (M n) ⊢ wp b (M (Suc n))
by(auto)
moreover from wp have θ ≤ 1 − p s
by auto
ultimately show (1 − p s) * wp b (M n) s ≤ (1 − p s) * wp b (M (Suc n))
by(blast intro:mult-left-mono)
from bbM show (1 − p s) * wp b (M n) s ≤ Sup (range (λi. (1 − p s) * wp
4.3. CONTINUITY

\[ b \ (M \ i \ s) \]
\[
\text{by} (\text{intro} \ c\text{Sup-upper} \ \text{bdd-above}I, \ \text{auto})
\]
\text{next}
\text{fix} \ e::\text{real}
\text{assume} \ pe: \ 0 < e
\text{from} \ bbM \ \text{have} \ \sup (\text{range} \ (\lambda i. \ (1 - p \ s) * \text{wp} \ b \ (M \ i \ s)) \in
\text{closure} \ (\text{range} \ (\lambda i. \ (1 - p \ s) * \text{wp} \ b \ (M \ i \ s)))
\text{by} (\text{blast} \ \text{intro:}\text{closure-contains-Sup})
\text{with} \ pe \ \text{obtain} \ y \ \text{where} \ yin: \ y \in \text{range} \ (\lambda i. \ (1 - p \ s) * \text{wp} \ b \ (M \ i \ s))
\text{and} \ dy: \ \text{dist} \ y \ (\sup (\text{range} \ (\lambda i. \ (1 - p \ s) * \text{wp} \ b \ (M \ i \ s)))) < e
\text{by} (\text{blast} \ \text{dest:}\text{iffD1}[\text{OF closure-approachable}])
\text{from} \ yin \ \text{obtain} \ i \ \text{where} \ y = (1 - p \ s) * \text{wp} \ b \ (M \ i \ s) \ \text{by} (\text{auto})
\text{with} \ dy \ \text{have} \ \text{dist} \ ((1 - p \ s) * \text{wp} \ b \ (M \ i \ s)) < e
\text{by} (\text{simp})
\text{moreover from} \ bbM
\text{have} \ (1 - p \ s) * \text{wp} \ b \ (M \ i \ s) \leq \sup (\text{range} \ (\lambda i. \ (1 - p \ s) * \text{wp} \ b \ (M \ i \ s)))
\text{by} (\text{intro} \ c\text{Sup-upper} \ \text{bdd-above}I, \ \text{auto})
\text{ultimately have} \ \sup (\text{range} \ (\lambda i. \ (1 - p \ s) * \text{wp} \ b \ (M \ i \ s))) \leq (1 - p \ s) * \text{wp} \ b \ (M \ i \ s) + e
\text{by} (\text{simp add:dist-real-def})
\text{thus} \ \exists i. \ \sup (\text{range} \ (\lambda i. \ (1 - p \ s) * \text{wp} \ b \ (M \ i \ s))) \leq (1 - p \ s) * \text{wp} \ b \ (M \ i \ s)
\text{by} (\text{auto})
\text{qed}

\text{from} \ \text{lina} \ \text{limb} \ \text{have} \ (\lambda i. \ p \ s * \text{wp} \ a \ (M \ i \ s) + (1 - p \ s) * \text{wp} \ b \ (M \ i \ s))$
\[ \sup (\text{range} \ (\lambda i. \ p \ s * \text{wp} \ a \ (M \ i \ s) + (1 - p \ s) * \text{wp} \ b \ (M \ i \ s)))
\text{by} (\text{rule tendsto-add})
\text{moreover from} \ \text{add-mono}[\text{OF baM} \ bbM]
\text{have} \ (\lambda i. \ p \ s * \text{wp} \ a \ (M \ i \ s) + (1 - p \ s) * \text{wp} \ b \ (M \ i \ s) \leq
\sup (\text{range} \ (\lambda i. \ p \ s * \text{wp} \ a \ (M \ i \ s) + (1 - p \ s) * \text{wp} \ b \ (M \ i \ s)))
\text{by} (\text{intro} \ c\text{Sup-upper} \ \text{bdd-above}I, \ \text{auto})
\text{ultimately have} \ \sup (\text{range} \ (\lambda i. \ p \ s * \text{wp} \ a \ (M \ i \ s)) +
\sup (\text{range} \ (\lambda i. \ (1 - p \ s) * \text{wp} \ b \ (M \ i \ s))) \leq
\sup (\text{range} \ (\lambda i. \ p \ s * \text{wp} \ a \ (M \ i \ s) + (1 - p \ s) * \text{wp} \ b \ (M \ i \ s)))
\text{by} (\text{blast} \ \text{intro:}\text{LIMSEQ-le-const}2)
\}
\text{also}
\{ \text{have} \ \text{range} \ (\lambda i. \ p \ s * \text{wp} \ a \ (M \ i \ s) + (1 - p \ s) * \text{wp} \ b \ (M \ i \ s) =
\{f s. f \in \text{range} \ (\lambda x s. \ p s * \text{wp} \ a \ (M x) s + (1 - p s) * \text{wp} \ b \ (M x) s}\}
\text{by} (\text{auto})
\text{hence} \ \sup (\text{range} \ (\lambda i. \ p \ s * \text{wp} \ a \ (M \ i \ s) + (1 - p \ s) * \text{wp} \ b \ (M \ i \ s))) =
\sup (\text{range} \ (\lambda x s. \ p s * \text{wp} \ a \ (M x) s + (1 - p s) * \text{wp} \ b \ (M x) s)) \ s
\text{by} (\text{simp add:}\text{Sup-exp-def cong del:}\text{SUP-cong-simp})
\}
\text{finally}
\text{have} \ p \ s * \text{wp} \ a \ (\text{Sup-exp} \ (\text{range} \ M)) s + (1 - p s) * \text{wp} \ b \ (\text{Sup-exp} \ (\text{range} \ M))
proofs for both are inductive, and rely on the above results on binary operation.

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

lemma SetPC-Bind:
  \[ \text{SetPC} \ a \ p = \text{Bind} \ p \ (\lambda p. \text{SetPC} \ a \ (\lambda x. p)) \]
  by (intro ext, simp add: SetPC-def Bind-def Let-def)

lemma SetPC-remove:
  assumes nz: \( p \ x \neq 0 \) and n1: \( p \ x \neq 1 \)
  and fsupp: finite (supp p)
  shows \( \text{SetPC} \ a \ (\lambda x. p) \ (\text{SetPC} \ a \ (\lambda x. \text{dist-remove} \ p x)) \)
  proof (intro ext, simp add: SetPC-def PC-def)
    fix \( a \ b \ P \ s \)
    from nz have \( x \in \text{supp} \ p \) by (simp add: supp-def)
    hence \( \text{supp} \ p = \text{insert} \ x \ (\text{supp} \ p - \{x\}) \)
      by (auto)
    hence \( (\sum x \in \text{supp} \ p. \ p x * a x ab P s) = \)
      \( (\sum x \in \text{insert} \ x \ (\text{supp} \ p - \{x\}). \ p x * a x ab P s) \)
      by (simp)
    also from fsupp have ...
      \( (\sum x \in \text{supp} \ p. \ p x * a x ab P s) = \)
      \( (\sum y \in \text{supp} \ p - \{x\}. \ p y / (1 - p x) * a y ab P s) \)
      by (simp add: field-simps)
    also have ...
      \( (\sum x \in \text{supp} \ p. \ p x * a x ab P s) = \)
      \( (\sum y \in \text{supp} \ p - \{x\}. \ \text{dist-remove} \ p x y * a y ab P s) \)
      by (simp add: dist-remove-def)
    also from nz n1 have ...
      \( (\sum x \in \text{supp} \ p. \ p x * a x ab P s) = \)
      \( (\sum y \in \text{supp} \ (\text{dist-remove} \ p x). \ \text{dist-remove} \ p x y * a y ab P s) \)
      by (simp add: supp-dist-remove)
  finally show \( (\sum x \in \text{supp} \ p. \ p x * a x ab P s) = \)
    \( (\sum y \in \text{supp} \ (\text{dist-remove} \ p x). \ \text{dist-remove} \ p x y * a y ab P s) \).
  qed

lemma cts-bot:
  bd-cts \( (\lambda (P::'s) expect) \ (s::'s). \ 0::real) \)
  proof
    have \( X: \lambda s::'s. \{ (P::'s) expect | P. P \in \text{range} \ (\lambda P s. 0) \} = \{0\} \) by (auto)
    show \( \text{thesis} \) by (intro bd-ctsI, simp add: Sup-exp-def o-def X)
  qed

lemma wp-SetPC-nil:
  \( \text{wp} \ (\text{SetPC} \ a \ (\lambda s a. 0)) = (\lambda P s. 0) \)
by(intro ext, simp add:wp-eval)

lemma SetPC-sgl:
  \( \text{supp } p = \{ x \} \implies \text{SetPC } a \ (\lambda x. \ p) = (\lambda \text{ab } P \ s \ p \ x \ + \ a \ \text{ab } P \ s) \)
  by(simp add:SetPC-def)

lemma bd-cts-scale:
  fixes a::'s trans
  assumes ca: bd-cts a
  and ha: healthy a
  and nnc: \( 0 \leq c \)
  shows bd-cts (\( \lambda P \ s \ c \ * \ a \ P \ s \))
proof(intro bd-ctsI ext, simp add:o-def)
  fix M::nat \Rightarrow 's expect and d::real and s::'
  assume chain: \( \bigwedge i. \ M \ i \vdash M \ (Suc \ i) \) and sM: \( \bigwedge i. \ \text{sound } (M \ i) \)
  and bm: \( \bigwedge i. \ \text{bounded-by } d \ (M \ i) \)
from sM have \( \bigwedge i. \ \text{nneg } (M \ i) \) 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-scaling[OF ha] nnc
have c * a (Sup-exp (range M)) s = a (\( \lambda s. c \ * \ Sup-exp \ (range M) \ s \)) s
  by(auto intro:scalingD)
also { have \( \bigwedge s. \ \{ f \ s \ | f. f \in \text{range } M \} = \text{range } (\lambda i. \ M \ i \ s) \) by(auto)
  hence a (\( \lambda s. c \ * \ Sup-exp \ (range M) \ s \)) s =
    a (\( \lambda s. c \ * \ Sup \ (\lambda i. \ M \ i \ s()) \)) s
    by(simp add:Sup-exp-def)
}
also { from bm have \( \bigwedge x. s. x \in \text{range } (\lambda i. \ M \ i \ s) \implies x \leq d \) by(auto)
  with nnc have a (\( \lambda s. c \ * \ Sup \ (\lambda i. \ M \ i \ s()) \)) s =
    a (\( \lambda s. Sup \ {c*x \mid x \in \text{range } (\lambda i. \ M \ i \ s())} \)) s
    by(subst cSup-mult, blast+)
}
also { have X: \( \bigwedge s. \ \{ c * x \mid x \in \text{range } (\lambda i. \ M \ i \ s()) \} = \text{range } (\lambda i. c * M \ i \ s) \) by(auto)
  have a (\( \lambda s. Sup \ {c * x \mid x \in \text{range } (\lambda i. \ M \ i \ s())} \)) s =
    a (\( \lambda s. Sup \ (\lambda i. c * M \ i \ s()) \)) s by(simp add:X)
}
also { have \( \bigwedge s. \text{range } (\lambda i. c * M \ i \ s) = \{ f \ s \ | f. f \in \text{range } (\lambda i. s. c * M \ i \ s) \} \)
    by(auto)
  hence (\( \lambda s. Sup \ (\lambda i. c * M \ i \ s()) \)) = Sup-exp (\( \lambda i. s. c * M \ i \ s()) \)
    by (simp add: Sup-exp-def cong del: SUP-cong-simp)
  hence a (\( \lambda s. Sup \ (\lambda i. c * M \ i \ s()) \)) s =
    a (Sup-exp (\( \lambda i. s. c * M \ i \ s()) \)) s by(simp)
}
also \{
\begin{align*}
\text{from} & \ \text{le-funD}[\text{OF chain}] \ nnc \\
\text{have} & \ \bigwedge_i \ (\lambda s. \ c \ast M \ i \ s) \vdash (\lambda s. \ c \ast M \ (\text{Suc} \ i) \ s) \\
& \quad \text{by (auto intro:le-funI[OF mult-left-mono])} \\
\text{moreover from} & \ sM \ nnc \\
\text{have} & \ \bigwedge_i \ \text{sound} \ (\lambda s. \ c \ast M \ i \ s) \\
& \quad \text{by (auto intro:sound-intros)} \\
\text{moreover from} & \ bM \ nnc \\
\text{have} & \ \bigwedge_i \ \text{bounded-by} \ (c \ast d) \ (\lambda s. \ c \ast M \ i \ s) \\
& \quad \text{by (auto intro:mult-left-mono)} \\
\text{ultimately} \\
\text{have} & \ a \ (\text{Sup-exp} \ (\text{range} \ (\lambda i s. \ c \ast M \ i \ s))) = \\
& \quad \text{Sup-exp} \ (\text{range} \ (a \ o \ (\lambda i s. \ c \ast M \ i \ s))) \\
& \quad \text{by (rule bd-ctsD[OF ca])} \\
\text{hence} & \ a \ (\text{Sup-exp} \ (\text{range} \ (\lambda i s. \ c \ast M \ i \ s))) \ s = \\
& \quad \text{Sup-exp} \ (\text{range} \ (a \ o \ (\lambda i s. \ c \ast M \ i \ s)) \ s) \\
& \quad \text{by (auto)} \\
\} \\
\text{also have} \ \text{Sup-exp} \ (\text{range} \ (a \ o \ (\lambda i s. \ c \ast M \ i \ s))) \ s = \\
& \quad \text{Sup-exp} \ (\text{range} \ (\lambda x. \ a \ (\lambda s. \ c \ast M \ x \ s))) \ s \\
& \quad \text{by (simp add:o-def)} \\
\} \\
\text{also \{} \\
\text{from} & \ nnc \ sM \\
\text{have} & \ \bigwedge x. \ a \ (\lambda s. \ c \ast M \ x \ s) = (\lambda s. \ c \ast a \ (M \ x) \ s) \\
& \quad \text{by (auto intro:scalingD[OF healthy-scalingD, OF ha, symmetric])} \\
\text{hence} & \ \text{Sup-exp} \ (\text{range} \ (\lambda x. \ a \ (\lambda s. \ c \ast M \ x \ s))) \ s = \\
& \quad \text{Sup-exp} \ (\text{range} \ (\lambda x s. \ c \ast a \ (M \ x) \ s)) \ s \\
& \quad \text{by (simp)} \\
\} \\
\text{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 \\
& \quad \text{by (auto)} \\
\} \\
\text{qed}
\end{align*}

\textbf{lemma} \ cts-wp-SetPC-const:
\textbf{fixes} \ a :: 'a \Rightarrow 's \ prog
\textbf{assumes} \ ca: \ \bigwedge x. \ x \in (\text{supp} \ p) \Longrightarrow \text{bd-cts} \ (wp \ (a \ x))
\textbf{and} \ ha: \ \bigwedge x. \ x \in (\text{supp} \ p) \Longrightarrow \text{healthy} \ (wp \ (a \ x))
\textbf{and} \ wp: \ \text{unitary} \ p
\textbf{and} \ \text{smp}: \ \text{sum} \ p \ (\text{supp} \ p) \leq 1
\textbf{and} \ \text{fsupp}: \ \text{finite} \ (\text{supp} \ p)
\textbf{shows} \ \text{bd-cts} \ (wp \ (\text{SetPC} \ a \ (\lambda - \ . \ p)))
\textbf{proof} (cases \ \text{supp} \ p = \ {}, \ \text{simp add:supp-empty SetPC-def wp-def cts-bot})
\textbf{assume} \ \text{nesupp}: \ \text{supp} \ p \ \neq \ \{} \\
\textbf{from} \ \text{fsupp} \ \text{have} \ \text{unitary} \ p \ \Longrightarrow \ \text{sum} \ p \ (\text{supp} \ p) \leq 1 \ \Longrightarrow \\
(\forall \ x \in \text{supp} \ p. \ \text{bd-cts} \ (wp \ (a \ x))) \ \Longrightarrow \\
(\forall \ x \in \text{supp} \ p. \ \text{healthy} \ (wp \ (a \ x))) \ \Longrightarrow \\
\text{bd-cts} \ (wp \ (\text{SetPC} \ a \ (\lambda - \ . \ p)))
\textbf{proof} (induct \ \text{supp} \ p \ \text{arbitrary}: p, \ \text{simp add:supp-empty wp-SetPC-nil cts-bot, clarify})
fix \( x::a \) and \( F::a \) set and \( p::a \Rightarrow \text{real} \)
assume \( F::\text{finite} \)
assume \( \text{insert } x F = \text{supp } p \)
hence \( \text{pstep}::\text{supp } p = \text{insert } x F \) by(simp)

hence \( x: \in \text{supp } p \) by(auto)
assume \( \text{up}:\text{unitary } p \) and \( \text{ca}: \forall x: \in \text{supp } p. \ \text{bd-cts} (wp (a x)) \)
and \( \text{ha}: \forall x: \in \text{supp } p. \ \text{healthy} (wp (a x)) \)
and \( \text{sum}: \ \text{sum } p (\text{supp } p) \leq 1 \)
and \( x:\notin F \)
assume \( \text{IH}: \bigwedge p. F = \text{supp } p \Rightarrow \text{unitary } p \Rightarrow \text{sum } p (\text{supp } p) \leq 1 \Rightarrow \)
\( (\forall x: \in \text{supp } p. \ \text{bd-cts} (wp (a x))) \Rightarrow \)
\( (\forall x: \in \text{supp } p. \ \text{healthy} (wp (a x))) \Rightarrow \)
\( \text{bd-cts} (wp (\text{SetPC } a (\lambda. p))) \)

from \( F::\text{pstep} \) have \( \text{fsupp}: \text{finite} (\text{supp } p) \) by(auto)

from \( x: \in \) have \( nzp: p x \neq 0 \) by(simp add:supp-def)

have \( \bigwedge y. y: \in \text{supp } p \Rightarrow y \neq x \Rightarrow p x + p y \leq \text{sum } p (\text{supp } p) \)
proof (rule contra)
  fix \( y \) assume \( y: \in \text{supp } p \) and \( y:\neq x \)
  from \( \text{up} \) have \( 0 \leq \text{sum } p (\text{supp } p - \{x,y\}) \)
    by(auto intro:sum-nonneg)
  hence \( p x + p y \leq p x + p y + \text{sum } p (\text{supp } p - \{x,y\}) \)
    by(auto)

  also {
    from \( y: \in \) \( y:\neq \) have \( \text{fsupp} \)
    have \( p y + \text{sum } p (\text{supp } p - \{x,y\}) = \text{sum } p (\text{supp } p - \{x\}) \)
      by(subst \text{sum.insert[symmetric]}, (blast intro!:sum.cong)+)
    moreover
    from \( x: \in \text{supp} \) have \( p x + \text{sum } p (\text{supp } p - \{x\}) = \text{sum } p (\text{supp } p) \)
      by(subst \text{sum.insert[symmetric]}, (blast intro!:sum.cong)+)
    ultimately
    have \( p x + p y + \text{sum } p (\text{supp } p - \{x, y\}) = \text{sum } p (\text{supp } p) \) by(simp)
  }

finally show \( p x + p y \leq \text{sum } p (\text{supp } p) \).
qed
also from yin yne have \( p \cdot x + p \cdot y \leq \text{sum} \ (\text{ supp } \ p) \)
by (rule \( \text{ xy-le-sum} \))

finally show False using \( \text{ sump by(simp) } \)

qed

show \( \text{ bd-cts } (wp \ (\text{SetPC} \ a \ (\lambda -. \ p))) \)

proof (cases \( F = \{ \} \))

case True with \( \text{ pstep } \) have \( \text{ supp } = \{ x \} \) by (simp)

hence \( \text{ wp } \ (\text{SetPC} \ a \ (\lambda -. \ p)) = (\lambda P \ s. \ p \cdot x * \ wp \ (a \ x) \ P \ s) \)

by (simp add: \( \text{ SetPC-sgl wp-def} \))

moreover {
from up ca ha xin have \( \text{ bd-cts } \ (wp \ (a \ x)) \) healthy \( \ (wp \ (a \ x)) \) \( 0 \leq p \ x \)

by (auto)

hence \( \text{ bd-cts } (\lambda P \ s. \ p \cdot x * \ wp \ (a \ x) \ P \ s) \)

by (rule \( \text{ bd-cts-scale} \))
}

ultimately show \( \ ? \text{ thesis } \) by (simp)

next

assume neF: \( F \neq \{ \} \)

then obtain \( y \) where \( \text{ yinF: } y \in F \) by (auto)

with xni have yne: \( y \neq x \) by (auto)

from yinF pstep have yin: \( y \in \text{ supp } \ p \) by (auto)

from supp-dist-remove[of \( p \ x, \ OF \ n z p \ n 1 p, \ OF \ yin \ yne \)]

have supp-sub: \( \text{ supp } \ (\text{ dist-remove } p \ x) \subseteq \text{ supp } \ p \) by (auto)

from xin ca have cax: \( \text{ bd-cts } (wp \ (a \ x)) \) by (auto)

from xin ha have hax: \( \text{ healthy } (wp \ (a \ x)) \) by (auto)

from supp-sub ha have hra: \( \forall x \in \text{ supp } \ (\text{ dist-remove } p \ x). \text{ healthy } (wp \ (a \ x)) \)

by (auto)

from supp-sub ca have cra: \( \forall x \in \text{ supp } \ (\text{ dist-remove } p \ x). \text{ bd-cts } (wp \ (a \ x)) \)

by (auto)

from supp-dist-remove[of \( p \ x, \ OF \ n z p \ n 1 p, \ OF \ yin \ yne \)]

pstep xni

have Fsupp: \( F = \text{ supp } (\text{ dist-remove } p \ x) \)

by (simp)

have udp: \( \text{ unitary } (\text{ dist-remove } p \ x) \)

proof (intro unitaryI2 nnegI bounded-byI)

fix \( y \)

show \( 0 \leq \text{ dist-remove } p \ x \ y \)

proof (cases \( y = x \), simp-all add: dist-remove-def)

from up have \( 0 \leq p \ y \ 0 \leq 1 - p \ x \)

by auto

thus \( 0 \leq p \ y / (1 - p \ x) \)

by (rule divide-nonneg-nonneg)

qed

show \( \text{ dist-remove } p \ x \ y \leq 1 \)
proof (cases y = x, simp-all add: dist-remove-def,
cases y ∈ supp p, simp-all add: nsupp-zero)

assume yne: y ≠ x and yin: y ∈ supp p

hence p x + p y ≤ sum p (supp p)
    by (auto intro: xy-le-sum)

also note sump

finally have p y ≤ 1 − p x by (auto)

moreover from up have p x ≤ 1 by (auto)

moreover from yin yne have p x ≠ 1 by (rule n1p)

ultimately show p y / (1 − p x) ≤ 1 by (auto)

qed

qed

from xin have pxn0: p x ≠ 0 by (auto simp: supp-def)
from yin yne have pxn1: p x ≠ 1 by (rule n1p)

from pxn0 pxn1 have sum (dist-remove p x) (supp (dist-remove p x)) =

    sum (dist-remove p x) (supp p − {x})
    by (simp add: supp-dist-remove)

also have ... = (∑ y ∈ supp p − {x}. p y / (1 − p x))
    by (simp add: dist-remove-def)

also have ... = (∑ y ∈ supp p − {x}. p y) / (1 − p x)
    by (simp add: sum-divide-distrib)

also {
    from xin have insert x (supp p) = supp p by (auto)
    with fsupp have p x + (∑ y ∈ supp p − {x}. p y) = sum p (supp p)
        by (simp add: sum.insert[symmetric])
    also note sump
    finally have sum p (supp p − {x}) ≤ 1 − p x by (auto)
    moreover {
        from up have p x ≤ 1 by (auto)
        with pxn1 have p x < 1 by (auto)
        hence 0 < 1 − p x by (auto)
    }
    ultimately have sum p (supp p − {x}) / (1 − p x) ≤ 1
        by (auto)
}

finally have sdp: sum (dist-remove p x) (supp (dist-remove p x)) ≤ 1.

from Fsupp udp sdp hra cra IH
have cts-dr: bd-cts (wp (SetPC a (λ-. dist-remove p x)))
    by (auto)

from up have upx: unitary (λ-. p x) by (auto)

from pxn0 pxn1 fsupp hra show ?thesis
    by (simp add: SetPC-remove,
          blast intro:cts-wp-PC caz cts-dr haz healthy-intros
          unitary-sound[OF udp] sdp upx)
4.3. CONTINUITY

qed
qed

with assms show ?thesis by(auto)

qed

lemma cts-wp-SetPC:
  fixes a::'a ⇒ 's prog
  assumes ca: "∀ x s. x ∈ (supp (p s)) ⇒ bd-cts (wp (a x))
  and ha: "∀ x s. x ∈ (supp (p s)) ⇒ healthy (wp (a x))
  and up: "∀ s. unitary (p s)
  and sump: "∀ s. sum (p s) (supp (p s)) ≤ 1
  and fsupp: "∀ s. finite (supp (p s))
  shows bd-cts (wp (SetPC a p))
proof −
  from assms have bd-cts (wp (Bind p (λ p. SetPC a (λ- p))))
    by(intro intro::cts-wp-Bind cts-wp-SetPC-const)
  thus ?thesis by(simp add:SetPC-Bind[symmetric])
qed

lemma wp-SetDC-Bind:
  SetDC a S = Bind S (λ S. SetDC a (λ- S))
by(intro ext, simp add:SetDC-def Bind-def)

lemma SetDC-finite-insert:
  assumes fS: "finite S"
  and neS: "S ≠ {}"
  shows "SetDC a (λ- (insert x S)) = a x ∩ SetDC a (λ- S)"
proof (intro ext, simp add: SetDC-def DC-def cong del: INF-cong-simp)
  fix ab P s
  from fS have A: "finite (insert (a x ab P s) ((λx. a x ab P s) ' S))"
    and B: "finite (((λx. a x ab P s) ' S))"
    by(auto)
  from neS have C: "insert (a x ab P s) ((λx. a x ab P s) ' S) ≠ {}"
    and D: "(λx. a x ab P s) ' S ≠ {}"
    by(auto)
  from A C have Inf (insert (a x ab P s) ((λx. a x ab P s) ' S)) =
    Min (insert (a x ab P s) ((λx. a x ab P s) ' S))
    by(auto intro::Inf-eq-Min)
  also from B D have "... = min (a x ab P s) (Min ((λx. a x ab P s) ' S))"
    by(auto intro:Min-insert)
  also from B D have "... = min (a x ab P s) (Inf ((λx. a x ab P s) ' S))"
    by(simp add::Inf-eq-Min)
  finally show (INF x∈insert x S. a x ab P s) =
    min (a x ab P s) (INF x∈S. a x ab P s)
    by (simp cong del: INF-cong-simp)
qed

lemma SetDC-singleton:
  SetDC a (λ- {x}) = a x
by (simp add: SetDC-def cong del: INF-cong-simp)
lemma cts-wp-SetDC-const:
fixes \( a :: 'a \Rightarrow 's \text{ prog} \)
assumes \( ca: \forall x. x \in S \implies \text{bd-cts} (\text{wp} (a \ x)) \)
and \( ha: \forall x. x \in S \implies \text{healthy} (\text{wp} (a \ x)) \)
and \( fS: \text{finite} S \)
and \( neS: S \neq \{\} \)
shows \( \text{bd-cts} (\text{wp} (\text{SetDC} a (\lambda.. S))) \)

proof –
have \( \text{finite} S \implies S \neq \{\} \implies \)
(\( \forall x \in S. \text{bd-cts} (\text{wp} (a \ x)) \)) \( \implies \)
(\( \forall x \in S. \text{healthy} (\text{wp} (a \ x)) \)) \( \implies \)
\( \text{bd-cts} (\text{wp} (\text{SetDC} a (\lambda.. S))) \)

proof\( (\text{induct} S \text{ rule:finite-induct, simp, clarsimp}) \)
fix \( x::'a \) and \( F::'a \text{ set} \)
assume \( fF: \text{finite} F \)
and \( IH: F \neq \{\} \implies \text{bd-cts} (\text{wp} (\text{SetDC} a (\lambda.. F))) \)
and \( cax: \text{bd-cts} (\text{wp} (a \ x)) \)
and \( hax: \text{healthy} (\text{wp} (a \ x)) \)
and \( haF: \forall x \in F. \text{healthy} (\text{wp} (a \ x)) \)
show \( \text{bd-cts} (\text{wp} (\text{SetDC} a (\lambda.. \text{insert} x F))) \)

proof\( (\text{cases} F = \{\}, \text{simp add:SetDC-singleton cax}) \)
assume \( F \neq \{\} \)
with \( fF cax hax haF IH \) show \( \text{bd-cts} (\text{wp} (\text{SetDC} a (\lambda.. \text{insert} x F))) \)
by\( (\text{auto intro!:cts-wp-DC healthy-intros simp:SetDC-finite-insert}) \)
qed

with \( \text{assms} \) show \( ?\text{thesis} \) by\( (\text{auto}) \)

qed

lemma cts-wp-SetDC:
fixes \( a :: 'a \Rightarrow 's \text{ prog} \)
assumes \( ca: \forall x s. x \in S s \implies \text{bd-cts} (\text{wp} (a \ x)) \)
and \( ha: \forall x s. x \in S s \implies \text{healthy} (\text{wp} (a \ x)) \)
and \( fS: \forall s. \text{finite} (S s) \)
and \( neS: \forall s. S s \neq \{\} \)
shows \( \text{bd-cts} (\text{wp} (\text{SetDC} a S)) \)

proof –
from \( \text{assms} \) have \( \text{bd-cts} (\text{wp} (\text{Bind} S (\lambda S. \text{SetDC} a (\lambda.. S)))) \)
by\( (\text{iprover intro!:cts-wp-Bind cts-wp-SetDC-const}) \)
thus \( ?\text{thesis} \) by\( (\text{simp add:wp-SetDC-Bind[symmetric]}) \)

qed

lemma cts-wp-repeat:
\( \text{bd-cts} (\text{wp} a) \implies \text{healthy} (\text{wp} a) \implies \text{bd-cts} (\text{wp} (\text{repeat} n a)) \)
by\( (\text{induct} n, \text{auto intro!:cts-wp-Skip cts-wp-Seq healthy-intros}) \)

lemma cts-wp-Embed:
\( \text{bd-cts} t \implies \text{bd-cts} (\text{wp} (\text{Embed} t)) \)
4.3. CONTINUITY

4.3.2 Continuity of a Single Loop Step

A single loop iteration is continuous, in the more general sense defined above for transformer transformers.

lemma cts-wp-loopstep:
  fixes body::'s prog
  assumes hb: healthy (wp body)
  and cb: bd-cts (wp body)
  shows bd-cts-tr (λx. wp (body ;; Embed x « G » ⊕ Skip)) (is bd-cts-tr ?F)
proof(rule bd-cts-trI, rule le-trans-antisym)
  fix M::nat ⇒ 's trans and b::real
  assume chain: ⋀i. le-trans (M i) (M (Suc i))
  and fM: ⋀i. feasible (M i)
  shows fw: le-trans (Sup-trans (range (M t Q)) (M t Q))
proof(rule le-transI[OF Sup-trans-least2], clarsimp)
    fix P Q::'s expect and t
    assume sP: sound P
    assume nQ: nneg Q and bP: bounded-by (bound-of P) Q
    hence sQ: sound Q by(auto)
  from fM have fSup: feasible (Sup-trans (range M))
  by(auto intro:feasible-Sup-trans)
  from sQ fM have M t Q ⊢ Sup-trans (range M) Q
  by(auto intro:Sup-trans-upper2)
  moreover from sQ fM have sQ 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) Q ⊢ ?F (Sup-trans (range M)) Q
      by(intro le-funI, simp add:wp-eval mult-left-mono)
  show nneg (wp (body ;; Embed (Sup-trans (range M)) « G » ⊕ Skip) Q)
    proof(rule nnegI, simp add:wp-eval)
      fix s::'s
      from fSup sQ have sound (Sup-trans (range M) Q) by(auto)
      with hb have sound (wp body (Sup-trans (range M) Q)) by(auto)
      hence 0 ≤ wp body (Sup-trans (range M) Q) s by(auto)
      moreover from sQ have 0 ≤ Q s by(auto)
      ultimately show 0 ≤ «G» s * wp body (Sup-trans (range M) Q) s + (1 – «G» s) * Q s
        by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)
    qed
  next
  fix P::'s expect assume sP: sound P
thus \( \text{nneq} \; P \text{ bounded-by} \; (\text{bound-of} \; P) \; P \text{ by(auto)} \)

\[
\begin{align*}
\text{show} \; & \forall \; t \in \text{range} \; \{(\lambda x. \; \text{wp body} \; ; \; \text{Embed} \; x \; _x \; G \; s \; (\otimes \; \text{Skip}) \; o \; M)\}. \\
& \forall \; R. \; \text{nneq} \; R \; \land \; \text{bounded-by} \; (\text{bound-of} \; P) \; R \rightarrow \\
& \text{nneq} \; (u \; R) \; \land \; \text{bounded-by} \; (\text{bound-of} \; P) \; (u \; R) \\
\text{proof} \; (\text{clarsimp, intro conjI} \; \text{nneqI} \; \text{bounded-byI}, \; \text{simp-all add:wp-eval}) \\
\text{fix} \; u::\text{nat} \; \text{and} \; R::'s \; \text{expect} \; \text{and} \; s::'s \\
\text{assume} \; sR: \; \text{nneq} \; R \; \text{and} \; bR: \; \text{bounded-by} \; (\text{bound-of} \; P) \; R \\
\text{hence} \; sR: \; \text{sound} \; R \text{ by(auto)} \\
\text{with} \; fM \; \text{have} \; s\text{MuR}: \; \text{sound} \; (M \; u \; R) \text{ by(auto)} \\
\text{with} \; hb \; \text{have} \; \text{sound} \; (\text{wp body} \; (M \; u \; R)) \text{ by(auto)} \\
\text{hence} \; 0 \leq \text{wp body} \; (M \; u \; R) \; s \text{ by(auto)} \\
\text{moreover from} \; nR \; \text{have} \; 0 \leq R \; s \text{ by(auto)} \\
\text{ultimately show} \; 0 \leq \langle G \rangle \; s \; \text{* wp body} \; (M \; u \; R) \; s \; + \; (1 - \langle G \rangle \; s) \; \text{* R} \; s \\
\text{by} \; (\text{auto intro add-nonneg nonneg mult-nonneg nonneg}) \\
\text{from} \; sR \; bR \; fM \; \text{have} \; \text{bounded-by} \; (\text{bound-of} \; P) \; (M \; u \; R) \text{ by(auto)} \\
\text{with} \; s\text{MuR} \; hb \; \text{have} \; \text{bounded-by} \; (\text{bound-of} \; P) \; (\text{wp body} \; (M \; u \; R)) \text{ by(auto)} \\
\text{hence} \; \text{wp body} \; (M \; u \; R) \; s \leq \text{bound-of} \; P \text{ by(auto)} \\
\text{moreover from} \; bR \; \text{have} \; R \; s \leq \text{bound-of} \; P \text{ by(auto)} \\
\text{ultimately have} \; \langle G \rangle \; s \; \text{* wp body} \; (M \; u \; R) \; s \; + \; (1 - \langle G \rangle \; s) \; \text{* R} \; s \leq \\
\langle G \rangle \; s \; \text{* bound-of} \; P \; + \; (1 - \langle G \rangle \; s) \; \text{* bound-of} \; P \\
\text{by} \; (\text{auto intro add mono mult left mono}) \\
\text{also have ... = bound-of} \; P \; \text{by} \; (\text{simpl add algebra simp}) \\
\text{finally show} \; \langle G \rangle \; s \; \text{* wp body} \; (M \; u \; R) \; s \; + \; (1 - \langle G \rangle \; s) \; \text{* R} \; s \leq \text{bound-of} \; P \\
\text{.} \\
\text{qed} \\
\text{qed} \\
\text{show} \; \text{le-trans} \; (?F \; (\text{Sup-trans} \; (\text{range} \; M))) \; (\text{Sup-trans} \; (\text{range} \; (?F \; o \; M))) \\
\text{proof} \; (\text{rule le-transI, rule le-funI, simp add: wp-eval cong del: image cong simp}) \\
\text{fix} \; P::'s \; \text{expect} \; \text{and} \; s::'s \\
\text{assume} \; sP: \; \text{sound} \; P \\
\text{have} \; \{t \; P \; | \; t \; \in \; \text{range} \; M\} \; = \; \text{range} \; (\lambda i. \; M \; i \; P) \\
\text{by} \; (\text{blast}) \\
\text{hence} \; \text{wp body} \; (\text{Sup-trans} \; (\text{range} \; M) \; P) \; s \; = \; \text{wp body} \; (\text{Sup-exp} \; (\text{range} \; (\lambda i. \; M \; i \; P))) \; s \\
\text{by} \; (\text{simpl add Sup-trans-def}) \\
\text{also} \; \{ \\
\text{from} \; sP \; fM \; \text{have} \; \lambda i. \; \text{sound} \; (M \; i \; P) \text{ by(auto)} \\
\text{moreover from} \; sP \; \text{chain} \; \text{have} \; \lambda i. \; M \; i \; P \; \vdash \; M \; (\text{Suc} \; i) \; P \text{ by(auto)} \\
\text{moreover} \; \{ \\
\text{from} \; sP \; \text{have} \; \text{bounded-by} \; (\text{bound-of} \; P) \; P \text{ by(auto)} \\
\text{with} \; sP \; fM \; \text{have} \; \lambda i. \; \text{bounded-by} \; (\text{bound-of} \; P) \; (M \; i \; P) \text{ by(auto)} \\
\} \\
\text{ultimately have} \; \text{wp body} \; (\text{Sup-exp} \; (\text{range} \; (\lambda i. \; M \; i \; P))) \; s \; = \\
\text{Sup-exp} \; (\text{range} \; (\lambda i. \; \text{wp body} \; (M \; i \; P))) \; s \\
\text{by} \; (\text{subst bd-ctsD[OF cb, auto simp:o def}) \\
\text{also have} \; \text{Sup-exp} \; (\text{range} \; (\lambda i. \; \text{wp body} \; (M \; i \; P))) \; s =
4.3. CONTINUITY

\[ \text{Sup} \{ f \mid f \in \text{range} (\lambda x. \text{wp body} (M \ i \ P)) \} \]

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

finally have \(\«G» \ s \ast \text{wp body} (\text{Sup-trans} (\text{range} M) \ P) \ s + (1 - «G» \ s) \ast P \)

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

also \{\text{from sP fM have} \land i. \text{sound} (M \ i \ P) \text{ by(auto)}\}

moreover \text{from sP fM have} \land i. \text{bounded-by} (\text{bound-of} P) (M \ i \ P) \text{ by(auto)}

ultimately have \land i. \text{bounded-by} (\text{bound-of} P) \text{ (wp body} (M \ i \ P)) \text{ using hb}\n
by(auto)

\text{by(auto)}

\text{having} \land i. \text{wp body} (M \ i \ P) \ s \leq \text{bound-of} P \text{ by(auto)}

moreover \{\text{have} \{ \«G» \ s \ast x \mid x \in \{ f \mid f \in \text{range} (\lambda x. \text{wp body} (M \ i \ P)) \} \} = \{ \«G» \ s \ast f \mid f \in \text{range} (\lambda x. \text{wp body} (M \ i \ P)) \} \text{ by(blast)}\}

ultimately \text{have} \{ \«G» \ s \ast \text{Sup} \{ f \mid f \in \text{range} (\lambda x. \text{wp body} (M \ i \ P)) \} = \text{Sup} \{ \«G» \ s \ast f \mid f \in \text{range} (\lambda x. \text{wp body} (M \ i \ P)) \} \text{ by(subst cSup-mult, auto)}\}

moreover \{\text{have} \{ x + (1 - «G» \ s) \ast P \mid x, \ x \in \{ «G» \ s \ast f \mid f \in \text{range} (\lambda x. \text{wp body} (M \ i \ P)) \} \} = \{ «G» \ s \ast f + (1 - «G» \ s) \ast P \mid f \in \text{range} (\lambda x. \text{wp body} (M \ i \ P)) \} \text{ by(blast)}\}

moreover \text{from bound sP have} \land i. \text{wp body} (M \ i \ P) \ s \leq \text{bound-of P}\n
\text{by(auto)}

ultimately \text{have} \{ \text{Sup} \{ «G» \ s \ast f \mid f \in \text{range} (\lambda x. \text{wp body} (M \ i \ P)) \} + (1 - «G» \ s) \ast P \ s = \text{Sup} \{ «G» \ s \ast f + (1 - «G» \ s) \ast P \mid f \in \text{range} (\lambda x. \text{wp body} (M \ i \ P)) \} \text{ by(subst cSup-add, auto)}\}

ultimately \text{have} \{ \text{Sup} \{ «G» \ s \ast f \mid f \in \text{range} (\lambda x. \text{wp body} (M \ i \ P)) \} + (1 - «G» \ s) \ast P \ s = \text{Sup} \{ «G» \ s \ast f + (1 - «G» \ s) \ast P \mid f \in \text{range} (\lambda x. \text{wp body} (M \ i \ P)) \} \text{ by(simp)}\}

also \{\text{have} \land i. \text{wp body} (M \ i \ P) \ s + (1 - «G» \ s) \ast P \ s = \text{ultimately have} \land i. \text{wp body} (\text{Embed} x \ a G \oplus \text{Skip}) \circ M \ i \ P \ s \text{ by(simp add:wp-eval)}\}

also \text{have} \land i. \ast \ i \leq \text{Sup} \{ f \mid f \in \{ t \mid t \in \text{range} ((\lambda x. \text{wp body} \text{ Embed} x \ a G \oplus) \} \}

\text{by(auto)}

CHAPTER 4. THE PGCL LANGUAGE

\[\text{Skip}) \circ M)\]

proof (intro cSup-upper bdd-aboveI, blast, clarsimp simp:wp-eval)
  fix \( i \)
  from \( \text{sP} \) have \( bP \): bounded-by \((\text{bound-of } P) \) \( P \) by (auto)
  with \( \text{sP} M \) have sound \((M i P) \) bounded-by \((\text{bound-of } P) \) \((M i P) \) by (auto)
  with \( bP \) have \( \text{wp body} (M i P) \) \( s \) \( \leq \) \( \text{bound-of } P \) \( P \) by (auto)
  hence \( «G» s \ast \text{wp body} (M i P) \) \( s \) \( + \) \((1 \ast «G» s) \ast P \) \( s \) \( \leq \)
  \((G) \) \( s \ast \) \((\text{bound-of } P) \) \( P \) \( s \) \( \leq \)
  by (auto intro:add-mono mult-left-mono)
  also have \( ... = \) \( \text{bound-of } P \) by (simp add: algebra-simps)
  finally show \( «G» s \ast \text{wp body} (M i P) \) \( s \) \( + \) \((1 \ast «G» s) \ast P \) \( s \) \( \leq \) \( \text{bound-of } P \).
  qed
finally
  have \( \text{Sup} \{ «G» s \ast f \ast s \ast (1 \ast «G» s) \ast P \ast s \mid f \ast f \in \text{range } (\lambda \text{i. } \text{wp body} (M i P))\} \leq \)
  \( \text{Sup} \{ f \ast f \in \{ t P \mid t \ast t \in \text{range } ((\lambda \text{x. } \text{wp body ;; Embed x «G» ⊕ Skip}) \circ M)\}\} \)
  by (blast intro:cSup-least)
  }
  also have \( \text{Sup} \{ f \ast f \in \{ t P \mid t \ast t \in \text{range } ((\lambda \text{x. } \text{wp body ;; Embed x «G» ⊕ Skip}) \circ M)\}\} = \)
  \( \text{Sup-trans (range ((\lambda \text{x. } \text{wp body ;; Embed x «G» ⊕ Skip}) \circ M) P s} \)
  by (simp add: Sup-trans-def Sup-exp-def)
  finally show \( «G» s \ast \text{wp body} (\text{Sup-trans (range } M) P s \ast \text{ (1 \ast «G» s) \ast P} \)
  \( s \leq \)
  \( \text{Sup-trans (range ((\lambda \text{x. } \text{wp body ;; Embed x «G» ⊕ Skip}) \circ M)) P} \)
  \( s \).
  qed
qed
end

4.4 Continuity and Induction for Loops

theory LoopInduction imports Healthiness Continuity begin

Showing continuity for loops requires a stronger induction principle than we have used so far, which in turn relies on the continuity of loops (inductively). Thus, the proofs are intertwined, and broken off from the main set of continuity proofs. This result is also essential in showing the sublinearity of loops.

A loop step is monotonic.

lemma wp-loop-step-mono-trans:
  fixes body::'s prog
  assumes sP: sound P
and hb: healthy (wp body)
shows mono-trans (λQ s. «G» s * wp body Q s + «N G» s * P s)
proof (intro mono-transI le-funI, simp)
fix Q R::′s expect and s::′s
assume sQ: sound Q and sR: sound R and le: Q ⊢ R
hence wp body Q ⊢ wp body R
by (rule mono-transD[OF healthy-monoD, OF hb])
thus «G» s * wp body Q s ≤ «G» s * wp body R s
by (auto dest: le-funD intro: mult-left-mono)
qed

We can therefore apply the standard fixed-point lemmas to unfold it:

lemma lfp-wp-loop-unfold:
fixes body::′s prog
assumes hb: healthy (wp body)
and sP: sound P
shows lfp-exp (λQ s. «G» s * wp body Q s + «N G» s * P s) =
(λs. «G» s * wp body (lfp-exp (λQ s. «G» s * wp body Q s + «N G» s * P s)) s + «N G» s * P s)
proof (rule lfp-exp-unfold)
from assms show mono-trans (λQ s. «G» s * wp body Q s + «N G» s * P s)
by (blast intro: wp-loop-step-mono-trans)
from assms show λs. «G» s * wp body (λs. bound-of P) s + «N G» s * P s ⊢
λs. bound-of P
by (blast intro: lfp-loop-fp)
from sP show sound (λs. bound-of P)
by (auto)
fix s::′s
from uQ hb have uwQ: sound (wp body Q s)
by (auto)
with sP show sound (λs. «G» s * wp body Q s + «N G» s * P s)
by (intro wp-loop-step-sound[unfolded wp-eval, simplified, folded negate-embed], auto)
qed

lemma wp-loop-step-unitary:
fixes body::′s prog
assumes hb: healthy (wp body)
and uP: unitary P and uQ: unitary Q
shows unitary (λs. «G» s * wp body Q s + «N G» s * P s)
proof (intro unitarilyI2 anegI bounded-byI)
fix s::′s
from uQ hb have uuQ: unitary (wp body Q) by (auto)
with uP have 0 ≤ wp body Q s 0 ≤ P s by (auto)
thus «G» s * wp body Q s + «N G» s * P s ≤ 0 by (auto)
by (auto intro: add-nonneg-nonneg mult-nonneg-nonneg)

from uP uuQ have wp body Q s ≤ 1 P s ≤ 1 by (auto)
hence «G» s * wp body Q s + «N G» s * P s ≤ «G» s * 1 + «N G» s * 1
by (blast intro: add-mono mult-left-mono)
also have \( \ldots = 1 \) by (simp add: negate-embed)
finally show \( \langle G \rangle \cdot s + wp \cdot body \cdot Q \cdot s + \langle N \rangle \cdot G \cdot s + P \cdot s \leq 1 \).
qed

lemma lfp-loop-unitary:
  fixes body ::'s prog
  assumes healthy: \( wp \cdot body \) 
  and unitary P
  shows unitary \( \langle \lambda Q \cdot s. \langle G \rangle \cdot s + wp \cdot body \cdot Q \cdot s + \langle N \rangle \cdot G \cdot s + P \cdot s \rangle \).
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 healthy: \( wp \cdot body \) 
  and nearly-healthy: \( wlp \cdot body \)
  and Limit: \( \forall S. \left[ \forall x \in S. \forall Q. unitary Q \implies unitary (snd \cdot x \cdot Q) \right] \implies \left[ P \cdot (\text{Sup-trans} (fst \cdot S)) \implies (\text{Inf-utrans} (snd \cdot S)) \right] \)
  and IH: \( \forall t u. \left[ P \cdot t \cdot u \cdot \text{feasible} \cdot t \cdot \forall Q. \text{unitary} \cdot Q \implies \text{unitary} \cdot (u \cdot Q) \right] \implies P \cdot t' \cdot u' \cdot \text{equiv-trans} \cdot t \cdot t' \cdot \text{equiv-utrans} \cdot u \cdot u' \)
  and P-equiv: \( \forall t t' u u'. \left[ P \cdot t \cdot u \cdot \text{equiv-trans} \cdot t \cdot t' \cdot \text{equiv-utrans} \cdot u \cdot u' \right] \implies P \cdot t' \cdot u' \)
  shows \( P \cdot (wp \cdot (do \ G \implies body \cdot od)) \cdot (wlp \cdot (do \ G \implies body \cdot od)) \)
  — The property must be preserved by equivalence.
  — 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 \(?X \cdot t = wp \cdot (body :: Embed t :: G \circ Skip)\)
let \(?Y \cdot t = wlp \cdot (body :: Embed t :: G \circ Skip)\)

let \(?M = \{ x. \text{feasible} \cdot (fst \cdot x) \cdot (snd \cdot x) \land \\
\quad \forall Q. \text{unitary} \cdot Q \implies \text{unitary} \cdot (snd \cdot x \cdot Q) \land \\
\quad \text{le-trans} \cdot (fst \cdot x) \cdot (\text{lfp-trans} \cdot ?X) \land \\
\quad \text{le-utrans} \cdot (\text{gfp-trans} \cdot ?Y) \cdot (snd \cdot x) \} \)

have \( \text{fsup: feasible} \cdot (\text{Sup-trans} (fst \cdot ?M)) \)
proof (intro feasibleI bounded-byI2 nnegI2)
fix Q ::'s expect and \( k :: \text{real} \)
assume \( nQ :: \text{nneg} \cdot Q \land bQ :: \text{bounded-by} \cdot b \cdot Q \)
4.4. CONTINUITY AND INDUCTION FOR LOOPS

show \( \text{Sup-trans} \ (\text{fst} \ ?M) \) \( Q \vdash \lambda s. \ b \)

unfolding \( \text{Sup-trans-def} \)
using \( nQ \ bQ \) by (auto intro; \( \text{Sup-exp\ least} \))

show \( \lambda s. \ 0 \vdash \text{Sup-trans} \ (\text{fst} \ ?M) \) \( Q \)

proof (cases)
assume empty: \( ?M = \{ \} \)
show \( ?\text{thesis} \) by (simp add: \( \text{Sup-trans-def} \) \( \text{Sup-exp-def} \) empty)
next
assume \( ?M \neq \{ \} \)
then obtain \( x \) where \( \text{xin: } x \in \ ?M \) by auto
hence \( \text{ffx: } \) feasible \( (\text{fst} \ x) \) by (simp)
with \( nQ \ bQ \) have \( \lambda s. \ 0 \vdash \text{Sup-trans} \ (\text{fst} \ ?M) \) \( Q \)
apply (intro \( \text{Sup-trans-upper2} \) [OF \( \text{imageI} \ - nQ \ bQ \)], assumption)
apply (clarsimp, blast intro: sound-nneg [OF feasible-sound] feasible-boundedD)
done
finally show \( \lambda s. \ 0 \vdash \text{Sup-trans} \ (\text{fst} \ ?M) \) \( Q \).

qed

qed

have \( u\text{Inf: } \bigwedge P. \ \text{unitary } P \Rightarrow \text{unitary } (\text{Inf-utrans} \ (\text{snd} \ ?M) \) \( P \)

proof (cases \( ?M = \{ \} \))
fix \( P \)
assume empty: \( ?M = \{ \} \)
show \( ?\text{thesis} \) \( P \) by (simp only: empty, simp add: \( \text{Inf-utrans-def} \))
next
fix \( P::\s \) expect
assume \( uP\) : \( \text{unitary } P \)
and ne: \( ?M \neq \{ \} \)
show \( ?\text{thesis} \) \( P \)
proof (intro unitaryI2 nnegI2 bounded-byI2)
from ne obtain \( x \) where \( \text{xin: } x \in \ ?M \) by auto
hence \( \text{sxin: } \) snd \( x \in \text{snd} \ ?M \) by (simp)

hence \( \text{le-utrans } \) (\( \text{Inf-utrans} \ (\text{snd} \ ?M) \)) \( (\text{snd} \ x) \)
by (intro Inf-utrans-lower, auto)
with \( uP \)
have \( \text{Inf-utrans} \ (\text{snd} \ ?M) \) \( P \vdash \text{snd} \ x \) \( P \) by (auto)
also {
from \( \text{xin uP} \) have \( \text{unitary } (\text{snd} \ x \) \( P \) \) by (simp)

hence \( \text{snd} \ x \) \( P \vdash \lambda s. \ 1 \) by (auto)
}
finally show \( \text{Inf-utrans} \ (\text{snd} \ ?M) \) \( P \vdash \lambda s. \ 1 \).

have \( \lambda s. \ 0 \vdash \text{Inf-trans} \ (\text{snd} \ ?M) \) \( P \)
unfolding \( \text{Inf-trans-def} \)
proof (rule Inf-exp-greatest)
from \( \text{sxin} \) show \( \{ t \ P\ | t \in \text{snd} \ ?M \} \neq \{ \} \) by (auto)
show \( \forall P \in \{ t \ P\ | t \in \text{snd} \ ?M \}. \lambda s. \ 0 \vdash P \)
proof (clarsimp)
fix \( t : \)'s trans
assume \( \forall Q. \text{unitary } Q \rightarrow \text{unitary } (t Q) \)
with \( uP \) have \( \text{unitary } (t P) \) by(auto)
thus \( \lambda s. 0 \vdash t P \) by(auto)
qed

also {
from \( ne \) have \( X : (\text{snd '} ?M = \{\}) = \text{False} \) by(simp)
have \( \text{Inf-trans } (\text{snd '} ?M) P = \text{Inf-utrans } (\text{snd '} ?M) P \)
unfolding \( \text{Inf-utrans-def by(subst \( \text{X} \), simp) } \)
}
finally show \( \lambda s. 0 \vdash \text{Inf-utrans } (\text{snd '} ?M) P \).
qed

have \( \text{wp-loop-mono} : \forall t u. \left[ \text{le-trans } t u ; \forall P. \text{sound } P \implies \text{sound } (t P) ; \right. \\
\left. \forall P. \text{sound } P \implies \text{sound } (u P) \right] \implies \text{le-trans } (?X t) (?X u) \)
proof(intro le-transI le-funI, simp add:wp-eval)
fix \( t u : \)'s trans and \( P : \)'s expect and \( s : \)'s
assume \( \text{le} : \text{le-trans } t u \)
and \( \text{st} : \forall P. \text{sound } P \implies \text{sound } (t P) \)
and \( \text{su} : \forall P. \text{sound } P \implies \text{sound } (u P) \)
and \( \text{uP} : \text{unitary } P \)
hence \( \text{unitary } (t P) \) \( \text{unitary } (u P) \) by(auto)
with healthy-monoD[OF \( \text{hwp} \) \( \text{le su} \) 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 intro:nearly-healthy-monoD)
thus \( \langle G \rangle s * \text{wp body } (t P) \) \( s \leq \langle G \rangle s * \text{wp body } (u P) \) \( s \) by(auto intro:mult-left-mono)
qed

have \( \text{wlp-loop-mono} : \forall t u. \left[ \text{le-utrans } t u ; \forall P. \text{unitary } P \implies \text{unitary } (t P) ; \right. \\
\left. \forall P. \text{unitary } P \implies \text{unitary } (u P) \right] \implies \text{le-utrans } (?Y t) (?Y u) \)
proof(intro le-utransI le-funI, simp add:wp-eval)
fix \( t u : \)'s trans and \( P : \)'s expect and \( s : \)'s
assume \( \text{le} : \text{le-utrans } t u \)
and \( \text{ut} : \forall P. \text{unitary } P \implies \text{unitary } (t P) \)
and \( \text{su} : \forall P. \text{unitary } P \implies \text{unitary } (u P) \)
and \( \text{uP} : \text{unitary } P \)
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-healthy-monoD[OF \( \text{hwlp} \) \( \text{le uP} \) have \( \text{wlp body } (t P) \) \( s \leq \text{wlp body } (u P) \) \( s \) by(auto)
thus \( \langle G \rangle s * \text{wlp body } (t P) \) \( s \leq \langle G \rangle s * \text{wlp body } (u P) \) \( s \)
by(auto intro:mult-left-mono)
qed

from \( \text{hwp} \) have \( \forall t. \text{healthy } t \implies \text{healthy } (?X t) \)
4.4. CONTINUITY AND INDUCTION FOR LOOPS

by (auto intro: healthy-intros)

from hwlp have hY: (∀ t. nearly-healthy t ↦ nearly-healthy (?Y t))
  by (auto intro: healthy-intros)

have PLimit: P (Sup-trans (fst ′ ?M)) (Inf-trans (snd ′ ?M))
  by (auto intro: Limit)

have feasible-lfp-loop:
  feasible (fp-trans ?X)
proof (intro feasibleI bounded-byI2 nnegI2, simp-all add: wp-Loop1 [simplified wp-eval] soundI2 hwlp)
  fix P::'a expect and b::real
  assume bP: bounded-by b P and nP: nneg P
  hence sP: sound P by (auto)
  show lfp-exp (λQ s. « G » s * wp body Q s + « N G » s * P s) ⊢ λs. b
proof (intro lfp-exp-lowerbound le-funI)
  fix s::'a
  from bP nP have nnb: 0 ≤ b by (auto)
  hence sound (λs. b) bounded-by b (λs. b) by (auto)
  with hwlp have bounded-by b (wp body (λs. b)) by (auto)
  with bP have wp body (λs. b) s ≤ b P s ≤ b by (auto)
  hence «G» s * wp body (λs. b) s + « N G » s * P s ≤ « G » s * b + « N G » s * b
    by (auto intro: add-mono mult-left-mono)
  thus «G» s * wp body (λs. b) s + « N G » s * P s ≤ b
    by (simp add: negate-embed algebra-simps)
  from nnb show sound (λs. b) by (auto)
  qed
from hwlp sP show λs. 0 ⊢ lfp-exp (λQ s. « G » s * wp body Q s + « N G » s * P s)
  by (blast intro!: lfp-exp-greatest lfp-loop-fp)
  qed

have unitary-gfp:
  (∀ P. unitary P ↦ unitary (gfp-trans ?Y P))
proof (intro unitaryI2 nnegI2 bounded-byI2, simp-all add: wp-Loop1 [simplified wp-eval] hwlp)
  fix P::'a expect
  assume uP: unitary P
  show λs. 0 ⊢ gfp-exp (λQ s. « G » s * wp body Q s + « N G » s * P s)
proof (rule gfp-exp-upperbound [OF le-funI])
  fix s::'a
  from hwlp uP have 0 ≤ wp body (λs. 0) s 0 ≤ P s by (auto dest!: unitary-sound)
  thus 0 ≤ «G» s * wp body (λs. 0) s + « N G » s * P s
    by (auto intro: add-nonneg-nonneg mult-nonneg-nonneg)
  show unitary (λs. 0) by (auto)
  qed
show gfp-exp (λQ s. « G » s * wp body Q s + « N G » s * P s) ⊢ λs. 1
have $fX$:

\[
\forall t. \text{feasible } t \implies \text{feasible }(\exists X t)
\]

\textbf{proof} (\text{intro feasibleI nnegI bounded-byI, simp-all add:wp-eval})

\textbf{fix} $t::s$ trans and $Q::s$ expect and $b::\text{real}$ and $s::s$'s
\textbf{assume} $ft$: feasible $t$ and $bQ$: bounded-by $b$ $Q$ and $nQ$: nneg $Q$
\textbf{hence} nneg ($t$ $Q$) bounded-by $b$ ($t$ $Q$) \textbf{by(auto)}

\textbf{moreover hence st} $Q$: sound ($t$ $Q$) \textbf{by(auto)}

\textbf{ultimately have $wp$ body} ($t$ $Q$) $s \leq b$ \textbf{using $hwlp$ by(auto)}

\textbf{moreover from $bQ$ have} $Q$ $s \leq b$ \textbf{by(auto)}

\textbf{ultimately have} «$G$» $s$ * $wp$ body ($t$ $Q$) $s$ + ($1 - «G» s$) * $Q$ $s$ \leq

«$G$» $s$ * $b$ + ($1 - «G» s$) * $b$

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

\textbf{thus} «$G$» $s$ * $wp$ body ($t$ $Q$) $s$ + ($1 - «G» s$) * $Q$ $s$ \leq $b$

\textbf{by(simp add:algebra-simps)}

\textbf{from $nQ$ st} $Q$ \textbf{hwlp have} $0 \leq wp$ body ($t$ $Q$) $s$ $0 \leq Q$ $s$ \textbf{by(auto)}

\textbf{thus} $0 \leq «G» s$ * $wp$ body ($t$ $Q$) $s$ + ($1 - «G» s$) * $Q$ $s$

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

\textbf{qed}

\textbf{have $uY$}:

\[
\forall t P. (\forall P. \text{unitary } P \implies \text{unitary } (t P)) \implies \text{unitary } P \implies \text{unitary } (?Y t P)
\]

\textbf{proof} (\text{intro unitaryI2 nnegI bounded-byI, simp-all add:wp-eval})

\textbf{fix} $t::s$ trans and $P::s$ expect and $s::s$'s
\textbf{assume} $ut$: \forall P. \text{unitary } P \implies \text{unitary } (t P)

\textbf{and} $uP$: \text{unitary } $P$
\textbf{hence} $uP$: \text{unitary } ($t$ $P$) \textbf{by(auto)}

\textbf{with $hwlp$ have} $ubtP$: \text{unitary} ($wp$ body ($t$ $P$)) \textbf{by(auto)}

\textbf{with} $uP$ \textbf{have} $0 \leq P$ $s$ $0 \leq wp$ body ($t$ $P$) $s$ \textbf{by(auto)}

\textbf{thus} $0 \leq «G» s$ * $wp$ body ($t$ $P$) $s$ + ($1 - «G» s$) * $P$ $s$

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

\textbf{from} $uP$ $ubtP$ \textbf{have} $P$ $s \leq 1$ $wp$ body ($t$ $P$) $s$ $\leq 1$ \textbf{by(auto)}

\textbf{hence} «$G» s * $wp$ body ($t$ $P$) $s$ + ($1 - «G» s$) * $P$ $s \leq «G» s * 1 + (1 - «G» s) * 1$

\textbf{by(blast intro:add-mono mult-left-mono)}

\textbf{also have} ... $= 1$ \textbf{by(simp add:algebra-simps)}

\textbf{finally show} «$G» s * $wp$ body ($t$ $P$) $s$ + ($1 - «G» s$) * $P$ $s \leq 1$$.
4.4. CONTINUITY AND INDUCTION FOR LOOPS

also have equiv-trans ... (lfp-trans ?X)

proof (rule iffD1[OF equiv-trans-comm, OF lfp-trans-unfold], iprover intro:wp-loop-mono)

fix t::'s trans and P::'s expect

assume st: \( \land Q \), sound \( Q \Rightarrow \) sound \( \{ t \} \)

and sP: sound \( P \)

show sound \( \{ X \ t \} \)

proof (intro sound2 bounded-byI nnegI, simp-all add:wp-eval)

fix s::'

from sP st hwp have \( 0 \leq P \ s \ 0 \leq \) wp body \( \{ t \} \) \( s \) by(auto)

thus \( 0 \leq \langle G \rangle \ s * \) wp body \( \{ t \} \ s + (1 - \langle G \rangle \ s) * P \ s \)

by(blast intro:add-nonneg-nonneg mult-nonneg-nonneg)

from sP st hwp have bounded-by (bound-of \( \{ t \} \) \( t \) \( P \)) by(auto)

with sP st hwp have bounded-by (bound-of \( \{ t \} \) \( wp \) body \( \{ t \} \)) by(auto)

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

moreover from sP st hwp have \( P \ s \leq \) bound-of \( \{ t \} \) \( P \) by(auto)

moreover have \( \langle G \rangle \ s \leq 1 \ 1 - \langle G \rangle \ s \leq 1 \) by(auto)

moreover from sP st hwp have \( 0 \leq wp \) body \( \{ t \} \ s \ 0 \leq P \ s \) by(auto)

moreover have \( (0::real) \leq 1 \) by(simp)

ultimately show \( \langle G \rangle \ s * \) wp body \( \{ t \} \ s + (1 - \langle G \rangle \ s) * P \ s \leq 1 * \) bound-of \( \{ t \} \) \( 1 * \) bound-of \( P \)

by(blast intro:add-mono mult-mono)

qed

next

let \( \tilde{fp} = \lambda R \ s. \) bound-of \( R \)

show le-trans \( \{ ?X \ \tilde{fp} \} \ \tilde{fp} \) by(auto intro:healthy-intros hwp)

fix P::'s expect assume sound \( P \)

thus sound \( \{ \tilde{fp} \ P \} \) by(auto)

qed

finally have le-lfp: le-trans \( \{ ?X \ (Sup-trans (fst' ?M)) \} \) (lfp-trans \( ?X \) ) .

have fu-gfp: le-utrans \( \{ gfp-trans \ ?Y \} \) (Inf-utrans \( \{ snd' \ ?M \} \)

by(auto intro:Inf-utrans-greatest unitary-gfp)

have 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 \( P \ (\{ X \ (Sup-trans (fst' ?M)) \} \) \( \{ ?Y \ (Inf-utrans \{ snd' \ ?M \}) \)

by(iprover intro:H)

moreover from \( \tilde{Sup} \) have feasible \( \{ ?X \ (Sup-trans (fst' ?M)) \} \) by(rule \( \{ X \})

moreover have \( \\bigwedge P, \) unitary \( P \Rightarrow \) unitary \( \{ ?Y \ (Inf-utrans \{ snd' \ ?M \}) \) \( P \)

by(auto intro:uY uInf)

moreover note le-lfp ge-gfp

ultimately have pair-in: \( \{ ?X \ (Sup-trans (fst' ?M)) \) \( ?Y \ (Inf-utrans \{ snd' \ ?M \}) \) \( \in ?M \)

by(simp)
have \( ?X (\text{Sup-trans} (\text{fst} ' ?M)) \in \text{fst} ' ?M \)
   by (rule imageI [OF pair-in, of fst, simplified])

have \( \text{le-trans} (?X (\text{Sup-trans} (\text{fst} ' ?M))) (\text{Sup-trans} (\text{fst} ' ?M)) \)
   proof (rule le-transI [OF Sup-trans-upper2 [where \( t = ?X (\text{Sup-trans} (\text{fst} ' ?M)) \) and \( \text{S} = \text{fst} ' ?M \)])

fix \( P :: 's \text{ expect} \)
assume \( sP :: \text{sound P} \)
thus \( \text{nneg P} \) by (auto)
from \( sP \) show \( \text{bounded-by (bound-of P) P} \) by (auto)

from \( sP \) show \( \forall u \in \text{fst} ' ?M. \forall Q. \text{nneg Q} \land \text{bounded-by (bound-of P) Q} \rightarrow \text{nneg (u Q)} \land \text{bounded-by (bound-of P) (u Q)} \)
   by (auto)

qed

have \( ?Y (\text{Inf-utrans} (\text{snd} ' ?M)) \in \text{snd} ' ?M \)
   by (rule imageI [OF pair-in, of snd, simplified])

hence \( \text{le-utrans} (\text{Inf-utrans} (\text{snd} ' ?M)) (?Y (\text{Inf-utrans} (\text{snd} ' ?M))) \)
   by (intro Inf-utrans-lower, auto)

hence \( \text{le-utrans} (\text{Inf-utrans} (\text{snd} ' ?M)) (\text{gfp-trans} ?Y) \)
   by (blast intro: gfp-trans-upperbound uInf)

from \( P\text{Limit} \) eqt equ show \( P (\text{lfp-trans} ?X) (\text{gfp-trans} ?Y) \) by (rule P-eqie)

proof

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 :: \( 's \text{ prog} 
\Rightarrow \text{'s booll} \Rightarrow \text{nat} 
\Rightarrow 's \text{ trans} \)
where iterates body \( G \ i \) = ((\lambda x. \text{wp (body ;; Embed x \ G } \oplus \text{ Skip})) \ ^^ i) (\lambda P s. 0)

lemma iterates-0[simp]:
   iterates body \( G \ 0 \) = (\lambda P s. 0)
   by (simp add: iterates-def)

lemma iterates-Suc[simp]:
   iterates body \( G \ (\text{Suc } i) \) = \( \text{wp (body ;; Embed (iterates body } G \ i) \ G } \oplus \text{ Skip} \)
   by (simp add: iterates-def)

All iterates are healthy.
4.4. CONTINUITY AND INDUCTION FOR LOOPS

**Lemma** \( \text{iterates-healthy} \):

\[
\text{healthy} (\text{wp body}) \implies \text{healthy} (\text{iterates body } G \ i)
\]

by (induct \( i \), auto intro:healthy-intros)

The iterates are an ascending chain.

**Lemma** \( \text{iterates-increasing} \):

fixes \( \text{body} :: 's \prog \)

assumes \( \text{hb: healthy} (\text{wp body}) \)

shows \( \text{le-trans} (\text{iterates body } G \ i) \ (\text{iterates body } G \ (\text{Suc } i)) \)

proof (induct \( i \))

show \( \text{le-trans} \ (\text{iterates body } G \ 0) \ (\text{iterates body } G \ (\text{Suc } 0)) \)

proof (simp add:iterates-def, rule le-transI)

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

assume sound \( P \)

with \( \text{hb have sound} (\text{wp body} ;; \text{Embed} (\lambda P s. 0) \oslash G \oslash \text{Skip}) P) \)

by (auto intro:wp-loop-step-sound)

thus \( \lambda s. 0 \vdash \vdash \text{wp body} ;; \text{Embed} (\lambda P s. 0) \oslash G \oslash \text{Skip} \)

by (auto)

qed

fix \( i \)

assume \( \text{IH: le-trans} (\text{iterates body } G \ i) \ (\text{iterates body } G \ (\text{Suc } i)) \)

have equiv-trans \( (\text{iterates body } G \ (\text{Suc } i)) \)

by (simp)

also from \( \text{iterates-healthy}[\text{OF hb}] \)

have le-trans ... (\( \text{wp body} ;; \text{Embed} (\text{iterates body } G \ (\text{Suc } i)) \oslash G \oslash \text{Skip} \))

by (blast intro:wp-loop-step-mono[\text{OF hb IH}])

also have equiv-trans ... (iterates body \( G \ (\text{Suc } i)) \)

by (simp)

finally show \( \text{le-trans} (\text{iterates body } G \ (\text{Suc } i)) \ (\text{iterates body } G \ (\text{Suc } i))) \)

qed

**Lemma** \( \text{wp-loop-step-bounded} \):

fixes \( t :: 's \trans \) and \( Q :: 's \expect \)

assumes \( \text{nQ: nneg } Q \)

and \( bQ: \text{bounded-by } b \ Q \)

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

and \( \text{hb: healthy} (\text{wp body}) \)

shows \( \text{bounded-by} b (\text{wp body} ;; \text{Embed} t \oslash G \oslash \text{Skip}) Q) \)

proof (rule bounded-byI, simp add:wp-eval)

fix \( s :: 's \)

from \( nQ \ bQ \text{ have } sQ: \text{sound } Q \) by (auto)

with \( bQ \text{ have sound} (t \ Q) \text{ bounded-by } b \ (t \ Q) \) by (auto)

with \( bQ \text{ have wp body} (t \ Q) s \leq b \ Q s \leq b \) by (auto)

hence \( \preceq G \ s \ast wp body (t \ Q) s + (1 - \neg G \ s) * Q s \leq \)

\( \oslash G \ s * b + (1 - \neg G \ s) * b \)

by (auto intro:add-mono mult-left-mono)
also have \ldots = b by (simp add: algebra-simps)

finally show \langle G \rangle s \ast wp body (t \ Q) s + (1 - \langle G \rangle s) \ast Q s \leq b.

qed

This is the key result: The loop is equivalent to the supremum of its iterates. This proof follows the pattern of lemma continuous_lfp in HOL/Library/Continuity.

lemma lfp-iterates:

fixes body :: 's prog

assumes hb: healthy (wp body)
and cb: bd-cts (wp body)
shows equiv-trans (wp (do G \rightarrow body od)) (Sup-trans (range (iterates body G)))
(is equiv-trans ?X ?Y)

proof (rule le-trans-antisym)
let \textit{?F} = \lambda x. wp (body ;; Embed x \langle G \rangle \oplus \texttt{Skip})
let \textit{?bot} = \lambda (P::'s \Rightarrow real) s::'s. 0::real

have HF: \( \forall i. \text{healthy} (\langle \textit{?F} \rangle ^\langle i \rangle \textit{?bot}) \)

proof
\-
fix i from hb show (thesis i)
  by (induct i, simp-all add: healthy-intros)

qed

from iterates-healthy[OF hb]

have \( \forall i. \text{feasible} (\text{iterates body G i}) \)

by (auto)

hence fSup: feasible (Sup-trans (range (iterates body G)))
    by (auto intro: feasible-Sup-trans)

\{
  \fix i
  have le-trans (\langle \textit{?F} \rangle ^\langle i \rangle \textit{?bot}) ?X
    proof (induct i)
      show le-trans (\langle \textit{?F} \rangle ^\langle 0 \rangle \textit{?bot}) ?X
        proof (simp, intro le-transI)
          \fix P::'s expect
          assume sound P
          with hb healthy-wp-loop
          have sound (wp (\mu x. body ;; x \langle G \rangle \oplus \texttt{Skip}) P)
            by (auto)
          thus \( \lambda s. 0 \vdash wp (\mu x. body ;; x \langle G \rangle \oplus \texttt{Skip}) P \)
            by (auto)
        qed
      \fix i
      assume IH: le-trans (\langle \textit{?F} \rangle ^\langle i \rangle \textit{?bot}) ?X
      have equiv-trans (\langle \textit{?F} \rangle ^\langle Suc i \rangle \textit{?bot}) (\langle \textit{?F} \rangle ^\langle i \rangle \textit{?bot}) by (simp)
      also have le-trans ... (\textit{?F} ?X)
      proof (rule wp-loop-step-mono[OF hb IH])
        \fix P::'s expect
        assume sP: sound P
    \}
4.4. CONTINUITY AND INDUCTION FOR LOOPS  

with hb healthy-wp-loop
show sound (wp (μ x. body ;; x « G ⊕ Skip) P)
  by(auto)
from sP show sound ((?F « i) ?bot P)
  by(rule healthy-sound[OF HF])
qed
also {  
from hb have X: le-trans (wp (body ;; Embed (λP s. bound-of P) « G ⊕ Skip))
  (λP s. bound-of P)
  by(intro le-transI, simp add:wp-eval, auto intro: lfp-loop-fp[unfolded negate-embed])  
  have equiv-trans (?F ?X) ?X
  apply (simp only: wp-eval)
  by(intro i[FD1[OF equiv-trans-comm, OF lfp-trans-unfold]
    wp-loop-step-mono[OF hb] wp-loop-step-sound[OF hb], (blast|rule X)+)
}  
finally show le-trans ((?F « Suc i) ?bot) ?X .
qed
}
hence ∨ i. le-trans (iterates body G i) (wp do G −→ body od)
by(simp add:iterates-def)
thus le-trans ?Y ?X
by(auto intro!:le-transI[OF Sup-trans-least2] sound-nneg
    healthy-sound[OF iterates-healthy, OF hb]
    healthy-bounded-byD[OF iterates-healthy, OF hb]
    healthy-sound[OF healthy-wp-loop hb])

show le-trans ?X ?Y
proof(simp only: wp-eval, rule lfp-trans-lowerbound)
from hb cb have bd-cts-tr ?F by(rule cts-wp-loopstep)
  with iterates-increasing[OF hb] iterates-healthy[OF hb]
  have equiv-trans (?F ?Y) (Sup-trans (range (?F o (iterates body G))))
  by (auto intro!: healthy-feasibleD bd-cts-trD cong del: image-cong-simp)
  also have le-trans (Sup-trans (range (?F o (iterates body G)))) ?Y
  proof(rule le-transI)
    fix P::'s expect
    assume sP: sound P
    show (Sup-trans (range (?F o (iterates body G)))) P ‹ ?Y P
  proof(rule Sup-trans-least2, clarsimp)
    show ∀ u∈range (λx. wp (body ;; Embed x « G ⊕ Skip)) o iterates body G).
      ∀ R. nneg R ∧ bounded-by (bound-of P) R −→
      nneg (u R) ∧ bounded-by (bound-of P) (u R)
  proof(clarsimp, intro conjI)
    fix Q::'s expect and i
    assume nQ: nneg Q and bQ: bounded-by (bound-of P) Q
    hence sound Q by(auto)
moreover from `iterates-healthy[OF hb]
have ` P. sound P \implies sound (iterates body G i P) by(auto)
moreover note hb
ultimately have sound (wp (body ;; Embed (iterates body G i) \( G \oplus \) Skip) Q)
by(auto)
thus nneg (wp (body ;; Embed (iterates body G i) \( G \oplus \) Skip) Q)
by(auto)
from nQ bQ `iterates-healthy[OF hb] hb
show bounded-by (bound-of P) (wp (body ;; Embed (iterates body G i) \( G \oplus \) Skip) Q)
by(rule wp-loop-step-bounded)
qed
from sP show nneg P bounded-by (bound-of P) P by(auto)
next
fix Q::'s expect
assume nQ: nneg Q and bQ: bounded-by (bound-of P) Q
hence sound Q by(auto)
with fSup have sound (Sup-trans (range (iterates body G))) Q by(auto)
thus nneg (Sup-trans (range (iterates body G))) Q by(auto)
fix i
show wp (body ;; Embed (iterates body G i) \( G \oplus \) Skip) Q \(\vdash\)
Sup-trans (range (iterates body G)) Q
proof(rule Sup-trans-upper2[OF - - nQ bQ])
from `iterates-healthy[OF hb]
show \(\forall u \in\) range (iterates body G).
\(\forall R.\) nneg R \land bounded-by (bound-of P) R \(\implies\)
nneg (u R) \land bounded-by (bound-of P) (u R)
by(auto)
have wp (body ;; Embed (iterates body G i) \( G \oplus \) Skip) = iterates body G (Suc i)
by(simp)
also have ... \(\in\) range (iterates body G)
by(blast)
finally show wp (body ;; Embed (iterates body G i) \( G \oplus \) Skip) \(\in\)
range (iterates body G) .
qed
qed
qed
fix P::'s expect
assume sound P
with fSup have sound (?Y P) by(auto)
qed
qed

Therefore, evaluated at a given point (state), the sequence of iterates gives
a sequence of real values that converges on that of the loop itself.

corollary loop-iterates:

\begin{align*}
\text{fixes} & \text{ body::'s prog} \\
\text{assumes} & \text{ hb: healthy (wp body)} \\
\text{and} & \text{ cb: bd-cts (wp body)} \\
\text{and} & \text{ sP: sound P}
\end{align*}

\text{shows} \quad \lambda i. \text{ iterates body G i P s} \longrightarrow \text{ wp (do G \rightarrow body od) P s}

\text{proof —}

\begin{align*}
\text{let} & \quad ?X = \{ f s \mid f, f \in \{ t P \mid t \in \text{ range (iterates body G) \} \} \\
\text{have} & \quad \text{ closure-Sup: Sup ?X \in \text{ closure } ?X}
\end{align*}

\text{proof (rule closure-contains-Sup, simp, clarsimp)}

\begin{align*}
\text{fix} & \quad i \\
\text{from} & \quad \text{ sP have bounded-by (bound-of P) P by (auto)} \\
\text{with} & \quad \text{ iterates-healthy[OF hb] sP have } \bigwedge j. \text{ bounded-by (bound-of P) (iterates body G j P)} \\
\text{by (auto)} \\
\text{thus} & \quad \text{ iterates body G i P s} \leq \text{ bound-of P by (auto)}
\end{align*}

\text{qed}

\begin{align*}
\text{have} & \quad \lambda i. \text{ iterates body G i P s} \longrightarrow \text{ Sup } \{ f s \mid f, f \in \{ t P \mid t \in \text{ range (iterates body G) \} \} \\
\text{proof (rule LIMSEQ-I)}
\end{align*}

\begin{align*}
\text{fix} & \quad r :: \text{ real assume posr: } 0 < r \\
\text{with} & \quad \text{ closure-Sup obtain y where gin: y \in ?X and eg: dist y (Sup ?X) < r by (simp only: closure-approachable, blast)} \\
\text{from} & \quad \text{ gin obtain i where git: y = iterates body G i P s by (auto)} \\
\end{align*}

\begin{align*}
\{ \\
\text{fix} & \quad j \\
\text{have} & \quad i \leq j \longrightarrow \text{ le-trans (iterates body G i) (iterates body G j)}
\end{align*}

\text{proof (induct j, simp, clarify)}

\begin{align*}
\text{fix} & \quad k \\
\text{assume} & \quad I H: i \leq k \longrightarrow \text{ le-trans (iterates body G i) (iterates body G k)} \\
\text{and} & \quad \text{ le: } i \leq \text{ Suc k} \\
\text{show} & \quad \text{ le-trans (iterates body G i) (iterates body G (Suc k))}
\end{align*}

\text{proof (cases i = Suc k, simp)}

\begin{align*}
\text{assume} & \quad i \neq \text{ Suc k} \\
\text{with} & \quad \text{ le have } i \leq k \text{ by (auto)} \\
\text{with} & \quad \text{ I H have le-trans (iterates body G i) (iterates body G k) by (auto)} \\
\text{also note} & \quad \text{ iterates-increasing[OF hb]} \\
\text{finally show} & \quad \text{ le-trans (iterates body G i) (iterates body G (Suc k))}
\end{align*}

\text{qed}

\text{qed}

\begin{align*}
\text{with sP have } & \forall j \geq i. \text{ iterates body G i P s} \leq \text{ iterates body G j P s} \\
\text{by (auto)} \\
\text{moreover} & \quad \{ \\
\text{from sP have bounded-by (bound-of P) P by (auto)} \\
\text{with} & \quad \text{ iterates-healthy[OF hb] sP have } \bigwedge j. \text{ bounded-by (bound-of P) (iterates body G j P)}
\end{align*}
by\,(auto)

\textbf{hence} \bigwedge_j. \text{iterates body } G \ j \ P \ s \leq \text{bound-of } P \ \textbf{by}(auto)

\textbf{hence} \bigwedge_j. \text{iterates body } G \ j \ P \ s \leq \text{Sup } ?X

\textbf{by}(\text{intro } c\text{-Sup-upper bdd-above}\,I, \text{auto})

\textbf{ultimately have} \bigwedge_j. i \leq j \implies

\text{norm } (\text{iterates body } G \ j \ P \ s - \text{Sup } ?X) \leq \text{norm } (\text{iterates body } G \ i \ P \ s - \text{Sup } ?X)

\textbf{by}(auto)

\textbf{also from ey yit have} \text{norm } (\text{iterates body } G \ i \ P \ s - \text{Sup } ?X) < r

\textbf{by}(\text{simp add: dist-real-def})

\textbf{finally show} \:\exists \text{no. } \forall n \geq \text{no. } \text{norm } (\text{iterates body } G \ n \ P \ s - \text{Sup \{f s | f. f \in \{t P | t. t \in \text{range (iterates body } G)\}\}}) < r

\textbf{by}(auto)

\textbf{qed}

\textbf{moreover}

\textbf{from} \text{hb cb sP have wp do } G \longrightarrow \text{body od } P \ s = \text{Sup-trans (range (iterates body } G)) \ P \ s

\textbf{by}(\text{simp add: equiv-trans})(OF lfp-iterates)

\textbf{moreover have} ... = \text{Sup \{f s | f. f \in \{t P | t. t \in \text{range (iterates body } G)\}\}}

\textbf{by}(\text{simp add: Sup-trans-def Sup-exp-def})

\textbf{ultimately show } \text{thesis by}(\text{simp})

\textbf{qed}

The iterates themselves are all continuous.

\textbf{lemma cts-iterates:}

\textbf{fixes} \text{body\:'s prog}

\textbf{assumes} \text{hb: healthy (wp body)}

\textbf{and} \text{cb: bd-cts (wp body)}

\textbf{shows} \text{bd-cts (iterates body } G \ i)

\textbf{proof(\text{induct } i, \text{ simp-all})}

\textbf{have} \text{range } (\lambda(n::nat) \ (s::'s). \ 0::real) = \{\lambda s. \ 0::real\}

\textbf{by}(auto)

\textbf{thus} \text{bd-cts } (\lambda P \ (s::'s). \ 0)

\textbf{by}(\text{intro bd-cts}\,I, \text{ simp add:o-def Sup-exp-def})

\textbf{next}

\textbf{fix } i

\textbf{assume} \text{IH: bd-cts (iterates body } G \ i)

\textbf{thus} \text{bd-cts (wp (body :: Embed (iterates body } G \ i) \ « \ G » \oplus \text{Skip})}

\textbf{by}(\text{blast intro: cts-wp-PC cts-wp-Seq cts-wp-Embed cts-wp-Skip}


\text{healthy-intros iterates-healthy cb hb})

\textbf{qed}

Therefore so is the loop itself.

\textbf{lemma cts-wp-loop:}

\textbf{fixes} \text{body\:'s prog}

\textbf{assumes} \text{hb: healthy (wp body)}

\textbf{and} \text{cb: bd-cts (wp body)}
4.4. CONTINUITY AND INDUCTION FOR LOOPS

shows bd-cts (wp do G \rightarrow body od)

proof(rule bd-ctsI)

fix M::nat \Rightarrow 's expect and b::real
assume chain: \forall i. M i \not\in M (Suc i)
and sM: \forall i. sound (M i)
and bM: \forall i. bounded-by b (M i)

from sM bM iterates-healthy[OF hb]
have \bigwedge i. bounded-by b \ensuremath{(\text{iterates body } G \ i \ (M \ j))} \ by(blast)

hence iB: \bigwedge i. s. iterates body G i (M j) s \leq b \ by(auto)

from sM bM have sSup: sound (\text{Sup-exp} (range M))
by(auto intro;Sup-exp-sound)

with lfp-iterates[OF hb cb]
have wp do G \rightarrow body od (\text{Sup-exp} (range M)) =
\text{Sup-trans} (range (\text{iterates body } G)) (\text{Sup-exp} (range M))
by(simp add:equiv-transD)

also {
from chain sM bM
have \bigwedge i. s. \text{iterates body } G i (\text{Sup-exp} \ (range M)) = \text{Sup-exp} (range \ (\text{iterates body } G \ i) o M))
by(blast intro:bd-ctsD cts-iterates[OF hb cb])

hence \{ t \ (\text{Sup-exp} (range M)) \ |. t \in range \ (\text{iterates body } G) \} =
\{ \text{Sup-exp} (range \ (t o M)) \ |. t \in range \ (\text{iterates body } G) \}
by(auto intro:sym)

hence \text{Sup-trans} (\text{range \ (\text{iterates body } G)}) (\text{Sup-exp} (range M)) =
\text{Sup-exp} \{ \text{Sup-exp} (range \ (t o M)) \ |. t \in range \ (\text{iterates body } G) \}
by(simp add:Sup-trans-def)
}

also {
have \bigwedge s. \{ f s \ |. \exists t. f = (\lambda s. \text{Sup} \{ f s \ |. f \in range \ (t o M) \}) \}\land
\ t \in range \ (\text{iterates body } G) \} =
\text{range} (\lambda i. \text{Sup} \ (\text{range} \ (\lambda j. \text{iterates body } G \ i \ (M \ j) \ s)))
(is \bigwedge s. \ ?X s = ?Y s)

proof(intro antisym subsetI)
fix s x
assume x \in ?X s
then obtain t where rwx: x = \text{Sup} \{ f s \ |. f \in range \ (t o M) \}
\ and t \in range \ (\text{iterates body } G) \ by(auto)

then obtain i where t = \text{iterates body } G \ i \ by(auto)

with rwx have x = \text{Sup} \{ f s \ |. f \in range \ (\lambda j. \text{iterates body } G \ i \ (M \ j)) \}
by(simp add:o-def)

moreover have \{ f s \ |. f \in 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} \ (\text{range} \ (\lambda j. \text{iterates body } G \ i \ (M \ j) \ s))
by(simp)

thus x \in range (\lambda i. \text{Sup} \ (\text{range} \ (\lambda j. \text{iterates body } G \ i \ (M \ j) \ s)))
by(auto)

next
fix $s \cdot x$
assume $x \in ?Y \cdot s$
then obtain $i$ where $A: x = \text{Sup} (\text{range} (\lambda j. \text{iterates body } G \cdot i \cdot (M \cdot j) \cdot s))$
by(auto)

have $\forall s. \{ f \cdot s \mid f. f \in \text{range} (\lambda j. \text{iterates body } G \cdot i \cdot (M \cdot j)) \}$ = 
\text{range} (\lambda j. \text{iterates body } G \cdot i \cdot (M \cdot j) \cdot s) by(auto)

hence $B: (\lambda s. \text{Sup} (\text{range} (\lambda j. \text{iterates body } G \cdot i \cdot (M \cdot j) \cdot s)))$
= (\lambda s. \text{Sup} (\{ f \cdot s \mid f. f \in \text{range} (\text{iterates body } G \cdot i \cdot o \cdot M) \})
by(simp add:o-def)

have $C: \text{iterates body } G \cdot i \in \text{range} (\text{iterates body } G)$ by(auto)

have $\exists f. x = f \cdot s \land$
(\exists t. f = (\lambda s. \text{Sup} \{ f \cdot s \mid f. f \in \text{range} (t \cdot o \cdot M) \}) \land
\quad t \in \text{range} (\text{iterates body } G))
by(intro intro:A B C)

thus $x \in ?X \cdot s$ by(simp)

qed

hence $\text{Sup-exp} \{ \text{Sup-exp} (\text{range} (t \cdot o \cdot M)) \mid t. t \in \text{range} (\text{iterates body } G) \}$ =
(\lambda s. \text{Sup} (\text{range} (\lambda i. \text{Sup} (\text{range} (\lambda j. \text{iterates body } G \cdot i \cdot (M \cdot j) \cdot s))))
by(simp add:Sup-exp-def)

} also have $(\lambda s. \text{Sup} (\text{range} (\lambda i. \text{Sup} (\text{range} (\lambda j. \text{iterates body } G \cdot i \cdot (M \cdot j) \cdot s)))))$
= 
(\lambda s. \text{Sup} (\text{range} (\lambda i. \text{iterates body } G \cdot i \cdot (M \cdot j) \cdot s)))
(is $?X = ?Y$)

proof(rule rule ext, rule antisym)

fix $s::\cdot s$

show $?Y \cdot s \leq ?X \cdot s$

proof(rule cSup-least, blast, clarify)

fix $i \cdot j::\cdot n a t$

from $iB$ have $\text{iterates body } G \cdot i \cdot (M \cdot j) \cdot s \leq \text{Sup} (\text{range} (\lambda j. \text{iterates body } G \cdot i \cdot (M \cdot j) \cdot s)))$
by(intro cSup-upper bdd-above1, auto)

also from $iB$ have $... \leq \text{Sup} (\text{range} (\lambda i. \text{Sup} (\text{range} (\lambda j. \text{iterates body } G \cdot i \cdot (M \cdot j) \cdot s))))$
by(intro cSup-upper cSup-least bdd-above1, (blast intro:cSup-least)+)

finally show $\text{iterates body } G \cdot i \cdot (M \cdot j) \cdot s \leq$
\text{Sup} (range (\lambda i. \text{Sup} (\text{range} (\lambda j. \text{iterates body } G \cdot i \cdot (M \cdot j) \cdot s)))) .

qed

have $\forall i. j. \text{iterates body } G \cdot i \cdot (M \cdot j) \cdot s \leq$
\text{Sup} (range (\lambda(i. j). \text{iterates body } G \cdot i \cdot (M \cdot j) \cdot s))
by(rule cSup-upper, auto intro:iB)

thus $?X \cdot s \leq ?Y \cdot s$
by(intro cSup-least, blast, clarify, simp, blast intro:cSup-least)

qed

also have $... = (\lambda s. \text{Sup} (\text{range} (\lambda j. \text{Sup} (\text{range} (\lambda i. \text{iterates body } G \cdot i \cdot (M \cdot j) \cdot s))))$
(is $?X = ?Y$)
proof(rule rule ext, rule antisym)
4.4. CONTINUITY AND INDUCTION FOR LOOPS

fix s; ’s
have ∃ i j. iterates body G i (M j) s ≤
  Sup (range (λ(i, j). iterates body G i (M j) s))
  by (rule cSup-upper, auto intro:iB)
thus ?Y s ≤ ?X s
by (intro cSup-least, blast, clarify, simp, blast intro:cSup-least)
show ?X s ≤ ?Y s
proof (rule cSup-least, blast, clarify)
fix i j::nat
from iB have iterates body G i (M j) s ≤ Sup (range (λi. iterates body G i (M j) s))
  by (intro cSup-upper bdd-aboveI, auto)
also from iB have ... ≤ Sup (range (λj. Sup (range (λi. iterates body G i (M j) s))))
  by (intro cSup-upper cSup-least bdd-aboveI, blast, blast intro:cSup-least)
finally show iterates body G i (M j) s ≤ Sup (range (λj. Sup (range (λi. iterates body G i (M j) s))))
qed
qed
also { have ∃ s. range (λj. Sup (range (λi. iterates body G i (M j) s)))) =
  {f s | f ∈ range ((λP s. Sup {f s | ∃ t. f = t P ∧
  t ∈ range (iterates body G)}) ◦ M)} (is ∃ s. ?X s = ?Y s)
proof (intro antisym subsetI)
fix s x
assume x ∈ ?X s
then obtain j where rwx: x = Sup (range (λi. iterates body G i (M j) s))
by (auto)
moreover { have ∃ s. range (λi. iterates body G i (M j) s) =
  {f s | f ∈ range (λP s. Sup {f s | ∃ t. f = t P ∧ t ∈ range (iterates body G)})}
  by (auto)
hence (∃s. Sup (range (λi. iterates body G i (M j) s))) ∈
  range ((λP s. Sup {f s | ∃ t. f = t P ∧ t ∈ range (iterates body G)}) ◦ M)
  by (simp add: o-def cong del: SUP-cong-simp)
}
ultimately show x ∈ ?Y s by (auto)
next
fix s x
assume x ∈ ?Y s
then obtain P where rwx: x = P s
  and Pin: P ∈ range ((λP s. Sup {f s | ∃ t. f = t P ∧
  t ∈ range (iterates body G)}) ◦ M)
  by (auto)
then obtain j where P = (λs. Sup {f s | ∃ t. f = t (M j) ∧
  t ∈ range (iterates body G)})
  by (auto)
also {
have \( \forall s \, \{ f \, s \mid \exists t. f = t (M \, j) \land t \in \text{range} (\text{iterates body } G) \} = \text{range} (\lambda i. \text{iterates body } G \, i \, (M \, j) \, s) \) by(auto)

hence \( (\lambda s. \text{Sup} \{ f \, s \mid \exists t. f = t (M \, j) \land t \in \text{range} (\text{iterates body } G) \}) = (\lambda s. \text{Sup} (\text{range} (\lambda i. \text{iterates body } G \, i \, (M \, j) \, s))) \) by(simp)

} finally have \( x = \text{Sup} (\text{range} (\lambda i. \text{iterates body } G \, i \, (M \, j) \, s)) \) by(simp add:wp-wz)

thus \( x \in \mathcal{X} \, s \) by(simp)

qed

hence \( (\lambda s. \text{Sup} (\text{range} (\lambda j. \text{Sup} (\text{range} (\lambda i. \text{iterates body } G \, i \, (M \, j) \, s)))))) = \text{Sup-exp} \, \text{range} (\text{Sup-trans} (\text{range} (\text{iterates body } G) \, o \, M)) \) by (simp add: Sup-exp-def Sup-trans-def cong del: SUP-cong-simp)

} also have \( \text{Sup-exp} \, \text{range} (\text{Sup-trans} (\text{range} (\text{iterates body } G) \, o \, M)) = \text{Sup-exp} \, \text{range} (\text{wp do } G \rightarrow \text{body od o } M) \) by(simp add:o-def equiv-transD[OF lfp-iterates, OF hb cb, OF sM])

finally show \( \text{wp do } G \rightarrow \text{body od} \, (\text{Sup-exp} \, \text{range} \, M) = \text{Sup-exp} \, \text{range} \, (\text{wp do } G \rightarrow \text{body od o } M) \).

qed

lemmas cts-intros =
cts-wp-Abort cts-wp-Skip
ccts-wp-Seqccts-wp-PC
ccts-wp-DC ccts-wp-Embembed
cts-wp-Apply ccts-wp-SetDC
ccts-wp-SetPCCcts-wp-Bind
ccts-wp-repeat

dend

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)
4.5. **SUBLINEARITY**

**Lemma** sublinear-wp-Apply:

```
sublinear (wp (Apply f))
```

by(auto simp:wp-eval)

**Lemma** sublinear-wp-Seq:

```
lemma sublinear-wp-Seq:
  fixes x::'s prog
  assumes slx: sublinear (wp x) and sly: sublinear (wp y)
  and hx: healthy (wp x) and hy: healthy (wp y)
  shows sublinear (wp (x ;; y))
```

proof (rule sublinearI, simp add: wp-eval)

fix P :: 's ⇒ real and Q :: 's ⇒ real and s :: 's

and a::real and b::real and c::real

assume sP: sound P and sQ: sound Q and nna: 0 ≤ a and nnb: 0 ≤ b and nnc: 0 ≤ c

with slx hy have a * wp x (wp y P) s + b * wp x (wp y Q) s ⊙ c ≤ wp x (λs. a * wp y P s + b * wp y Q s ⊙ c) s

by (blast intro: sublinearD)

also {
  from sP sQ nna nnb nnc sly
  have λs. a * wp y P s + b * wp y Q s ⊙ c ≤ wp y (λs. a * P s + b * Q s ⊙ c) s

  by (blast intro: sublinearD)

  moreover from sP sQ by
  have sound (wp y P) and sound (wp y Q) by (auto)

  moreover with nna nnb nnc
  have sound (λs. a * wp y P s + b * wp y Q s ⊙ c)

  by (auto intro!: sound-intros tminus-sound)

  moreover from sP sQ by
  have sound (λs. a * P s + b * Q s ⊙ c)

  by (auto intro!: sound-intros tminus-sound)

  moreover with hy have sound (wp y (λs. a * P s + b * Q s ⊙ c))

  by (blast)

  ultimately
  have wp x (λs. a * wp y P s + b * wp y Q s ⊙ c) s ≤ wp x (wp y (λs. a * P s + b * Q s ⊙ c)) s

  by (blast intro!: le-funD[OF mono-transD[OF healthy-monoD[OF hx]]])
}

finally show a * wp x (wp y P) s + b * wp x (wp y Q) s ⊙ c ≤ wp x (wp y (λs. a * P s + b * Q s ⊙ c)) s .

qed

**Lemma** sublinear-wp-PC:

```
lemma sublinear-wp-PC:
  fixes x::'s prog
  assumes slx: sublinear (wp x) and sly: sublinear (wp y)
  and uP: unitary P
  shows sublinear (wp (x ⊕ y))
```

proof (rule sublinearI, simp add: wp-eval)

fix R :: 's ⇒ real and Q :: 's ⇒ real and s :: 's
and $a::real$ and $b::real$ and $c::real$

assume $sR: sound R$ and $sQ: sound Q$

and $nna: 0 \leq a$ and $nnb: 0 \leq b$ and $nnn: 0 \leq c$

have $a * (P s * wp x Q s + (1 - P s) * wp y Q s) +$
$\quad b * (P s * wp x R s + (1 - P s) * wp y R s) \circ c =$
$(P s * a * wp x Q s + (1 - P s) * a * wp y Q s) +$
$(P s * b * wp x R s + (1 - P s) * b * wp y R s) \circ c$

by(simp add:field-simps)
also
have $\cdots = (P s * a * wp x Q s + P s * b * wp x R s) +$
$\quad ((1 - P s) * a * wp y Q s + (1 - P s) * b * wp y R s) \circ c$

by(simp add:ac-simps)
also
have $\cdots = P s * (a * wp x Q s + b * wp x R s) \circ P s * c) +$
$\quad (1 - P s) * (a * wp y Q s + b * wp y R s) \circ (1 - P s) * c)$

by(simp add:field-simps)
also
have $\cdots \leq (P s * (a * wp x Q s + b * wp x R s) \circ P s * c) +$
$\quad ((1 - P s) * (a * wp y Q s + b * wp y R s) \circ (1 - P s) * c)$

by(rule tminus-add-mono)
also \{ 
  from $uP$ have $0 \leq P s$ and $0 \leq 1 - P s$
  by auto
  hence $(P s * (a * wp x Q s + b * wp x R s) \circ P s * c) +$
$\quad ((1 - P s) * (a * wp y Q s + b * wp y R s) \circ (1 - P s) * c) =$
$\quad P s * (a * wp x Q s + b * wp x R s \circ c) +$
$\quad (1 - P s) * (a * wp y Q s + b * wp y R s \circ c)$

by(simp add:tminus-left-distrib)
\}
also \{ 
  from $sQ sR \ nna \ nnb \ nnc \ slx$
  have $a * wp x Q s + b * wp x R s \circ c \leq$
$\quad wp x (\lambda s. a * Q s + b * R s \circ c) s$
  by(blast)
moreover
from $sQ sR \ nna \ nnb \ nnc \ slx$
  have $a * wp y Q s + b * wp y R s \circ c \leq$
$\quad wp y (\lambda s. a * Q s + b * R s \circ c) s$
  by(blast)
moreover
from $uP$ have $0 \leq P s$ and $0 \leq 1 - P s$
  by auto
ultimately
have $P s * (a * wp x Q s + b * wp x R s \circ c) +$
$\quad (1 - P s) * (a * wp y Q s + b * wp y R s \circ c) \leq$
$\quad P s * wp x (\lambda s. a * Q s + b * R s \circ c) s +$
$\quad (1 - P s) * wp y (\lambda s. a * Q s + b * R s \circ c) s$
4.5. **SUBLINEARITY**

by \((\text{blast intro:add-mono mult-left-mono})\)

} finally
show \(a \ast (P \ast q \ast wp x Q s + (1 - P) \ast wp y Q s) +\)
\(b \ast (P \ast q \ast wp x R s + (1 - P) \ast wp y R s) \sqsubseteq c \leq\)
\(P \ast q \ast wp x (\lambda s. a \ast Q s + b \ast R s \sqsubseteq c) s +\)
\((1 - P) \ast wp y (\lambda s. a \ast Q s + b \ast R s \sqsubseteq c) s .\)

qed

**Lemma** sublinear-wp-DC:

fixes \(x::'a\) 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 \(R::'a\) prog and \(Q::'a\) prog and \(s::'a\) prog and \(a::real\) and \(b::real\) and \(c::real\)
assume slR: sound \(R\) and sQ: sound \(Q\)
and nna: \(0 \leq a\) and nnb: \(0 \leq b\) and nnc: \(0 \leq c\)
from nna nnb
have \(a \ast \min (wp x Q s) (wp y Q s) +\)
\(b \ast \min (wp x R s) (wp y R s) \sqsubseteq c =\)
\(\min (a \ast wp x Q s) (a \ast wp y Q s) +\)
\(\min (b \ast wp x R s) (b \ast wp y R s) \sqsubseteq c\)
by (simp add:min-distrib)
also
have \(\ldots \leq \min (a \ast wp x Q s + b \ast wp x R s)\)
\((a \ast wp y Q s + b \ast wp y R s) \sqsubseteq c\)
by (auto intro!:tminus-left-mono)
also
have \(\ldots = \min (a \ast wp x Q s + b \ast wp x R s \sqsubseteq c)\)
\((a \ast wp y Q s + b \ast wp y R s \sqsubseteq c)\)
by (rule min-tminus-distrib)
also \{
from slx sR nna nnb nnc
have \(a \ast wp x Q s + b \ast wp x R s \sqsubseteq c \leq\)
\(wp x (\lambda s. a \ast Q s + b \ast R s \sqsubseteq c) s\)
by (blast)
moreover
from sly sQ sR nna nnb nnc
have \(a \ast wp y Q s + b \ast wp y R s \sqsubseteq c \leq\)
\(wp y (\lambda s. a \ast Q s + b \ast R s \sqsubseteq c) s\)
by (blast)
ultimately
have \(\min (a \ast wp x Q s + b \ast wp x R s \sqsubseteq c)\)
\((a \ast wp y Q s + b \ast wp y R s \sqsubseteq c) \leq\)
\(\min (wp x (\lambda s. a \ast Q s + b \ast R s \sqsubseteq c) s)\)
\((wp y (\lambda s. a \ast Q s + b \ast R s \sqsubseteq c) s)\)
by (auto)
As for continuity, we insist on a finite support.

**Lemma** \textbf{sublinear-wp-SetPC}: 
fixes \( p::a \Rightarrow 's \) prog
assumes slp: \( \forall s. \forall a a < sups (P s) \Longrightarrow \) sublinear \( (wp (p a)) \)
and sum: \( \forall s. (\sum s. a < sups (P s) \cdot P s a) \leq 1 \)
and nnP: \( \forall s. 0 \leq P s a \)
and fin: \( \forall s. \) finite \( (supp (P s)) \)
shows \( \) sublinear \( (wp (SetPC p P)) \)
proof (rule sublinearI, simp add:wp-egal)
fix \( R::s \Rightarrow real \) and \( Q::s \Rightarrow real \) and \( s::s \)
and \( a::real \) and \( b::real \) and \( c::real \)
assume sk: \( \) sound \( R \) and \( sQ: \) sound \( Q \)
and nna: \( 0 \leq a \) and \( nnb: 0 \leq b \) and \( nnc: 0 \leq c \)

have \( a < (\sum s. a < sups (P s) \cdot P s a' < wp (p a') Q s) + \)
\( b < (\sum s. a < sups (P s) \cdot P s a' < wp (p a') R s) \otimes c = \)
\( (\sum s. a < sups (P s) \cdot P s a' < (a < wp (p a') Q s + b < wp (p a') R s) \otimes c < \)
by (simp add:field-simps sum-distrib-left sum_distrib)
also have \( \leq \)
\( (\sum s. a < sups (P s) \cdot P s a' < (a < wp (p a') Q s + b < wp (p a') R s)) \otimes c < \)
proof (rule tminus-right-antimono)
have \( (\sum s. a < sups (P s) \cdot P s a' < c) \leq (\sum s. a < sups (P s) \cdot P s a') < c < \)
by (simp add:sum_distrib-right)
also from sum and nnc have \( \leq 1 * c < \)
by (rule mult_right_mono)
finally show \( (\sum s. a < sups (P s) \cdot P s a' < c) \leq c < \) by (simp)
qed
also from fin
have \( \leq (\sum s. a < sups (P s) \cdot P s a' < (a < wp (p a') Q s + b < wp (p a') R s) \otimes P s a' < c) < \)
by (blast intro:tminus-sum-mono)
also have \( = (\sum s. a < sups (P s) \cdot P s a' < (a < wp (p a') Q s + b < wp (p a') R s) \otimes c) < \)
by (simp add:nnP tminus_left_distrib)
also \{ 
from slp sQ sR nna nnb nnc
have \( (\sum s. a < sups (P s) \Longrightarrow a < wp (p a') Q s + b < wp (p a') R s \otimes c \leq \)
\( wp (p a') (\lambda s. a < Q s + b < R s \otimes c) s < \)
by (blast)
with nnP
have \( (\sum s. a < sups (P s) \cdot P s a' < (a < wp (p a') Q s + b < wp (p a') R s \otimes c)) < \)
\leq \)
\( \)
4.5. SUBLINEARITY

\[(\sum a' \in \text{supp} (P s), P s a' \ast \text{wp} (p a')) (\lambda s. a \ast Q s + b \ast R s \odot c) s)\]

by (\text{blast intro:sum-mono mult-left-mono})

\}

finally

show \(a \ast (\sum a' \in \text{supp} (P s), P s a' \ast \text{wp} (p a')) Q s\)

\(b \ast (\sum a' \in \text{supp} (P s), P s a' \ast \text{wp} (p a')) R s \odot c \leq\)

\((\sum a' \in \text{supp} (P s), P s a' \ast \text{wp} (p a')) (\lambda s. a \ast Q s + b \ast R s \odot c) s)\).

qed

\textbf{lemma} \textit{sublinear-wp-SetDC:}

\textbf{fixes} \(p::'a \Rightarrow 's \text{ prog}\)

\textbf{assumes} \(\text{slp: } \bigwedge s \ a \ a \in S \ s \Rightarrow \text{sublinear} (\text{wp} (p a))\)

\textbf{and} \(\text{hp: } \bigwedge s \ a \ a \in S \ s \Rightarrow \text{healthy} (\text{wp} (p a))\)

\textbf{and} \(\text{nc: } \bigwedge s \ S \ s \neq \{\}\)

\textbf{shows} \textit{sublinear} (\text{wp} (SetDC p S))

\textbf{proof}\[(\text{rule sublinearI, simp add:wp-eval, rule cInf-greatest})\]

\textbf{fix} \(P::'s \Rightarrow \text{real} \text{ and } Q::'s \Rightarrow \text{real} \text{ and } s::'s \text{ and } x y\)

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

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

\textbf{and} \(\text{nnn: } 0 \leq a \text{ and } \text{nnb: } 0 \leq b \text{ and } \text{nnc: } 0 \leq c\)

\textbf{from} \(\text{ne show} (\lambda x. \text{wp} (p x) (\lambda s. a \ast P s + b \ast Q s \odot c) s) \mid S s \neq \{\}\) by (\text{auto})

\textbf{asssume} \(\text{yin: } y \in (\lambda x. \text{wp} (p x) (\lambda s. a \ast P s + b \ast Q s \odot c) s) \mid S s\)

\textbf{then obtain} \(x \textbf{ where} \ xin: x \in S s \textbf{ and} \ \text{rwy: } y = \text{wp} (p x) (\lambda s. a \ast P s + b \ast Q s \odot c) s\)

by (\text{auto})

\textbf{from} \(\text{zxin hp sP nna}
\textbf{have} a \ast \text{Inf} ((\lambda a. \text{wp} (p a) P s) \mid S s) \leq a \ast \text{wp} (p x) P s\)

by (\text{intro mult-left-mono[of cInf-lower] bdd-belowI[where } m=0], \text{ blast+})

\textbf{moreover from} \(\text{zxin hp sQ nnb}
\textbf{have} b \ast \text{Inf} ((\lambda a. \text{wp} (p a) Q s) \mid S s) \leq b \ast \text{wp} (p x) Q s\)

by (\text{intro mult-left-mono[of cInf-lower] bdd-belowI[where } m=0], \text{ blast+})

\textbf{ultimately}

\textbf{have} a \ast \text{Inf} ((\lambda a. \text{wp} (p a) P s) \mid S s) +

\(b \ast \text{Inf} ((\lambda a. \text{wp} (p a) Q s) \mid S s) \odot c \leq\)

\(a \ast \text{wp} (p x) P s + b \ast \text{wp} (p x) Q s \odot c\)

by (\text{blast intro:tninuss-left-mono add-mono})

\textbf{also from} \(\text{zxin slP sQ nna nmb nnc}
\textbf{have} \ldots \leq \text{wp} (p x) (\lambda s. a \ast P s + b \ast Q s \odot c) s\)

by (\text{blast})

\textbf{finally show} \(a \ast \text{Inf} ((\lambda a. \text{wp} (p a) P s) \mid S s) + b \ast \text{Inf} ((\lambda a. \text{wp} (p a) Q s) \mid S s) \odot c \leq y\)

by (\text{simp add:rwy})

\textbf{qed}
lemma sublinear-wp-Embed:
\[\text{sublinear } t \implies \text{sublinear } (wp \ (\text{Embed } t))\]
\[\text{by } (\text{simp add: wp-evl})\]

lemma sublinear-wp-repeat:
\[\text{[ sublinear } (wp \ p); \text{ healthy } (wp \ p) \ ] \implies \text{sublinear } (wp \ (\text{repeat } n \ p))\]
\[\text{by } (\text{induct } n, \text{ simp-all add: sublinear-wp-Seq} \text{ sublinear-wp-Skip} \text{ healthy-wp-repeat})\]

lemma sublinear-wp-Bind:
\[\text{[ \Prod s. sublinear } (wp \ (a \ f s)) \ ] \implies \text{sublinear } (wp \ (\text{Bind } f a))\]
\[\text{by } (\text{rule sublinearI, simp add: wp-evl, auto})\]

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:
\[\text{fixes body :: }'s \ \text{prog} \ \text{assumes sdb: sub-distrib } (wp \ \text{body}) \ \text{and hb: healthy } (wp \ \text{body}) \ \text{and nhb: nearly-healthy } (wp \ \text{body}) \ \text{shows} \ \text{sub-distrib } (wp \ (do \ G \rightarrow \text{body} \ od))\]
\[\text{proof} – \ \text{have } \forall P \ s. \ \text{sound } P \rightarrow \wp (\text{do } G \rightarrow \text{body} \ od) P s \ominus 1 \leq \wp (\text{do } G \rightarrow \text{body} \ od) (\lambda s. \ P s \ominus 1) s\]
\[\text{proof} (\text{rule loop-induct}[OF hb nhb], \text{safe})\]
\[\text{fix } S :: (′s \ \text{trans} \times ′s \ \text{trans}) \set \text{and P :: } ′s \ \text{expect} \ \text{and} s :: ′s \)
\[\text{assume saS: } \forall x \in S. \ \forall P \ s. \ \text{sound } P \rightarrow \text{fst } x P s \ominus 1 \leq \text{fst } x (\lambda s. \ P s \ominus 1) s \]
\[\text{and sP: sound } P \]
\[\text{and fS: } \forall x \in S. \ \text{feasible } (\text{fst } x)\]
\[\text{from sP have sPm: sound } (\lambda s. \ P s \ominus 1) \ \text{by } (\text{auto intro:tminus-sound})\]
\[\text{have mnSup: } \exists s. \ 0 \leq \text{Sup-trans } (\text{fst } ^t S) (\lambda s. \ P s \ominus 1) s\]
\[\text{proof} (\text{cases } S = \{\}, \ \text{simp add: Sup-trans-def Sup-exp-def})\]
\[\text{fix s}\]
\[\text{assume } S \neq \{\}\]
\[\text{then obtain } x \ \text{where xin: } x \in S \ \text{by } (\text{auto})\]
\[\text{with fS sPm have } 0 \leq \text{fst } x (\lambda s. \ P s \ominus 1) s \ \text{by } (\text{auto})\]
\[\text{also from xin fS sPm have } ... \leq \text{Sup-trans } (\text{fst } ^t S) (\lambda s. \ P s \ominus 1) s\]
\[\text{by } (\text{auto intro: le-funD[OF Sup-trans-upper2]})\]
\[\text{finally show } ?thesis s .\]
\[\text{qed}\]
\[\text{have } \prod x s. \ \text{fst } x P s \leq (\text{fst } x P s \ominus 1) + 1 \ \text{by } (\text{simp add: tminus-def})\]
\[\text{also from saS sP}\]
\[\text{have } \prod x s. \ x \in S \implies (\text{fst } x P s \ominus 1) + 1 \leq \text{fst } x (\lambda s. \ P s \ominus 1) s + 1\]
\[\text{by } (\text{auto intro:add-right-mono})\]
\[\text{also } \{\]
4.5. **Sublinearity**

from \(sP\) have **sound** \((\lambda s. P s \ominus 1)\) **by** \(\text{auto intro:tmminus-sound}\)

with \(fS\) have \(\forall x. x \in S \Rightarrow \text{fst } x (\lambda s. P s \ominus 1) s + 1 \leq \text{Sup-trans} (\text{fst } ' S) (\lambda s. P s \ominus 1) s + 1\)

**by** \(\text{blast intro: add-right-mono le-funD}[\text{OF Sup-trans-upper2}]\)

} finally have le: \(\forall s. \forall x \in S. \text{fst } x P s \leq \text{Sup-trans} (\text{fst } ' S) (\lambda s. P s \ominus 1) s + 1\)

**by** \(\text{auto}\)

moreover from \(\text{nnSup}\) have **nn**: \(\forall s. 0 \leq \text{Sup-trans} (\text{fst } ' S) (\lambda s. P s \ominus 1) s + 1\)

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

ultimately have leSup: \(\text{Sup-trans} (\text{fst } ' S) P s \ominus 1 \leq \text{Sup-trans} (\text{fst } ' S) (\lambda s. P s \ominus 1) s\)

**proof** (cases \(\text{Sup-trans} (\text{fst } ' S) P s \leq 1\), simp-all add:nnSup)

from leSup have **Sup-trans** \((\text{fst } ' S) P s - 1 \leq \text{Sup-trans} (\text{fst } ' S) (\lambda s. P s \ominus 1) s + 1 - 1\)

**by** \(\text{auto}\)

thus \(\text{Sup-trans} (\text{fst } ' S) P s - 1 \leq \text{Sup-trans} (\text{fst } ' S) (\lambda s. P s \ominus 1) s\)

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

qed

next

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

assume IH: \(\forall P s. \text{sound } P \Rightarrow t P s \ominus 1 \leq t (\lambda a. P a \ominus 1) s\)

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

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

from \(sP\) have **sound** \((\lambda s. P s \ominus 1)\) **by** \(\text{auto intro:tmminus-sound}\)

with \(ft\) have \(s2::\text{sound } (t (\lambda s. P s \ominus 1))\) **by** \(\text{auto}\)

from \(sP ft\) have **sound** \((t P)\) **by** \(\text{auto}\)

**hence** \(s3::\text{sound } (\lambda s. t P s \ominus 1)\) **by** \(\text{auto intro:tmminus-sound}\)

show \(\text{wp} (\text{body } :: \text{Embed } t _G \text{ Skip}) P s \ominus 1 \leq \text{wp} (\text{body } :: \text{Embed } t _G \text{ Skip}) (\lambda a. P a \ominus 1) s\)

**proof** (simp add:wp-eval)

have \(\{ \text{wp body } (t P) s + (1 - \text{\text{\text{G}}}) * P s \ominus 1 = \text{\text{\text{G}}}) * \text{wp body } (t P) s + (1 - \text{\text{\text{G}}}) * P s \ominus (\text{\text{\text{G}}}) * (1 - \text{\text{\text{G}}}) s\}\)

**by** (simp)

also have \(\{ \text{wp body } (t P) s \ominus \text{\text{\text{G}}}) \}

**by** (rule tminus-add-mono)

also have \(\{ \text{wp body } (t P) s \ominus 1 = (1 - \text{\text{\text{G}}}) * (P s \ominus 1)\}

**by** (simp add:tminus-left-distrib)

also \{

from \(ft sP\) have **wp body** \((t P) s \ominus 1 \leq \text{wp body } (\lambda s. t P s \ominus 1) s\)

**by** \(\text{auto intro: sub-distribD}[\text{OF sdb}]\)

also \{


from IH sP have $\lambda s. t\ P\ s \odot 1 \vdash t\ (\lambda s. P\ s \odot 1)$ by (auto)
with sP $f t\ s2\ s3$ have $\text{wp body}\ (\lambda s. t\ P\ s \odot 1)\ s \leq \text{wp body}\ (t\ (\lambda s. P\ s \odot 1))$
by (blast intro:le-funD[OF mono-transD, OF healthy-monoD, OF hb])
}
finally have $\langle G \rangle\ s \ast (\text{wp body}\ (t\ P)\ s \odot 1) + (1 - \langle G \rangle\ s) * (P\ s \odot 1) \leq$
$\langle G \rangle\ s \ast \text{wp body}\ (t\ (\lambda s. P\ s \odot 1))\ s + (1 - \langle G \rangle\ s) * (P\ s \odot 1)$
by (auto intro:add-right-mono mult-left-mono)
}
finally show $\langle G \rangle\ s \ast \text{wp body}\ (t\ P)\ s \odot 1 \leq$
$\langle G \rangle\ s \ast \text{wp body}\ (t\ (\lambda s. P\ s \odot 1))\ s + (1 - \langle G \rangle\ s) * (P\ s \odot 1)$.
qed
next
fix $t\ t':\ s\ \text{trans and } P::'s\ \text{expect and } s::'s$
assume IH: $\forall P\ s.\ \text{sound}\ P\ \rightarrow\ t\ P\ s \odot 1 \leq t\ (\lambda a. P\ a \odot 1)\ s$
and eq: equiv-trans $t\ t'$ and sP: sound $P$
from sP have $t'\ P\ s \odot 1 = t\ P\ s \odot 1$ by (simp add:equiv-transD[OF eq])
also from sP IH have ... $\leq t\ (\lambda s. P\ s \odot 1)\ s$ by (auto)
also {
from sP have sound $(\lambda s. P\ s \odot 1)$ by (simp add:tminus-sound)
hence $t\ (\lambda s. P\ s \odot 1)\ s = t'\ (\lambda s. P\ s \odot 1)\ s$ by (simp add:equiv-transD[OF eq])
}
finally show $t'\ P\ s \odot 1 \leq t'\ (\lambda s. P\ s \odot 1)\ s$.
qed
thus $\text{thesis by (auto intro!:sub-distribI)}$
qed

For sub-additivity, we again use the limit-of-iterates characterisation. Firstly, all iterates are sublinear:

**Lemma** sublinear-iterates:

- **Assumes**
  - $\text{hb: healthy (wp body)}$
  - and $\text{sb: sublinear (wp body)}$
- **Shows** sublinear (iterates body $G\ i$)
- by (induct $i$, auto intro!:sublinear-wp-PC sublinear-wp-Seq sublinear-wp-Skip sublinear-wp-Embed
  - **Assms** healthy-intros iterates-healthy)

From this, sub-additivity follows for the limit (i.e. the loop), by appealing to the property at all steps.

**Lemma** sub-add-wp-loop:

- **Fixes** $\text{body::'}s\ \text{prog}$
- **Assumes**
  - $\text{sb: sublinear (wp body)}$
  - and $\text{cb: bd-cts (wp body)}$
  - and $\text{hwp: healthy (wp body)}$
- **Shows** sub-add (wp (do $G \rightarrow\ \text{body od}$))

**Proof**

- **Fix** $P\ Q::'s\ \text{expect and } s::'s$
assume $sP$: sound $P$ and $sQ$: sound $Q$

from $hwp \ cb \ sP$ have $(\lambda i.\ iterates\ body\ G\ i\ P\ s) \rightarrow\ wp\ do\ G\ \rightarrow\ body\ od\ P\ s$
by (rule loop-iterates)
moreover
from $hwp \ cb \ sQ$ have $(\lambda i.\ iterates\ body\ G\ i\ Q\ s) \rightarrow\ wp\ do\ G\ \rightarrow\ body\ od\ Q\ s$
by (rule loop-iterates)
ultimately
have $(\lambda i.\ iterates\ body\ G\ i\ P\ s + \ iterates\ body\ G\ i\ Q\ s) \rightarrow\ wp\ do\ G\ \rightarrow\ body\ od\ P\ s + wp\ do\ G\ \rightarrow\ body\ od\ Q\ s$
by (rule tendsto-add)
moreover {
  from $\text{sublinear-subadd}[OF sublinear-iterates, OF hwp sb, OF healthy-feasibleD[OF iterates-healthy, OF hwp]]\ sP\ sQ$
  have $(\lambda i.\ iterates\ body\ G\ i\ P\ s + \ iterates\ body\ G\ i\ Q\ s) \leq \ iterates\ body\ G\ i\ (\lambda s.\ P\ s + Q\ s)\ s$
  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 $(\lambda i.\ iterates\ body\ G\ i\ (\lambda s.\ P\ s + Q\ s)\ s) \rightarrow\ wp\ do\ G\ \rightarrow\ body\ od\ (\lambda s.\ P\ s + Q\ s)\ s$
  by (blast intro: loop-iterates)
}
ultimately
show $wp\ do\ G\ \rightarrow\ body\ od\ P\ s + wp\ do\ G\ \rightarrow\ body\ od\ Q\ s \leq wp\ do\ G\ \rightarrow\ body\ od\ (\lambda s.\ P\ s + Q\ s)\ s$
by (blast intro: LIMSEQ-le)
qed

lemma sublinear-wp-loop:
  fixes body::’s prog
  assumes $hb$: healthy $(wp\ body)$
  and $nhb$: nearly-healthy $(wlp\ body)$
  and $sb$: sublinear $(wp\ body)$
  and $cb$: bd-cts $(wp\ body)$
  shows sublinear $(wp\ (do\ G\ \rightarrow\ body\ od))$
  using sublinear-sub-distrib[OF $sb$] sublinear-subadd[OF $sb$]
  $hb$ healthy-feasibleD[OF $hb$]
  by (iprover intro: sd-sa-sublinear[OF - - healthy-wp-loop[OF $hb$]]
  sub-distrib-wp-loop sub-add-wp-loop assms)

lemmas sublinear-intros =
sublinear-wp-Abort
sublinear-wp-Skip
sublinear-wp-Apply
sublinear-wp-Seq
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)

wp Abort is not additive.

lemma additive-wp-Skip:
  additive (wp (Skip))
by (auto simp: wp-eval)

lemma additive-wp-Apply:
  additive (wp (Apply f))
by (auto simp: wp-eval)

lemma additive-wp-Seq:
  fixes a::'s prog
  assumes adda: additive (wp a)
  and addb: additive (wp b)
  and wb: well-def b
  shows additive (wp (a ;; b))
proof (rule additiveI, unfold wp-eval o-def)
  fix P::'s ⇒ real and Q::'s ⇒ real and s::'s
  assume sP: sound P and sQ: sound Q

  note hb = well-def-wp-healthy[OF wb]

  from addb sP sQ
  have wp b (λs. P s + Q s) = (λs. wp b P s + wp b Q s)
4.6. DETERMINISM

by (blast dest: additiveD)
with adda sP sQ hb
show wp a (wp b (λs. P s + Q s)) s =
wp a (wp b P) s + (wp a (wp b Q)) s
by (auto intro: fun-cong [OF additiveD])
qed

lemma additive-wp-PC:
[ additive (wp a); additive (wp b) ] ⇒ additive (wp (a ⊕ b))
by (rule additiveI, simp add: additiveD field-simps wp-eval)

DC is not additive.

lemma additive-wp-SetPC:
[ ∀x s. x ∈ supp (p s) ⇒ additive (wp (a x)); ∃s. finite (supp (p s)) ] ⇒
additive (wp (SetPC a p))
by (rule additiveI,
    simp add: wp-eval additiveD distrib-left sum distr)

lemma additive-wp-Bind:
[ ∀x. additive (wp (a (f x))) ] ⇒ additive (wp (Bind f a))
by (simp add: wp-eval additive-def)

lemma additive-wp-Embed:
[ additive t ] ⇒ additive (wp (Embed t))
by (simp add: wp-eval)

lemma additive-wp-repeat:
additive (wp a) ⇒ well-def a ⇒ additive (wp (repeat n a))
by (induct n, auto simp: additive-wp-Skip intro: additive-wp-Seq wd-intros)

lemmas fa-intros =
additive-wp-Abort additive-wp-Skip
additive-wp-Apply additive-wp-Seq
additive-wp-PC   additive-wp-SetPC
additive-wp-Bind additive-wp-Embed
additive-wp-repeat

4.6.2 Maximality

lemma max-wp-Skip:
maximal (wp Skip)
by (simp add: maximal-def wp-eval)

lemma max-wp-Apply:
maximal (wp (Apply f))
by (auto simp: wp-eval o-def)

lemma max-wp-Seq:
[ maximal (wp a); maximal (wp b) ] ⇒ maximal (wp (a ;; b))
lemma max-wp-PC:
\[ \maximal(\wp a) ; \maximal(\wp b) \implies \maximal(\wp (a \oplus b)) \]
by (rule maximalI, simp add: maximalD field-simps wp-eval)

lemma max-wp-DC:
\[ \maximal(\wp a) ; \maximal(\wp b) \implies \maximal(\wp (a \sqcap b)) \]
by (rule maximalI, simp add: maximalD)

lemma max-wp-SetPC:
\[ \bigwedge s a. a \in \text{supp}(P s) = \implies \maximal(\wp (p a)) ; \bigwedge s. (\sum a \in \text{supp}(P s). P s a) = I \] 
\[ \implies \maximal(\wp (\text{SetPC} p P)) \]
by (auto simp:maximalD wp-def SetPC-def sum-distrib-right[symmetric])

lemma max-wp-SetDC:
fixes p :: 'a ⇒ 's prog
assumes mp: \[ \bigwedge s a. a \in S s \implies \maximal(\wp (p a)) \]
and ne: \[ \bigwedge s. S s \neq {} \]
shows \[ \maximal(\wp (\text{SetDC} p S)) \]
proof (rule maximalI, rule ext, unfold wp-eval)
fix c::real and s:'s
assume 0 ≤ c
hence Inf ((λa. wp (p a) (λ-. c) s) ' S s) = Inf ((λ-. c) ' S s)
  using mp by (simp add:maximalD cong:image-cong)
also { 
  from ne obtain a where a ∈ S s by blast
  hence Inf ((λ-. c) ' S s) = c
    by (auto simp add: image-constant-cong cong del: INF-cong-simp)
}
finally show Inf ((λa. wp (p a) (λ-. c) s) ' S s) = c .
qed

lemma max-wp-Embed:
\[ \maximal t \implies \maximal(\wp (\text{Embed} t)) \]
by (simp add:wp-eval)

lemma max-wp-repeat:
\[ \maximal(\wp a) \implies \maximal(\wp (\text{repeat} n a)) \]
by (induct n, simp-all add: max-wp-Skip max-wp-Seq)

lemma max-wp-Bind:
assumes ma: \[ \bigwedge s. \maximal(\wp (a \ f s)) \]
shows \[ \maximal(\wp (\text{Bind} f a)) \]
proof (rule maximalI, rule ext, simp add:wp-eval)
fix c::real and s
assume 0 ≤ c
with ma have \[ \wp (a \ f s) (λ-. c) = (λ-. c) \]
  by (blast)
4.6. DETERMINISM

thus \( wp \ (a \ (f \ s)) \ (\lambda s. c) \ s = c \) by(auto)
qed

lemmas max-intros =
  max-wp-Skip \ max-wp-Apply
  max-wp-Seq \ max-wp-PC
  max-wp-DC \ max-wp-SetPC
  max-wp-SetDC \ max-wp-Embed
  max-wp-Bind \ max-wp-repeat

A healthy transformer that terminates is maximal.

lemma healthy-term-max:
  assumes ht: healthy \( t \)
  and trm: \( \lambda s. \ 1 \vdash t \ (\lambda s. \ 1) \)
  shows maximal \( t \)
proof(intro maximalI ext)
  fix \( c::real \) 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:
\( (\forall x. x \in \text{supp} \ (p \ s) \implies \text{determ} \ (wp \ (a \ x))) \implies \) 
\( (\forall s. \text{finite} \ (\text{supp} \ (p \ s)) \implies \) 
\( (\forall s. \text{sum} \ (p \ s) \ (\text{supp} \ (p \ s)) = 1) \implies \) 
\( \text{determ} \ (wp \ (\text{SetPC} \ a \ p)) \) 
\text{by} (\text{intro determI fa-intros max-intros, auto}) 

\textbf{lemma det-wp-Bind:} 
\( (\forall x. \text{determ} \ (wp \ (a \ (f \ x))) \implies \text{determ} \ (wp \ (\text{Bind} \ f \ a)) \) 
\text{by} (\text{intro determI fa-intros max-intros, auto}) 

\textbf{lemma det-wp-Embed:} 
\( \text{determ} \ t \implies \text{determ} \ (wp \ (\text{Embed} \ t)) \) 
\text{by} (\text{simp add: wp-eval}) 

\textbf{lemma det-wp-repeat:} 
\( \text{determ} \ (wp \ a) \implies \text{well-def} \ a \implies \text{determ} \ (wp \ (\text{repeat} \ n \ a)) \) 
\text{by} (\text{intro determI fa-intros max-intros, auto}) 

\textbf{lemmas determ-intros =} 
\textbf{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 

\section*{4.7 Well-Defined Programs.} 

\textbf{theory WellDefined imports} 
\textbf{Healthiness Sublinearity LoopInduction} 
\textbf{begin} 

\text{The definition of a well-defined program collects the various notions of healthiness and well-behavedness that we have so far established: healthiness of the strict and liberal transformers, continuity and sublinearity of the strict transformers, and two new properties. These are that the strict transformer always lies below the liberal one (i.e. that it is at least as \textit{strict}, recalling the standard embedding of a predicate), and that expectation conjunction is distributed between them in a particular manner, which will be crucial in establishing the loop rules.} 

\subsection*{4.7.1 Strict Implies Liberal} 

\text{This establishes the first connection between the strict and liberal interpretations (\textit{wp} and \textit{wlp}).}
4.7. WELL-DEFINED PROGRAMS.

definition
wp-under-wlp :: 's prog ⇒ bool

where
wp-under-wlp prog ≡ ∀ P. unitary P → wp prog P ⊢ wlp prog P

lemma wp-under-wlpI[intro]:
[ ∀P. unitary P → wp prog P ⊢ wlp prog P ] ⇒ wp-under-wlp prog
unfolding wp-under-wlp-def by(simp)

lemma wp-under-wlpD[dest]:
[ wp-under-wlp prog; unitary P ] ⇒ wp prog P ⊢ wlp prog P
unfolding wp-under-wlp-def by(simp)

lemma wp-under-le-trans:
wp-under-wlp a ⇒ le-utrans (wp a) (wlp a)
by(blast)

lemma wp-under-wlp-Abort:
wp-under-wlp Abort
by(rule wp-under-wlpI, unfold wp-eval, auto)

lemma wp-under-wlp-Skip:
wp-under-wlp Skip
by(rule wp-under-wlpI, unfold wp-eval, blast)

lemma wp-under-wlp-Apply:
wp-under-wlp (Apply f)
by(auto simp:wp-eval)

lemma wp-under-wlp-Seq:
assumes h-wlp-a: nearly-healthy (wlp a)
and h-wlp-b: healthy (wp b)
and h-wlp-b: nearly-healthy (wlp b)
and wp-a-a: wp-under-wlp a
and wp-a-b: wp-under-wlp b
shows wp-under-wlp (a ;; b)
proof(rule wp-under-wlpI, unfold wp-eval o-def)
fix P::'a ⇒ real assume uP: unitary P
with h-wlp-b have unitary (wp b P) by(blast)
with wp-u-a have wp a (wp b P) ⊢ wlp a (wp b P) by(auto)
also { from wp-u-b and uP have wp b P ⊢ wlp b P by(blast)
with h-wlp-a and h-wlp-b and h-wlp-b and uP
have wlp a (wp b P) ⊌ wlp a (wp b P)
by(blast intro:nearly-healthy-monoD[OF h-wlp-a])
}
finally show wp a (wp b P) ⊢ wlp a (wp b P) .
qed
lemma \textit{wp-under-wlp-PC}:
assumes \( h\text{-wp}\text{-a} \): \text{healthy} \((wp\text{ a})\)
and \( h\text{-wp}\text{-a} \): \text{nearly-healthy} \((wlp\text{ a})\)
and \( h\text{-wp}\text{-b} \): \text{healthy} \((wp\text{ b})\)
and \( h\text{-wp}\text{-b} \): \text{nearly-healthy} \((wlp\text{ b})\)
and \( \text{wp}\text{-u}\text{-a} \): \text{wp-under-wlp a}
and \( \text{wp}\text{-u}\text{-b} \): \text{wp-under-wlp b}
and \( uP \): \text{unitary P}
shows \( \text{wp-under-wlp} (a \oplus b) \)
proof\( (\text{rule wp-under-wlpI}, \text{unfold wp-eval}, \text{rule le-funI})\)
fix \( Q::'a \Rightarrow \text{real} \text{ and } s\)
assume \( uQ \): \text{unitary Q}
from \( uP \) have \( P\ s \leq 1 \) by\( (\text{blast})\)
hence \( 0 \leq 1 - P\ s \) by\( (\text{simp})\)
moreover
from \( uQ \) and \( \text{wp}\text{-u}\text{-b} \) have \( \text{wp b Q s} \leq \text{wlp b Q s} \) by\( (\text{blast})\)
ultimately
have \( (1 - P\ s)*\text{wp b Q s} \leq (1 - P\ s)*\text{wlp b Q s} \)
by\( (\text{blast intro:mult-left-mono})\)
moreover \{ 
from \( uQ \) and \( \text{wp}\text{-u}\text{-a} \) have \( \text{wp a Q s} \leq \text{wlp a Q s} \) by\( (\text{blast})\)
with \( uP \) have \( P\ s*\text{wp a Q s} \leq P\ s*\text{wlp a Q s} \)
by\( (\text{blast intro:mult-left-mono})\)
\}
ultimately
show \( P\ s*\text{wp a Q s} + (1 - P\ s)*\text{wp b Q s} \leq \)
\( P\ s*\text{wlp a Q s} + (1 - P\ s)*\text{wlp b Q s} \)
by\( (\text{blast intro: add-mono})\)
qed

lemma \textit{wp-under-wlp-DC}:
assumes \( \text{wp}\text{-u}\text{-a} \): \text{wp-under-wlp a}
and \( \text{wp}\text{-u}\text{-b} \): \text{wp-under-wlp b}
shows \( \text{wp}\text{-under-wlp} (a \prod b) \)
proof\( (\text{rule wp-under-wlpI}, \text{unfold wp-eval}, \text{rule le-funI})\)
fix \( Q::'a \Rightarrow \text{real} \text{ and } s\)
assume \( uQ \): \text{unitary Q}
from \( \text{wp}\text{-u}\text{-a} \ uQ \) have \( \text{wp a Q s} \leq \text{wlp a Q s} \) by\( (\text{blast})\)
moreover
from \( \text{wp}\text{-u}\text{-b} \ uQ \) have \( \text{wp b Q s} \leq \text{wlp b Q s} \) by\( (\text{blast})\)
ultimately
show \( \min (\text{wp a Q s}) (\text{wp b Q s}) \leq \min (\text{wlp a Q s}) (\text{wlp b Q s}) \)
by\( (\text{auto})\)
qed

lemma \textit{wp-under-wlp-SetPC}:
assumes \( wp-u-f : \forall s. a \in \text{supp}(P s) \Rightarrow wp-under-wlp\ (f a) \)
and \( nP : \forall s. a \in \text{supp}(P s) \Rightarrow 0 \leq P s a \)
shows \( wp-under-wlp\ (\text{SetPC}\ f\ P) \)

proof (rule wp-under-wlpI, unfold wp-eval, rule le-funI)
fix \( Q : 'a \Rightarrow \text{real} \) and \( s \)
assume \( uQ : \text{unitary}\ Q \)
show \((\sum a \in \text{supp}(P s). P s a \ast wp\ (f a)\ Q s) \leq (\sum a \in \text{supp}(P s). P s a \ast wlp\ (f a)\ Q s)\)
by (auto intro!: sum-mono mult-left-mono)
qed

lemma \( wp-under-wlp-SetDC \):
assumes \( wp-u-f : \forall s. a \in S s \Rightarrow wp-under-wlp\ (f a) \)
and \( hf : \forall s. a \in S s \Rightarrow \text{healthy}\ (wp\ (f a)) \)
and \( nS : \forall s. S s \neq \{\} \)
shows \( wp-under-wlp\ (\text{SetDC}\ f\ S) \)
proof (rule wp-under-wlpI, rule le-funI, unfold wp-eval)
fix \( Q : 'a \Rightarrow \text{real} \) and \( s \)
assume \( uQ : \text{unitary}\ Q \)
show \( \inf ((\lambda a. wp\ (f a)\ Q s) \cdot S s) \leq \inf ((\lambda a. wlp\ (f a)\ Q s) \cdot S s) \)
proof (rule cInf-mono)
from \( nS \) show \( (\lambda a. wlp\ (f a)\ Q s) \cdot S s \neq \{\} \) by (blast)

next

lemma \( wp-under-wlp-Embed \):
\( wp-under-wlp\ (\text{Embed}\ t) \)
by (rule wp-under-wlpI, unfold wp-eval, blast)
lemma wp-under-wlp-loop:
  fixes body::'s prog
  assumes hwp: healthy (wp body)
  and hwlp: nearly-healthy (wlp body)
  and wp-under: wp-under-wlp body
  shows wp-under-wlp (do G → body od)
proof (rule wp-under-wlpI)
  fix P::'s expect
  assume uP: unitary P hence sP: sound P by (auto)
  let ?X Q s = «G» s * wp body Q s + «N G» s * P s
  let ?Y Q s = «G» s * wlp body Q s + «N G» s * P s

  show wp (do G → body od) P ⊢ wlp (do G → body od) P
  proof (simp add: hwp hwlp sP uP wp-Loop1 wlp-Loop1, rule gfp-exp-upperbound)
    from hwp sP have lfp-exp ?X = ?X (lfp-exp ?X)
      by (rule lfp-wp-loop-unfold)
    hence lfp-exp ?X ⊢ ?X (lfp-exp ?X) by (simp)
    also {
      from hwp uP have wp body (lfp-exp ?X) ⊢ wlp body (lfp-exp ?X)
        by (auto intro: wp-under-wlpD[OF wp-under lfp-loop-unitary])
        by (auto intro: add-mono mult-left-mono)
    }
  from hwp uP show unitary (lfp-exp ?X)
    by (auto intro: lfp-loop-unitary)
  qed

lemma wp-under-wlp-repeat:
  [ healthy (wp a); nearly-healthy (wlp a); wp-under-wlp a ] ⇒
  wp-under-wlp (repeat n a)
  by (induct n, auto intro: wp-under-wlp-Skip wp-under-wlp-Seq healthy-intros)

lemma wp-under-wlp-Bind:
  [ f s, wp-under-wlp (a (f s)) ] ⇒ wp-under-wlp (Bind f a)
  unfolding wp-under-wlp-def by (auto simp: wp-eval)

lemmas wp-under-wlp-intros =
  wp-under-wlp-Abort wp-under-wlp-Skip
  wp-under-wlp-Apply wp-under-wlp-Seq
  wp-under-wlp-PC wp-under-wlp-DC
  wp-under-wlp-SetPC wp-under-wlp-SetDC
  wp-under-wlp-Embed wp-under-wlp-loop
  wp-under-wlp-repeat wp-under-wlp-Bind
4.7. WELL-DEFINED PROGRAMS.

4.7.2 Sub-Distributivity of Conjunction

definition
sub-distrib-pconj :: 's prog ⇒ bool

where
sub-distrib-pconj prog ≡
∀ P Q. unitary P → unitary Q →
  wlp prog P & wlp prog Q ⊢ wp prog (P & Q)

lemma sub-distrib-pconjI[intro]:
[∀ P Q. [ unitary P; unitary Q ] ⇒ wp prog P & wp prog Q ⊢ wp prog (P & Q) ]
sub-distrib-pconj prog

unfolding sub-distrib-pconj-def by(simp)

lemma sub-distrib-pconjD[dest]:
∀ P Q. [ sub-distrib-pconj prog; unitary P; unitary Q ] ⇒ wp prog P & wp prog Q ⊢ wp prog (P & Q)

unfolding sub-distrib-pconj-def by(simp)

lemma sdp-Abort:
sub-distrib-pconj Abort
by(rule sub-distrib-pconjI, unfold wp-eval, auto intro:exp-conj-rzero)

lemma sdp-Skip:
sub-distrib-pconj Skip
by(rule sub-distrib-pconjI, simp add:wp-eval)

lemma sdp-Seq:
fixes a and b
assumes sdp-a: sub-distrib-pconj a
  and sdp-b: sub-distrib-pconj b
  and h-wp-a: healthy (wp a)
  and h-wp-b: healthy (wp b)
  and h-wlp-b: nearly-healthy (wlp b)
shows sub-distrib-pconj (a ;; b)
proof(rule sub-distrib-pconjI, unfold wp-eval a-def)
fix P::'a ⇒ real and Q::'a ⇒ real
assume uP: unitary P and uQ: unitary Q

with h-wp-b and h-wlp-b
have wp a (wp b P) & wp a (wp b Q) ⊢ wp a (wp b P & wp b Q)
by(blast intro!:sub-distrib-pconjD[OF sdp-a])
also {from sdp-b and uP and uQ
have wp b P & wp b Q ⊢ wp b (P & Q) by(blast)
with h-wp-a h-wp-b h-wlp-b uP uQ
have wp a (wp b P & wp b Q) ⊢ wp a (wp b (P & Q))
by(blast intro!:mono-transD[OF healthy-monoD, OF h-wp-a] unitary-sound
unitary-intros sound-intros)
lemma sdp-Apply:
sub-distrib-pconj (Apply f)
by (rule sub-distrib-pconjI, simp add: wp-eval)

lemma sdp-DC:
fixes a::'s prog and b
assumes sdp-a: sub-distrib-pconj a
and sdp-b: sub-distrib-pconj b
and h-wp-a: healthy (wp a)
and h-wp-b: nearly-healthy (wlp b)
shows sub-distrib-pconj (a ∩ b)
proof (rule sub-distrib-pconjI, unfold wp-eval, rule le-funI)
fix P::'s ⇒ real and Q::'s ⇒ real and s::'s
assume uP: unitary P and uQ: unitary Q
have ((λs. min (wlp a P s) (wlp b P s)) & &
(λs. min (wp a Q s) (wp b Q s))) s ≤
min (wlp a P s & & wp a Q s) (wlp b P s & & wp b Q s)
unfolding exp-conj-def by (rule min-pconj)
also { have (λs. wlp a P s & & wp a Q s) = wlp a P & & wp a Q
by (simp add: exp-conj-def)
also from sdp-a uP uQ have ... ⊨ wp a (P & & Q)
by (blast dest: sub-distrib-pconjD)
finally have wlp a P s & & wp a Q s ≤ wp a (P & & Q) s
by (rule le-funD)
moreover { have (λs. wlp b P s & & wp b Q s) = wlp b P & & wp b Q
by (simp add: exp-conj-def)
also from sdp-b uP uQ have ... ⊨ wp b (P & & Q)
by (blast)
finally have wlp b P s & & wp b Q s ≤ wp b (P & & Q) s
by (rule le-funD)
}
ultimately have min (wlp a P s & & wp a Q s) (wlp b P s & & wp b Q s) ≤
min (wp a (P & & Q) s) (wp b (P & & Q) s) by (auto)
}
finally show ((λs. min (wlp a P s) (wlp b P s)) & &
(λs. min (wp a Q s) (wp b Q s))) s ≤
min (wp a (P & & Q) s) (wp b (P & & Q) s) .
qed
4.7. WELL-DEFINED PROGRAMS.

lemma sdp-PC:
  fixes a::'s prog and b
assumes sdp-a:  sub-distrib-pconj a
  and sdp-b:  sub-distrib-pconj b
  and h-wp-a: healthy (wp a)
  and h-wp-b: healthy (wp b)
  and h-wlp-b: nearly-healthy (wlp b)
  and uP:  unitary P
shows sub-distrib-pconj (a ⊕ b)
proof (rule sub-distrib-pconjI, unfold wp-eval, rule le-funI)
  fix Q::'s ⇒ real and R::'s ⇒ real and s::'s
  assume uQ:  unitary Q and uR:  unitary R

have nnA: 0 ≤ P s and nnB:  P s ≤ 1
  using uP by auto
note nn = nnA nnB

have ((λs. P s * wlp a Q s + (1 − P s) * wp b Q s) &&
  (λs. P s * wp a R s + (1 − P s) * wp b R s)) s =
  ((P s * wlp a Q s + (1 − P s) * wp b Q s) +
  (P s * wp a R s + (1 − P s) * wp b R s)) ⊕ 1
  by(simp add:exp-conj-def pconj-def)
also have ... = P s *
  (wlp a Q s + wp a R s) +
  (1 − P s) * (wp b Q s + wp b R s) ⊕ 1
  by(simp add:field-simps)
also have ... = P s *
  (wlp a Q s + wp a R s) +
  (1 − P s) * (wp b Q s + wp b R s) ⊕
  (P s + (1 − P s))
  by(simp)
also have ... ≤ (P s *
  (wlp a Q s + wp a R s) ⊕ P s) +
  ((1 − P s) * (wp b Q s + wp b R s) ⊕ (1 − P s))
  by(rule tminus-add-mono)
also have ... = (P s *
  (wlp a Q s + wp a R s ⊕ 1)) +
  ((1 − P s) * (wp b Q s + wp b R s ⊕ 1))
  by(simp add:nn tminus-left-distrib)
also have ... = P s *
  (wlp a Q & & wp a R) s +
  (1 − P s) * ((wp b Q & & wp b R) s)
  by(simp add:exp-conj-def pconj-def)
also { from sdp-a sdp-b uQ uR
  have P s * (wlp a Q & & wp a R) s ≤ P s * wp a (Q & & R) s
  and (1 − P s) * (wp b Q & & wp b R) s ≤ (1 − P s) * wp b (Q & & R) s
  by (simp-all add: entailsD mult-left-mono nn sub-distrib-pconjD)
hence P s *
  (wlp a Q & & wp a R) s +
  (1 − P s) * (wp b Q & & wp b R) s ≤
  P s * wp a (Q & & R) s + (1 − P s) * wp b (Q & & R) s
  by(auto)
}
finally show ((λs. P s * wlp a Q s + (1 − P s) * wp b Q s) &&

\( (\lambda s. P \ s \ \circ \ wp \ a \ R \ s + (1 - P \ s) \ \circ \ wp \ b \ R \ s) \ s \leq \ P \ s \ \circ \ wp \ a \ (Q \ &\& \ R) \ s + (1 - P \ s) \ \circ \ wp \ b \ (Q \ &\& \ R) \ s \).

\text{qed}

\text{lemma sdp-Embed:}

\[ \frac{\text{unitary } P; \text{ unitary } Q}{\Rightarrow \ t \ P \ &\& \ t \ Q \ \vdash \ (P \ &\& \ Q)} \Rightarrow \text{sub-distrib-pconj (Embed } t) \]

\text{by (auto simp: wp-eval)}

\text{lemma sdp-repeat:}

\text{fixes } a ::'s \ prog
\text{assumes } sdp\ a: \text{ sub-distrib-pconj } a
\text{ and } hwp: \text{ healthy (wp } a \text{) and } hwlp: \text{ nearly-healthy (wlp } a \text{)}
\text{shows } \text{ sub-distrib-pconj (repeat } n \ a \text{) (is } ?X \ n)\]

\text{proof } (\text{induct } n)
\text{show } ?X \ 0 \ \text{by} (\text{simp add: sdp-Skip})
\text{fix } n \ \text{assume } IH: \ ?X \ n
\text{show } ?X \ (Suc \ n) \ \text{proof } (\text{rule sub-distrib-pconjI, simp add: wp-eval})

\text{fix } P ::'s \Rightarrow \text{ real and } Q ::'s \Rightarrow \text{ real}
\text{assume } uP: \text{ unitary } P \text{ and } uQ: \text{ unitary } Q
\text{from asms have } hwlp: \text{ nearly-healthy (wlp (repeat } n \ a \text{))}
\text{ and } hwpa: \text{ healthy (wp (repeat } n \ a \text{))}
\text{by (auto intro: healthy-intros)}

\text{from } uP \text{ and } hwlp \text{ have unitary (wlp (repeat } n \ a \text{) } P \text{) by (blast)}
\text{moreover from } uQ \text{ and } hwpa \text{ have unitary (wp (repeat } n \ a \text{) } Q \text{) by (blast)}
\text{ultimately have wp } a \ (wlp \ (repeat } n \ a \text{) } P \text{) } &\& \text{ wp } a \ (wp \ (repeat } n \ a \text{) } Q \text{ by (blast)}
\text{using sdp } a \text{ by (blast)}

\text{also \{}
\text{from hwlp \ have \ nearly-healthy \ (wlp \ (repeat } n \ a \text{)) \ by (rule healthy-intros)}
\text{with } uP \text{ have sound (wp (repeat } n \ a \text{) } P \text{) by (auto)}
\text{moreover from hwp } uQ \text{ have sound (wp (repeat } n \ a \text{) } Q \text{) by (auto intro: healthy-intros)}
\text{ultimately have sound (wp (repeat } n \ a \text{) } P \text{) } &\& \text{ wp (repeat } n \ a \text{) } Q \text{ by (rule exp-conj-sound)}
\text{moreover \{}
\text{from } uP \text{ uQ have sound (P } &\& \text{ Q) by (auto intro: exp-conj-sound)}
\text{with } hwpa \text{ have sound (wp (repeat } n \ a \text{) } (P \ &\& \text{ Q)) by (auto intro: healthy-intros)}
\text{\}}
\text{moreover from } uP \ uQ \text{ IH}
\text{have wp } (\text{repeat } n \ a \text{) } P \text{) } &\& \text{ wp (repeat } n \ a \text{) } Q \text{ by (blast)}
\text{ultimately have wp } a \ (wp \ (repeat } n \ a \text{) } P \text{) } &\& \text{ wp (repeat } n \ a \text{) } Q \text{ by (rule mono-transD [OF healthy-monoD, OF hwp])}
4.7. WELL-DEFINED PROGRAMS.

\[
\begin{align*}
\text{finally show } & \text{ wlp } a \ (\text{wlp } (\text{repeat } n \ a) \ P) \ & \& \ \text{wlp } a \ (\text{wp } (\text{repeat } n \ a) \ Q) \ \vdash \\
& \text{wp } a \ (\text{wp } (\text{repeat } n \ a) \ (P \ & \& \ Q)) .
\end{align*}
\]

\text{qed}

\text{qed}

\text{lemma } \text{sdp-SetPC}:
\begin{itemize}
\item \text{fixes } p::'a \Rightarrow 's \ prog
\item \text{assumes } \text{sdp}: \forall s. \ a \in \text{supp } (P s) \implies \text{sub-distrib-pconj } (p \ a)
\item \text{and } \text{fin}: \forall s. \ \text{finite } (\text{supp } (P s))
\item \text{and } \text{nnp}: \forall s. \ 0 \leq P s \ a
\item \text{and } \text{sub}: \forall s. \ \text{sum } (P s) \ (\text{supp } (P s)) \leq 1
\end{itemize}

\text{shows } \text{sub-distrib-pconj } (\text{SetPC } p \ P)

\text{proof}(\text{rule sub-distrib-pconjI}, \text{ simp add:wp-eval, rule le-funI})
\begin{itemize}
\item \text{fix } Q::'s \Rightarrow \text{real } \text{and } R::'s \Rightarrow \text{real } \text{and } s::'s
\item \text{assume } uQ: \text{unitary } Q \text{ and } uR: \text{unitary } R
\item \text{have } ((\lambda s. \sum a \in \text{supp } (P s). \ P s a * \text{wlp } (p \ a) \ Q s) \ & \& \\
& ((\lambda s. \sum a \in \text{supp } (P s). \ P s a * \text{wp } (p \ a) \ R s) s) = \\
& (\sum a \in \text{supp } (P s). \ P s a * \text{wlp } (p \ a) \ Q s) + \ (\sum a \in \text{supp } (P s). \ P s a * \text{wp } (p \ a) \ R s) \ & \& \ 1
\end{itemize}

\text{by}(\text{simp add:exp-conj-def pconj-def})
\begin{itemize}
\item \text{also have } ... = (\sum a \in \text{supp } (P s). \ P s a * (\text{wlp } (p \ a) \ Q s + \text{wp } (p \ a) \ R s)) \ & \& \ 1
\end{itemize}

\text{by}(\text{simp add: sum.distrib field-simps})
\begin{itemize}
\item \text{also from } \text{sub}
\item \text{have } ... \leq (\sum a \in \text{supp } (P s). \ P s a * (\text{wlp } (p \ a) \ Q s + \text{wp } (p \ a) \ R s)) \ & \& \\
& (\sum a \in \text{supp } (P s). \ P s a)
\end{itemize}

\text{by}(\text{rule tmminus-right-antimono})
\begin{itemize}
\item \text{also from } \text{fin}
\item \text{have } ... = (\sum a \in \text{supp } (P s). \ P s a * (\text{wlp } (p \ a) \ Q s + \text{wp } (p \ a) \ R s)) \ & \& P s a
\end{itemize}

\text{by}(\text{rule tmminus-sum-mono})
\begin{itemize}
\item \text{also from } \text{nnp}
\item \text{have } ... = (\sum a \in \text{supp } (P s). \ P s a * (\text{wlp } (p \ a) \ Q s + \text{wp } (p \ a) \ R s \ & \& 1))
\end{itemize}

\text{by}(\text{simp add:tmminus-left-distrib})
\begin{itemize}
\item \text{also have } ... = (\sum a \in \text{supp } (P s). \ P s a * (\text{wlp } (p \ a) \ Q & \& \text{wp } (p \ a) \ R s) s)
\end{itemize}

\text{by}(\text{simp add:pconj-def exp-conj-def})
\begin{itemize}
\item \text{also }
\item \text{from } \text{sdp } uQ \ aR
\item \text{have } \forall a. \ a \in \text{supp } (P s) \implies \text{wlp } (p \ a) \ Q & \& \text{wp } (p \ a) \ R \vdash \text{wp } (p \ a) \ (Q & \& R)
\end{itemize}

\text{by}(\text{blast intro:sub-distrib-pconjD})

\text{with } \text{nnp}
\begin{itemize}
\item \text{have } (\sum a \in \text{supp } (P s). \ P s a * (\text{wlp } (p \ a) \ Q \ & \& \text{wp } (p \ a) \ R) s) \leq \\
& (\sum a \in \text{supp } (P s). \ P s a * (\text{wp } (p \ a) \ (Q & \& R)) s)
\end{itemize}

\text{by}(\text{blast intro:sum-mono mult-left-mono})
\end{itemize}

\text{finally show } ((\lambda s. \sum a \in \text{supp } (P s). \ P s a * \text{wlp } (p \ a) \ Q s) \ & \& \\
& (\lambda s. \sum a \in \text{supp } (P s). \ P s a * \text{wp } (p \ a) \ R s) s \leq \\
& (\sum a \in \text{supp } (P s). \ P s a * \text{wp } (p \ a) \ (Q & \& R) s) .
\]

\text{qed}
lemma sdp-SetDC:
  fixes p: 'a ⇒ 's prog
  assumes sdp: \( \forall s. a \in S \Rightarrow \text{sub-distrib-pconj} (p a) \)
  and hwp: \( \forall s. a \in S \Rightarrow \text{healthy} (wp (p a)) \)
  and hwlp: \( \forall s. a \in S \Rightarrow \text{nearly-healthy} (wlp (p a)) \)
  and ne: \( \forall s. S \neq \{\} \)
  shows \( \text{sub-distrib-pconj} (\SetDC p S) \)
proof (rule sub-distrib-pconjI, rule le-funI)
  fix P: 's ⇒ real and Q: 's ⇒ real and s:'s
  assume uP: unitary P and uQ: unitary Q
  from uP hwlp have \( \forall a. a \in S \Rightarrow \text{unitary} x \) by (auto)
  hence \( \forall x. x \in (\lambda a. wlp (p a) P) S \Rightarrow \text{unitary} x \) by (auto)
  unfolding wp-eval by (intro cInf-lower bdd-belowI, auto)
moreover {
  from uQ hwp have \( \forall a. a \in S \Rightarrow 0 \leq wp (p a) Q \) by (blast)
  hence \( \forall a. a \in S \Rightarrow wp (\SetDC p S) Q s \leq wp (p a) Q s \)
  unfolding wp-eval by (intro cInf-lower bdd-belowI, auto)
}
ultimately have \( \forall a. a \in S \Rightarrow \text{wp} (\SetDC p S) P s + wp (p a) Q s \leq wp (p a) P \land wp (p a) Q s \)
  by (auto intro: tminus-left-mono add-mono)
also have \( \forall a. wp (p a) P s + wp (p a) Q s \leq wp (\lambda s. P s + Q s) s \)
  by (simp add: exp-conj-def pconj-def)
also from sdp uP uQ
have \( \forall a. a \in S \Rightarrow \ldots a \leq wp (p a) (P \land \land Q) s \)
  by (blast)
also have \( \forall a. \ldots a = wp (p a) (\lambda s. P s + Q s \land I) s \)
  by (simp add: exp-conj-def pconj-def)
finally show \( wp (\lambda s. P s + wlp (\SetDC p S) Q s \leq wp (\lambda s. P s + Q s) s \)
  unfolding exp-conj-def pconj-def wp-eval
  using ne by (blast intro: cInf-greatest)
qed

lemma sdp-Bind:
  \[ \text{sub-distrib-pconj} (p (f s)) \] \( \Rightarrow \text{sub-distrib-pconj} (\Bind f p) \)
unfolding sub-distrib-pconj-def wp-eval exp-conj-def pconj-def
  by (blast)

For loops, we again appeal to our transfinite induction principle, this time taking advantage of the simultaneous treatment of both strict and liberal transformers.
4.7. WELL-DEFINED PROGRAMS.

**Lemma sdp-loop**

**Fixes** `body :: 's prog`

**Assumes** `sdp-body :: sub-distrib-pconj body`

and `hwlp :: nearly-healthy (wp body)`

and `hwp :: healthy (wp body)`

**Shows** `sub-distrib-pconj (do G → body od)`

**Proof**

**Fix** `P Q :: 's expect` and `S :: 's trans × 's trans`

**Assume** `uP :: unitary P` and `uQ :: unitary Q`

and `ffst :: ∀ x ∈ S. feasible (fst x)`

and `usnd :: ∀ x ∈ S. ∀ Q. unitary Q → unitary (snd x Q)`

and `IH :: ∀ x ∈ S. snd x P && fst x Q ⊢ ⊢ fst x (P && Q)`

**Show** `Inf-utrans (snd ' S) P && Sup-trans (fst ' S) Q ⊢ Sup-trans (fst ' S) (P && Q)`

**Proof (cases)**

**Assume** `S = {}`

**Thus** `thesis`

by `(simp add: Inf-trans-def Sup-trans-def Inf-utrans-def Inf-exp-def Sup-exp-def exp-conj-def)`

**Next**

**Assume** `ne :: S ≠ {}`

**Let** `?f s = 1 + Sup-trans (fst ' S) (P && Q) s − Inf-utrans (snd ' S) P s`

from `ne obtain t where tin: t ∈ fst ' S by (auto)`

from `ne obtain u where uin: u ∈ snd ' S by (auto)`

from `tin ffst uP uQ have utPQ: unitary (t (P && Q))`

by `(auto intro: exp-conj-unitary)`

hence `∀ s. 0 ≤ t (P && Q) s by (auto)`

also {

from `ffst tin have le: le-utrans t (Sup-trans (fst ' S))`

by `(auto intro: Sup-trans-upper)`

with `uP uQ have ∃ s. t (P && Q) s ≤ Sup-trans (fst ' S) (P && Q) s`

by `(auto intro: exp-conj-unitary)`

}

finally have `nn-rhs: ∃ s. 0 ≤ Sup-trans (fst ' S) (P && Q) s`.

**Have** `∀ R. Inf-utrans (snd ' S) P && R ⊢ Sup-trans (fst ' S) (P && Q) → R ≤ ?f`

**Proof**

**Fix** `R`

**Assume** `¬ R ≤ ?f`

then obtain `s where ¬ R s ≤ ?f s by (auto)`

**Hence** `gt: ?f s < R s by (simp)`

from `nn-rhs` have `g1: 1 ≤ 1 + Sup-trans (fst ' S) (P && Q) s by (auto)`

**Hence** `Sup-trans (fst ' S) (P && Q) s = Inf-utrans (snd ' S) P s && ?f s`
by (simp add: pconj-def)
also from g1 have \ldots = \text{Inf-utrans (snd} \cdot S) \; P \; s + \neg f \; s - 1
  by (simp)
also from gt have \ldots < \text{Inf-utrans (snd} \cdot S) \; P \; s + R \; s - 1
  by (simp)
also {
  with g1 have \ldots \leq \text{Inf-utrans (snd} \cdot S) \; P \; s + R \; s
  by (simp)
  hence \text{Inf-utrans (snd} \cdot S) \; P \; s + R \; s - 1 = \text{Inf-utrans (snd} \cdot S) \; P \; s \& R \; s
  by (simp add: pconj-def)
}
finally have \neg \text{Inf-utrans (snd} \cdot S) \; P \; & & R \; s
  by (simp)
qed

moreover have \forall t \in \text{fst} \cdot S. \; \text{Inf-utrans (snd} \cdot S) \; P \; & & t \; Q \; \vdash \; \text{Sup-trans (fst} \cdot S) \; (P \; & & Q)
proof
fix t assume tin: t \in \text{fst} \cdot S
then obtain x where xin: x \in S and fx: t = \text{fst} \; x
  by (auto)

from xin have snd x \in \text{snd} \cdot S by (auto)
with uP usnd have \text{Inf-utrans (snd} \cdot S) \; P \; \vdash \; \text{snd} \; x \; P
  by (auto intro: le-utransD [OF Inf-utrans-lower])
  hence \text{Inf-utrans (snd} \cdot S) \; P \; & & \text{fst} \; x \; Q \; \vdash \; \text{snd} \; x \; P \; & & \text{fst} \; x \; Q
  by (auto intro: entails-frame)
also from xin IH have \ldots \vdash \; \text{fst} \; (P \; & & Q)
  by (auto)
also from xin \text{ffst exp-conj-unitary} [OF uP uQ]
  have \ldots \vdash \; \text{Sup-trans (fst} \cdot S) \; (P \; & & Q)
  by (auto intro: le-utransD [OF Sup-trans-upper])
finally show \text{Inf-utrans (snd} \cdot S) \; P \; & & t \; Q \; \vdash \; \text{Sup-trans (fst} \cdot S) \; (P \; & & Q)
  by (simp add: fx)
qed
ultimately have bt: \forall t \in \text{fst} \cdot S. \; t \; Q \; \vdash \; \neg f
  by (blast)

have \text{Sup-trans (fst} \cdot S) \; Q = \text{Sup-exp} \; \{t \; Q \; | t \; \in \text{fst} \cdot S\}
  by (simp add: Sup-trans-def)
also have \ldots \vdash \neg f
proof (rule Sup-exp-least)
from bt show \forall R \in \{t \; Q \; | t \; \in \text{fst} \cdot S\}. \; R \; \vdash \; \neg f
  by (blast)
from ne obtain t where tin: t \in \text{fst} \cdot S
  by (auto)
with \text{ffst} \; uQ have \text{unitary} \; (t \; Q)
  by (auto)
  hence \lambda s. 0 \; \vdash \; t \; Q
  by (auto)
also from tin bt have \ldots \vdash \neg f
  by (auto)
finally show \text{nneg (} \lambda s. 1 + \text{Sup-trans (fst} \cdot S) \; (P \; & & Q) \; s \; -
4.7. WELL-DEFINED PROGRAMS.

\[ \text{Inf-utrans (snd 'S) P s} \]

\[ \text{by(auto)} \]
\[ \text{qed} \]
\[ \text{finally have Inf-utrans (snd 'S) P \\& Sup-trans (fst 'S) Q \Rightarrow} \]
\[ \text{Inf-utrans (snd 'S) P \\& ?f} \]

\[ \text{by(auto intro:entails-frame)} \]
\[ \text{also from un-rhs have \ldots \Rightarrow Sup-trans (fst 'S) (P \\& Q)} \]

\[ \text{by(simp add:wp-eval exp-conj-def pconj-def)} \]

\[ \text{finally show ?thesis .} \]
\[ \text{qed} \]

next
\[ \text{fix P Q::'s expect and t ::'s trans} \]
\[ \text{assume uP: unitary P and uQ: unitary Q} \]
\[ \text{and ft: feasible t} \]
\[ \text{and uu: \( \lambda Q. \text{unitary Q} \Rightarrow \text{unitary (u Q)} \)} \]
\[ \text{and IH: u P \&\& t Q \Rightarrow (P \&\& Q)} \]
\[ \text{show wlp (body :: Embed u \_ G \_@ Skip) P \&\&} \]
\[ \text{wp (body :: Embed t \_ G \_@ Skip) Q \Rightarrow} \]
\[ \text{wp (body :: Embed t \_ G \_@ Skip) (P \&\& Q)} \]

\[ \text{proof(rule le-funI, simp add:wp-eval exp-conj-def pconj-def)} \]

\[ \text{fix s::'}s \]
\[ \text{have « G » s \* wp body (u P) s + (1 - « G » s) * P s +} \]
\[ \text{(« G » s \* wp body (t Q) s + (1 - « G » s) * Q s) \&\& 1 =} \]
\[ \text{(« G » s \* wp body (u P) s + « G » s \* wp body (t Q) s) +} \]
\[ \text{((1 - « G » s) * P s + (1 - « G » s) * Q s) \&\& (1 - « G » s))} \]

\[ \text{by(simp add:ac-simps)} \]
\[ \text{also have \ldots \leq} \]
\[ \text{(« G » s \* wp body (u P) s + « G » s \* wp body (t Q) s \&\& « G » s) +} \]
\[ \text{((1 - « G » s) * P s + (1 - « G » s) * Q s \&\& (1 - « G » s))} \]

\[ \text{by(rule tminus-add-mono)} \]
\[ \text{also have \ldots =} \]
\[ « G » s \* (wp body (u P) s + wp body (t Q) s \&\& I) +} \]
\[ (1 - « G » s) * (P s + Q s \&\& I) \]

\[ \text{by(simp add:tminus-left-distrib distrib-left)} \]
\[ \text{also \{} \]
\[ \text{from uP uQ ft uu} \]
\[ \text{have wp body (u P) \&\& wp body (t Q) \Rightarrow wp body (u P \&\& t Q)} \]

\[ \text{by(auto intro:sub-distrib-pconjD[OF sdp-body])} \]
\[ \text{also from IH unitary-sound[OF uP] unitary-sound[OF uQ] ft} \]
\[ \text{unitary-sound[OF uu[OF uP]]} \]

\[ \text{have \ldots \leq wp body (t (P \&\& Q))} \]

\[ \text{by(blast intro!:mono-contrareD[OF healthy-monoD, OF hwp] exp-conj-sound)} \]
\[ \text{finally have wp body (u P) s + wp body (t Q) s \&\& I \leq} \]
\[ \text{wp body (t (\lambda s. P s + Q s \&\& I)) s} \]

\[ \text{by(auto simp:exp-conj-def pconj-def)} \]
\[ \text{hence « G » s \* (wp body (u P) s + wp body (t Q) s \&\& I) +} \]
\[ (1 - « G » s) * (P s + Q s \&\& I) \leq} \]
\[ « G » s \* wp body (t (\lambda s. P s + Q s \&\& I)) s +} \]
\[(1 - « G » s) \ast (P s + Q s \odot 1)\]

by (auto intro: add-right-mono mult-left-mono)

\}

finally

show « G » s \ast wlp body \((u P) s + (1 - « G » s) \ast P s +
\((« G » s \ast \text{wp body} (t Q) s + (1 - « G » s) \ast Q s) \odot 1 \leq
\((1 - « G » s) \ast (P s + Q s \odot 1)\) .

qed

next

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

assume unitary \(P\), unitary \(Q\)

equiv-trans \(t t'\) equiv-utrans \(u u'\)

thus \(a' P \&\& t Q \vdash t (P \&\& Q)\)

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 healthy (wp prog) \&\& nearly-healthy (wlp prog)

\&\& wp-under-wlp prog \&\& sub-distrib-pconj prog

\&\& sublinear (wp prog) \&\& bd-cts (wp prog)

lemma well-defI[intro]:

\[
\begin{align*}
\text{健康 (wp prog); 近似健康 (wlp prog);}
\quad \text{wp-under-wlp prog; sub-distrib-pconj prog; sublinear (wp prog);}
\quad \text{bd-cts (wp prog)}
\end{align*}
\]

\Rightarrow

well-def prog

unfolding well-def-def by (simp)

lemma well-def-wp-healthy[dest]:

well-def prog \Rightarrow healthy (wp prog)

unfolding well-def-def by (simp)

lemma well-def-wlp-nearly-healthy[dest]:

well-def prog \Rightarrow nearly-healthy (wlp prog)

unfolding well-def-def by (simp)

lemma well-def-wp-under[dest]:
4.7. WELL-DEFINED PROGRAMS.

well-def prog \Rightarrow wp\text{-}under\text{-}wlp prog

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

**lemma** well-def-sdp[dest]:
well-def prog \Rightarrow sub-distrib-pconj prog

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

**lemma** well-def-wp-sublinear[dest]:
well-def prog \Rightarrow sublinear (wp prog)

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

**lemma** well-def-wp-cts[dest]:
well-def prog \Rightarrow bd-cts (wp prog)

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

**lemmas** wd-dests =
well-def-wp-healthy well-def-wlp-nearly-healthy
well-def-wp-under well-def-sdp
well-def-wp-sublinear well-def-wp-cts

**lemma** wd-Abort:
well-def Abort

by (blast intro: healthy-wp-Abort nearly-healthy-wlp-Abort
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] \Rightarrow well-def (a ;; b)

by (blast intro: healthy-wp-Seq nearly-healthy-wlp-Seq
wp-under-wlp-Seq sdp-Seq sublinear-wp-Seq
cts-wp-Seq)

**lemma** wd-PC:
[well-def a; well-def b; unitary P] \Rightarrow well-def (a \circ p \circ b)

by (blast intro: healthy-wp-PC nearly-healthy-wlp-PC
wp-under-wlp-PC sdp-PC sublinear-wp-PC
cts-wp-PC)
**Lemma** \( \text{wd-DC} \):

\[
\text{well-def } a; \text{well-def } b \implies \text{well-def } (a \bigcap b)
\]

**Proof**

\[
\text{by (blast intro:healthy-wp-DC nearly-healthy-wlp-DC wp-under-wlp-DC sdp-DC sublinear-wp-DC cts-wp-DC)}
\]

**Lemma** \( \text{wd-SetDC} \):

\[
[ \forall x. x \in S \implies \text{well-def } (a x); \forall s. S s \neq \{\}; \forall s. \text{finite } (S s) ] \implies \text{well-def } (\text{SetDC } a S)
\]

**Proof**

\[
\text{by (simp add: cts-wp-SetDC ex-in-conv healthy-intros wp-under-wlp-DC sdp-SetDC sublinear-wp-SetDC} \]

**Lemma** \( \text{wd-SetPC} \):

\[
[ \forall x. x \in (\text{supp } (p s)) \implies \text{well-def } (a x); \forall s. \text{unitary } (p s); \forall s. \text{finite } (\text{supp } (p s)); \forall s. \text{sum } (p s) (\text{supp } (p s)) \leq 1 ] \implies \text{well-def } (\text{SetPC } a p)
\]

**Proof**

\[
\text{by (iprover intro!:well-defI healthy-wp-SetPC nearly-healthy-wlp-SetPC wp-under-wlp-SetPC sdp-SetPC sublinear-wp-SetPC cts-wp-SetPC dest:wd-dests unitary-sound sound-nneg [OF unitary-sound] nnegD)}
\]

**Lemma** \( \text{wd-Embed} \):

**Proof**

\[
\text{fixes } t :: \tau \text{ trans}
\]

**Assumes**

\( \text{ht: healthy } t \text{ and st: sublinear } t \text{ and ct: bd-cts } t \)

**Shows**

\( \text{well-def } (\text{Embed } t) \)

**Proof**

\[
\text{from } \text{ht show healthy } (\text{wp } (\text{Embed } t)) \text{ nearly-healthy } (\text{wlp } (\text{Embed } t))\]

**By**

\[
\text{(simp add: wp-def wlp-def Embed-def healthy-nearly-healthy)}
\]

**From**

\( \text{st show sublinear } (\text{wp } (\text{Embed } t)) \text{ by (simp add: wp-under-wlp-def wp-eval)} \)

**Show**

\( \text{sub-distrib-pconj } (\text{Embed } t) \)

**By**

\[
\text{(rule sub-distrib-pconjI, auto intro: le-funI [OF sublinearD [OF st, where a=1 and b=1 and c=1, simplified]] simp: exp-conj-def pconj-def wp-def wlp-def Embed-def])}
\]

**From**

\( \text{ct show bd-cts } (\text{wp } (\text{Embed } t)) \)

**By**

\[
\text{(simp add: wp-def Embed-def)}
\]

**Qed**

**Lemma** \( \text{wd-repeat} \):

\( \text{well-def } a \implies \text{well-def } (\text{repeat } n a) \)

**Proof**

\[
\text{by (blast intro: healthy-wp-repeat nearly-healthy-wlp-repeat wp-under-wlp-repeat sdp-repeat sublinear-wp-repeat cts-wp-repeat)}
\]

**Lemma** \( \text{wd-Bind} \):

\[
[ \forall s. \text{well-def } (a (f s)) ] \implies \text{well-def } (\text{Bind } f a)\]

**Proof**

\[
\text{by (blast intro: healthy-wp-Bind nearly-healthy-wlp-Bind wp-under-wlp-Bind sdp-Bind sublinear-wp-Bind cts-wp-Bind)}
\]
4.8. THE LOOP RULES

**Lemma wd-loop:**
well-def body \(\Rightarrow\) well-def (do G \(\rightarrow\) body od)


**Lemmas wd-intros =**
wd-Abort wd-Skip wd-Apply
wd-Embed wd-Seq wd-PC
wd-DC wd-SetPC wd-SetDC
wd-Bind wd-repeat wd-loop

**End**

### 4.8 The Loop Rules

**Theory Loops imports WellDefined begin**

Given a well-defined body, we can annotate a loop using an invariant, just as in the classical setting.

#### 4.8.1 Liberal and Strict Invariants.

A probabilistic invariant generalises a boolean one: it *entails* itself, given the loop guard.

**Definition**
wp-inv :: (′s ⇒ bool) ⇒ ′s prog ⇒ (′s ⇒ real) ⇒ bool

*where*
wp-inv G body I \(\Leftarrow\) (\(\forall\) s. «G» s \(\ast\) I s \(\leq\) wp body I s)

**Lemma** wp-invI:
\(\bigwedge\! I. (\bigwedge\! s. «G» s \(\ast\) I s \(\leq\) wp body I s) \Rightarrow 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 \(\Leftarrow\) (\(\forall\) s. «G» s \(\ast\) I s \(\leq\) wlp body I s)

**Lemma** wlp-invI:
\(\bigwedge\! I. (\bigwedge\! s. «G» s \(\ast\) I s \(\leq\) wlp body I s) \Rightarrow wlp-inv G body I\)

*by* (simp add:wlp-inv-def)

**Lemma** wlp-invD:

wlp-inv G body I \(\Rightarrow\) «G» s \(\ast\) I s \(\leq\) wlp body I s

*by* (simp add:wlp-inv-def)

For standard invariants, the multiplication reduces to conjunction.
**4.8.2 Partial Correctness**


**Lemma wp-inv-stdD:**
- ** Assumes** \( \text{inv: wp-inv } G \text{ body } «I» \)
  - **and** \( \text{hb: healthy (wp body)} \)
- ** Shows** \( G \land \langle I \rangle \vdash \text{wp body } «I» \)

**Proof:** (rule le-funI)
- Fix \( s \)
- Show \( (G \land \langle I \rangle) \leq \text{wp body } «I» \)
- Proof (cases \( G \))
  - Case \( \text{False} \)
    - With \( \text{hb} \) show \( ?\text{thesis} \)
    - By (auto simp: exp-conj-def)
  - Case \( \text{True} \)
    - Hence \( (G \land \langle I \rangle) s = (G s) * I s \)
    - By (simp add: exp-conj-def)
    - Also from \( \text{inv} \) have \( (G s) * I s \leq \text{wp body } «I» \)
      - By (simp add: wp-inv-def)
    - Finally show \( ?\text{thesis} \)
- QED

**Lemma wlp-Loop:**
- ** Assumes** \( \text{wd: well-def body} \)
  - **and** \( \text{ui: unitary I} \)
  - **and** \( \text{inv: wlp-inv } G \text{ body I} \)
- ** Shows** \( I \leq \text{wlp do } G \rightarrow \text{body od } (\lambda s. \langle \mathcal{N} G \rangle s * I s) \)
  - (is \( I \leq \text{wp do } G \rightarrow \text{body od } ?P \))
- ** Proof:**
  - Let \( Q s = \langle G \rangle s * \text{wlp body } Q s + \langle \mathcal{N} G \rangle s * ?P s \)
  - Have \( I \vdash \text{gfp-exp } !f \)
  - Proof (rule gfp-exp-upperbound[OF - ui])
    - Have \( I = (\lambda s. \langle G \rangle s + \langle \mathcal{N} G \rangle s) * I s \)
      - By (simp add: negate-embed)
    - Also have \( ... = (\lambda s. \langle G \rangle s * I s + \langle \mathcal{N} G \rangle s * I s) \)
      - By (simp add: algebra-simps)
    - Also have \( ... = (\lambda s. \langle G \rangle s * (\langle G \rangle s * I s) + \langle \mathcal{N} G \rangle s * (\langle \mathcal{N} G \rangle s * I s)) \)
      - By (simp add: embed-bool-idem algebra-simps)
    - Also have \( ... \vdash (\lambda s. \langle G \rangle s * \text{wp body } I s + \langle \mathcal{N} G \rangle s * (\langle \mathcal{N} G \rangle s * I s)) \)
      - Using \( \text{inv} \)
      - Finally show \( I \vdash (\lambda s. \langle G \rangle s * \text{wp body } I s + \langle \mathcal{N} G \rangle s * (\langle \mathcal{N} G \rangle s * I s)) \)
  - QED
- Also from \( \text{ui well-def-wlp-nearly-healthy}[OF wd] \) have \( ... = \text{wp do } G \rightarrow \text{body od } ?P \)
  - By (auto intro! : wlp-Loop1 [symmetric] unitary-intros)
  - Finally show \( ?\text{thesis} \)
- QED
4.8. THE LOOP RULES

4.8.3 Total Correctness

The first total correctness lemma for loops which terminate with probability
1 [McIver and Morgan, 2004, Lemma 7.3.1, §7, p. 186].

Lemma wp-Loop:
  assumes wd: well-def body
  and inv: wlp-inv G body I
  and unit: unitary I
  shows I \&\& wp (do G \rightarrow body od) (\lambda s. I) \vdash wp (do G \rightarrow body od) (\lambda s. «N
  G» s \ast I s)
  (is I \&\& ?T \vdash wp ?loop ?X)

Proof –

We first appeal to the liberal loop rule:

from assms have I \&\& ?T \vdash wp ?loop ?X \&\& ?T
  by (blast intro: exp-conj-mono-left wlp-Loop)

Next, by sub-conjunctivity:

also {
  from wd have sdp-loop: sub-distrib-pconj (do G \rightarrow body od)
  by (blast intro: sdp-intros)

  from wd unit have wp ?loop ?X \&\& ?T \vdash wp ?loop (?X \&\& (\lambda s. I))
  by (blast intro: sub-distrib-pconjD sdp-intros unitary-intros)
}

Finally, the conjunction collapses:

finally show ?thesis
  by (simp add: exp-conj-1-right sound-intros sound-nneg unit unitary-sound)

qed

4.8.4 Unfolding

Lemma wp-loop-unfold:
  fixes body :: 's prog
  assumes sP: sound P
  and h: healthy (wp body)
  shows wp (do G \rightarrow body od) P =
  (\lambda s. «N G» s \ast P s + «G» s \ast wp body (wp (do G \rightarrow body od) P) s)

Proof (simp only: wp-eval)

let ?X t = wp (body :: Embed t « G » \oplus Skip)

have eqvia-trans (lfp-trans ?X)
  (wp (body :: Embed (lfp-trans ?X) « G » \oplus Skip))

proof (intro lfp-trans-unfold)
  fix t::'s trans and P::'s expect
  assume st: \bigwedge Q. sound Q \rightarrow sound (t Q)
  and sP: sound P
  with h show sound (?X t P)
  by (rule wp-loop-step-sound)
next
fix t u::′s trans
assume le-trans t u (∀ P. sound P ⇒ sound (t P))
(∀ P. sound P ⇒ sound (u P))
with h show le-trans (wp (body ;; Embed t △ G ⊕ Skip))
(wp (body ;; Embed u △ G ⊕ Skip))
by (iprover intro:wp-loop-step-mono)
next
let ?v = λ P s. bound-of P from h show le-trans (wp (body ;; Embed ?v △ G ⊕ Skip)) ?v
by (intro le-transI, simp add: wp-eval lfp-loop-fp [unfolded negate-embed])
fix P::′s expect
assume sound P thus sound (?v P) by (auto)
qed
also have equiv-trans ...
(∀ P. «N G» s * P s + «G» s * wp body (wp (Embed (lfp-trans ?X) P) s))
by (rule equiv-transI, simp add: wp-eval algebra-simps negate-embed)
finally show lfp-trans ?X P =
(∀ s. «N G» s * P s + «G» s * wp body (lfp-trans ?X P) s)
using sP unfolding wp-eval by (blast)
qed

lemma wp-loop-nguard:
[ healthy (wp body); sound P; ¬ G s ] ⇒ wp do G → body od P s = P s
by (subst wp-loop-unfold, simp-all)

lemma wp-loop-guard:
[ healthy (wp body); sound P; G s ] ⇒ wp do G → body od P s = wp (body ;; do G → body od) P s
by (subst wp-loop-unfold, simp-all add: wp-eval)
end

4.9 The Algebra of pGCL

theory Algebra imports WellDefined begin

Programs in pGCL have a rich algebraic structure, largely mirroring that for GCL. We show that programs form a lattice under refinement, with \( a \sqcap b \) 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.
4.9. THE ALGEBRA OF PGCL

**definition**
refines :: 's prog ⇒ 's prog ⇒ bool (infix ⊑)

**where**
prog ⊑ prog' ≡ ∀ P. sound P → wp prog P ⊢ wp prog' P

**lemma** refinesI[intro]:
[ ∀P. sound P → wp prog P ⊢ wp prog' P ] ⇒ prog ⊑ prog'

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

**lemma** refinesD[dest]:
[ prog ⊑ prog'; sound P ] ⇒ wp prog P ⊢ wp prog' P

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

The equivalence relation below will turn out to be that induced by refinement. It is also the application of equiv-trans to the weakest precondition.

**definition**
pequiv :: 's prog ⇒ 's prog ⇒ bool (infix≃)

**where**
prog ≃ prog' ≡ ∀ P. sound P → wp prog P = wp prog' P

**lemma** pequivI[intro]:
[ ∀P. sound P → wp prog P = wp prog' P ] ⇒ prog ≃ prog'

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

**lemma** pequivD[dest,simp]:
[ prog ≃ prog'; sound P ] ⇒ wp prog P = wp prog' P

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

**lemma** pequiv-equiv-trans:
a ≃ b ↔ equiv-trans (wp a) (wp b)

**by**(auto)

4.9.2 Simple Identities

The following identities involve only the primitive operations as defined in Section 4.1.1, and refinement as defined above.

**Laws following from the basic arithmetic of the operators separately**

**lemma** DC-comm[ac-simps]:
a ∩ b = b ∩ a

**unfolding** DC-def by(simp add:ac-simps)

**lemma** DC-assoc[ac-simps]:
a ∩ (b ∩ c) = (a ∩ b) ∩ c

**unfolding** DC-def by(simp add:ac-simps)
lemma \( DC\text{-idem} \):
\[
\begin{align*}
    a \sqcap a &= a \\
    \text{unfolding } DC\text{-def by (simp)}
\end{align*}
\]

lemma \( AC\text{-comm}[ac\text{-simps}] \):
\[
\begin{align*}
    a \sqcup b &= b \sqcup a \\
    \text{unfolding } AC\text{-def by (simp add: ac\text{-simps})}
\end{align*}
\]

lemma \( AC\text{-assoc}[ac\text{-simps}] \):
\[
\begin{align*}
    a \sqcup (b \sqcup c) &= (a \sqcup b) \sqcup c \\
    \text{unfolding } AC\text{-def by (simp add: ac\text{-simps})}
\end{align*}
\]

lemma \( AC\text{-idem} \):
\[
\begin{align*}
    a \sqcup a &= a \\
    \text{unfolding } AC\text{-def by (simp)}
\end{align*}
\]

lemma \( PC\text{-quasi-comm} \):
\[
\begin{align*}
    a p \oplus b &= b (\lambda s. 1 - p s) \oplus a \\
    \text{unfolding } PC\text{-def by (simp add: algebra\text{-simps})}
\end{align*}
\]

lemma \( PC\text{-idem} \):
\[
\begin{align*}
    a p \oplus a &= a \\
    \text{unfolding } PC\text{-def by (simp add: algebra\text{-simps})}
\end{align*}
\]

lemma \( Seq\text{-assoc}[ac\text{-simps}] \):
\[
\begin{align*}
    A ;; (B ;; C) &= A ;; B ;; C \\
    \text{by (simp add: Seq\text{-def o\text{-def})}
\end{align*}
\]

lemma \( Abort\text{-refines}[intro] \):
\[
\begin{align*}
    \text{well-def } a &= \Rightarrow Abort \sqsubseteq a \\
    \text{by (rule refinesI, unfold wp\text{-eval}, auto dest!: well-def-wp\text{-healthy})}
\end{align*}
\]

Laws relating demonic choice and refinement

lemma \( left\text{-refines-DC} \):
\[
\begin{align*}
    (a \sqcap b) \sqsubseteq a \\
    \text{by (auto intro!: refinesI simp: wp\text{-eval})}
\end{align*}
\]

lemma \( right\text{-refines-DC} \):
\[
\begin{align*}
    (a \sqcap b) \sqsubseteq b \\
    \text{by (auto intro!: refinesI simp: wp\text{-eval})}
\end{align*}
\]

lemma \( DC\text{-refines} \):
\[
\begin{align*}
    \text{fixes } a::\text{'s prog and } b \text{ and } c \\
    \text{assumes } rab: a \sqsubseteq b \text{ and rac: } a \sqsubseteq c \\
    \text{shows } a \sqsubseteq (b \sqcap c) \\
    \text{proof} \\
    \text{fix } P::\text{'s } \Rightarrow \text{ real assume } sP: \text{ sound } P \\
    \text{with } \text{assms have } \text{wp } a \ P \vdash \text{wp } b \ P \text{ and wp } a \ P \vdash \text{wp } c \ P
\end{align*}
\]
by (auto dest: refinesD)
thus \( \text{wp } a \ P \vdash \text{wp } (b \sqcup c) \ P \)
  by (auto simp: wp-eval intro: min.boundedI)
qed

lemma DC-mono:
  fixes \( a' \) 's prog
  assumes rab: \( a \subseteq b \) and rcd: \( c \subseteq d \)
  shows \( (a \sqcap c) \subseteq (b \sqcap d) \)
proof (rule refinesI, unfold wp-eval, rule le-funI)
  fix \( P' \) 's \( \Rightarrow \) real and \( s' \) 's
  assume \( sP \): sound \( P \)
  with assms have \( \text{wp } a \ P \ s \leq \text{wp } b \ P \ s \) and \( \text{wp } c \ P \ s \leq \text{wp } d \ P \ s \)
    by (auto)
  thus \( \min (\text{wp } a \ P \ s) (\text{wp } c \ P \ s) \leq \min (\text{wp } b \ P \ s) (\text{wp } d \ P \ s) \)
    by (auto)
qed

Laws relating angelic choice and refinement

lemma left-refines-AC:
  \( a \subseteq (a \sqcup b) \)
  by (auto intro!: refinesI simp: wp-eval)

lemma right-refines-AC:
  \( b \subseteq (a \sqcup b) \)
  by (auto intro!: refinesI simp: wp-eval)

lemma AC-refines:
  fixes \( a' \) 's prog and \( b \) and \( c \)
  assumes rac: \( a \subseteq c \) and rbc: \( b \subseteq c \)
  shows \( (a \sqcup b) \subseteq c \)
proof
  fix \( P' \) 's \( \Rightarrow \) real
  assume \( sP \): sound \( P \)
  with assms have \( \bigwedge s. \text{wp } a \ P \ s \leq \text{wp } c \ P \ s \)
    and \( \bigwedge s. \text{wp } b \ P \ s \leq \text{wp } c \ P \ s \)
    by (auto dest: refinesD)
  thus \( \text{wp } (a \sqcup b) \ P \vdash \text{wp } c \ P \)
    unfolding wp-eval by (auto)
qed

lemma AC-mono:
  fixes \( a' \) 's prog
  assumes rab: \( a \subseteq b \) and rcd: \( c \subseteq d \)
  shows \( (a \sqcup c) \subseteq (b \sqcup d) \)
proof (rule refinesI, unfold wp-eval, rule le-funI)
  fix \( P' \) 's \( \Rightarrow \) real and \( s' \) 's
  assume \( sP \): sound \( P \)
  with assms have \( \text{wp } a \ P \ s \leq \text{wp } b \ P \ s \) and \( \text{wp } c \ P \ s \leq \text{wp } d \ P \ s \)
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 \( a \uplus b \) together

**Lemma** DC-refines-PC:

- **Assumes** unit: unitary \( p \)
- **Shows** \( (a \uplus b) \subseteq (a \oplus b) \)

**Proof**

- **Rule** refinesI, unfold \( wp\text{-}eval \), rule \( le\text{-}fanI \)
- **Fix** \( s \) and \( P :'a \Rightarrow real \)
- **Assume** sound: sound \( P \)
- From unit have \( mn\text{-}p: 0 \leq p s \) by(blast)
- From unit have \( p s \leq 1 \) by(blast)
- Hence \( mn\text{-}np: 0 \leq 1 - p s \) by(simp)
- **Show** \( \min (wp \ a \ P \ s) (wp \ b \ P \ s) \leq p s \ast wp \ a \ P \ s + (1 - p s) \ast wp \ b \ P \ s \)

**Proof**

- **Cases** \( wp \ a \ P \ s \leq wp \ b \ P \ s \),
  - simp-all add:min.absorb1 min.absorb2
  - Case True
    - Note \( le = this \)
    - Have \( wp \ a \ P \ s = (p s + (1 - p s)) \ast wp \ a \ P \ s \) by(simp)
    - Also have \( \ldots = p s \ast wp \ a \ P \ s + (1 - p s) \ast wp \ a \ P \ s \)
    - Finally show \( wp \ a \ P \ s \leq p s \ast wp \ a \ P \ s + (1 - p s) \ast wp \ b \ P \ s \).
  - Next
    - Case False
      - Then have \( lc: wp \ b \ P \ s \leq wp \ a \ P \ s \) by(simp)
      - Have \( wp \ b \ P \ s = (p s + (1 - p s)) \ast wp \ b \ P \ s \) by(simp)
      - Also have \( \ldots = p s \ast wp \ b \ P \ s + (1 - p s) \ast wp \ b \ P \ s \)
      - Finally show \( wp \ b \ P \ s \leq p s \ast wp \ a \ P \ s + (1 - p s) \ast wp \ b \ P \ s \).
  - qed

**Lemma** PC-refines-AC:

- **Assumes** unit: unitary \( p \)
- **Shows** \( (a \uplus b) \subseteq (a \oplus b) \)

**Proof**

- **Rule** refinesI, unfold \( wp\text{-}eval \), rule \( le\text{-}fanI \)
- **Fix** \( s \) and \( P :'a \Rightarrow real \)
- **Assume** sound: sound \( P \)
- From unit have \( mn\text{-}p: 0 \leq p s \) by(blast)
- From unit have \( p s \leq 1 \) by(blast)
- Hence \( mn\text{-}np: 0 \leq 1 - p s \) by(simp)
- **Show** \( \min (wp \ a \ P \ s) (wp \ b \ P \ s) \leq p s \ast wp \ a \ P \ s + (1 - p s) \ast wp \ b \ P \ s \)

**Proof**

- **Cases** \( wp \ a \ P \ s \leq wp \ b \ P \ s \),
  - simp-all add:min.absorb1 min.absorb2
  - Case True
    - Note \( le = this \)
    - Have \( wp \ a \ P \ s = (p s + (1 - p s)) \ast wp \ a \ P \ s \) by(simp)
    - Also have \( \ldots = p s \ast wp \ a \ P \ s + (1 - p s) \ast wp \ a \ P \ s \)
    - Finally show \( wp \ a \ P \ s \leq p s \ast wp \ a \ P \ s + (1 - p s) \ast wp \ b \ P \ s \).
  - Next
    - Case False
      - Then have \( lc: wp \ b \ P \ s \leq wp \ a \ P \ s \) by(simp)
      - Have \( wp \ b \ P \ s = (p s + (1 - p s)) \ast wp \ b \ P \ s \) by(simp)
      - Also have \( \ldots = p s \ast wp \ b \ P \ s + (1 - p s) \ast wp \ b \ P \ s \)
      - Finally show \( wp \ b \ P \ s \leq p s \ast wp \ a \ P \ s + (1 - p s) \ast wp \ b \ P \ s \).
  - qed
  - qed
assumes unit: unitary p
shows \((a \uplus b) \sqsubseteq (a \sqcup b)\)
proof (rule refinesI, unfold wp-eval, rule le-funI)
fix s and P::'a ⇒ real assume sound: sound P

from unit have \(p s \leq 1\) by (blast)
hence nn-np: \(0 \leq 1 - p s\) by (simp)

show \(p s \ast wp a P s + (1 - p s) \ast wp b P s \leq\)
\(\max (wp a P s) (wp b P s)\)
proof (cases wp a P s \(\leq\) wp b P s)
case True note leab = this
with unit nn-np
have \(p s \ast wp a P s + (1 - p s) \ast wp b P s \leq\)
\(p s \ast wp b P s + (1 - p s) \ast wp b P s\)
by (auto intro: add-mono mult-left-mono)
also have \(\ldots = wp b P s\)
by (auto simp: field-simps)
also from leab
have \(\ldots = \max (wp a P s) (wp b P s)\)
by (auto)
finally show \(?thesis\).

next
case False note leba = this
with unit nn-np
have \(p s \ast wp a P s + (1 - p s) \ast wp b P s \leq\)
\(p s \ast wp a P s + (1 - p s) \ast wp a P s\)
by (auto intro: add-mono mult-left-mono)
also have \(\ldots = wp a P s\)
by (auto simp: field-simps)
also from leba
have \(\ldots = \max (wp a P s) (wp b P s)\)
by (auto)
finally show \(?thesis\).
qed

Laws depending on the arithmetic of \(a \sqcup b\) and \(a \sqcap b\) together

lemma DC-refines-AC:
\((a \sqcap b) \sqsubseteq (a \sqcup b)\)
by (auto intro!: refinesI simp: wp-eval)

Laws Involving Refinement and Equivalence

lemma pr-trans[trans]:
fixes A::'a prog
assumes prAB: \(A \sqsubseteq B\)
and prBC: \(B \sqsubseteq C\)
shows \(A \sqsubseteq C\)
proof
fix \( P : 'a \Rightarrow real \) assume \( sP : sound P \)
with \( prAB \) have \( wp A P \vdash wp B P \) by (blast)
also from \( sP \) and \( prBC \) have \( \ldots \vdash wp C P \) by (blast)
finally show \( wp A P \vdash \ldots \).
qed

lemma pequiv-refl[intro, simp]:
\( a \simeq a \)
by (auto)

lemma pequiv-comm[ac-simps]:
\( a \simeq b \leftrightarrow b \simeq a \)
unfolding pequiv-def
by (rule iffI, safe, simp-all)

lemma pequiv-pr[dest]:
\( a \simeq b \Longrightarrow a \subseteq b \)
by (auto)

lemma pequiv-trans[intro, trans]:
\[ [ a \simeq b; b \simeq c ] \Longrightarrow a \simeq c \]
unfolding pequiv-def by (auto intro!: order-trans)

lemma pequiv-pr-trans[intro, trans]:
\[ [ a \subseteq b; b \simeq c ] \Longrightarrow a \subseteq c \]
unfolding pequiv-def refines-def by (simp)

lemma pr-pequiv-trans[intro, trans]:
\[ [ a \subseteq b; b \simeq c ] \Longrightarrow a \subseteq c \]
unfolding pequiv-def refines-def by (simp)

Refinement induces equivalence by antisymmetry:

lemma pequiv-antisym:
\[ [ a \subseteq b; b \subseteq a ] \Longrightarrow a \simeq b \]
by (auto intro!: antisym)

lemma pequiv-DC:
\[ [ a \simeq c; b \simeq d ] \Longrightarrow (a \cap b) \simeq (c \cap d) \]
by (auto intro!: DC-mono pequiv-antisym simp: ac-simps)

lemma pequiv-AC:
\[ [ a \simeq c; b \simeq d ] \Longrightarrow (a \cup b) \simeq (c \cup d) \]
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
4.9. THE ALGEBRA OF PGCL

order) among sub-additive programs.

**lemma** refines-determ:

- **fixes** \( a :: 's prog \)
- **assumes** \( da :: determ (wp a) \)
  - **and** \( wa :: well-def a \)
  - **and** \( wb :: well-def b \)
  - **and** \( dr :: a \subseteq b \)
- **shows** \( a \approx b \)

**Proof by contradiction.**

**proof** (rule pequivI, rule contrapos-pp)

- **from** \( wb \) **have** feasible \((wp b)\) **by** (auto)
- **with** \( wb \) **have** sub; sub-add \((wp b)\)
  - **by** (auto dest: sublinear-subadd[OF well-def-wp-sublinear])
- **fix** \( P :: 's \Rightarrow real \)
  - **assume** \( sP \): sound \( P \)

Assume that \( a \) and \( b \) are not equivalent:

- **assume** \( ne :: wp a P \neq wp b P \)

Find a point at which they differ. As \( a \subseteq b \), \( wp b P s \) must by strictly greater than \( wp a P s \) here:

- **hence** \( \exists s. \ wp a P s < wp b P s \)
- **proof** (rule contrapos-np)
  - **assume** \( \neg \exists s. \ wp a P s < wp b P s \)
  - **hence** \( \forall s. \ wp b P s \leq wp a P s \) **by** (auto simp:not-less)
  - **hence** \( wp b P \vdash wp a P \) **by** (auto)
  - moreover from \( sP \) **have** \( wp a P \vdash wp b P \) **by** (auto)
  - **ultimately show** \( wp a P = wp b P \) **by** (auto)
  - **then obtain** \( s \) **where** \( less :: wp a P s < wp b P s \) **by** (blast)

Take a carefully constructed expectation:

- **let** \( ?Pc = \lambda s. bound-of P - P s \)
- **have** \( sPc :: sound ?Pc \)
- **proof** (rule soundI)
  - **from** \( sP \) **have** \( \forall s. 0 \leq P s \) **by** (auto)
  - **hence** \( \forall s. ?Pc s \leq bound-of P \) **by** (auto)
  - **thus bounded** \( ?Pc \) **by** (blast)
  - **from** \( sP \) **have** \( \forall s. P s \leq bound-of P \) **by** (auto)
  - **hence** \( \forall s. 0 \leq ?Pc s \)
    - **by** auto
  - **thus** \( nneg \ ?Pc \) **by** (auto)

**qed**

We then show that \( wp b \) violates feasibility, and thus healthiness.

- **from** \( sP \) **have** \( 0 \leq bound-of P \) **by** (auto)
  - **with** \( da \) **have** \( bound-of P = wp a (\lambda s. bound-of P) s \)
    - **by** (simp add: maximalD determ-maximalD)
also have ... = wp a (λs. ?Pc s + P s) s
  by(simp)
also from da sP sPc have ... = wp a ?Pc s + wp a P s
  by(subst additiveD[OF determ-additiveD], simp-all add:sP sPc)
also from sPc dr have ... ≤ wp b ?Pc s
  by(auto)
also from less have ... < wp b ?Pc s + wp b P s
  by(auto)
also from sab sP sPc have ... ≤ wp b (λs. ?Pc s + P s) s
  by(blast)
finally have ¬ wp b (λs. bound-of P) s ≤ bound-of P
  by(simp)
thus ¬ bounded-by (bound-of P) (wp b (λs. bound-of P))
  by(auto)

next

However,

fix P::'s ⇒ real assume sP: sound P
hence nneg (λs. bound-of P) by(auto)
moreover have bounded-by (bound-of P) (λs. bound-of P) by(auto)
ultimately
show bounded-by (bound-of P) (wp b (λs. bound-of P))
  using wb by(auto dest!:well-def-wp-healthy)
qed

4.9.4 The Algebraic Structure of Refinement

Well-defined programs form a half-bounded semilattice under refinement, where Abort is bottom, and a ∩ b is inf. There is no unique top element, but all fully-deterministic programs are maximal.

The type that we construct here is not especially useful, but serves as a convenient way to express this result.

quotient-type 's program =
  's prog / partial : λa b. a ≃ b ∧ well-def a ∧ well-def b

proof (rule part-equivpI)
  have Skip ≃ Skip and well-def Skip by(auto intro:wd-intros)
  thus ∃x. x ≃ x ∧ well-def x ∧ well-def x by(blast)
  show symp (λa b. a ≃ b ∧ well-def a ∧ well-def b)
    proof (rule sympI, safe)
        fix a::'a prog and b
        assume a ≃ b
        hence equiv-trans (wp a) (wp b)
          by(simp add:pequiv-equiv-trans)
        thus b ≃ a by(simp add:ac-simps pequiv-equiv-trans)
      qed
    show transp (λa b. a ≃ b ∧ well-def a ∧ well-def b)
      by(rule transpI, safe, rule pequiv-trans)
  qed

qed
4.9. THE ALGEBRA OF PGCL

instantiation program :: (type) semilattice-inf begin
lift-definition
  less-eq-program :: 'a program ⇒ 'a program ⇒ bool is refines
proof(safe)
  fix a::'a prog and b c d
  assume a ≃ b hence b ≃ a by(simp add:ac-simps)
  also assume a ⊑ c
  also assume c ≃ d
  finally show b ⊑ d.
next
  fix a::'a prog and b c d
  assume a ≃ b
  also assume b ⊑ d
  also assume c ≃ d hence d ≃ c by(simp add:ac-simps)
  finally show a ⊑ c.
qed

lift-definition
  less-program :: 'a program ⇒ 'a program ⇒ bool
  is λa b. a ⊑ b ∧ ¬b ⊑ a
proof(safe)
  fix a::'a prog and b c d
  assume a ≃ b hence b ≃ a by(simp add:ac-simps)
  also assume a ⊑ c
  also assume c ≃ d
  finally show b ⊑ d.
next
  fix a::'a prog and b c d
  assume a ≃ b
  also assume b ⊑ d
  also assume c ≃ d hence d ≃ c by(simp add:ac-simps)
  finally show a ⊑ c.
next
  fix a b and c::'a prog and d
  assume c ≃ d
  also assume d ⊑ b
  also assume a ≃ b hence b ≃ a by(simp add:ac-simps)
  finally have c ⊑ a.
  moreover assume ¬c ⊑ a
  ultimately show False by(auto)
next
  fix a b and c::'a prog and d
  assume c ≃ d hence d ≃ c by(simp add:ac-simps)
  also assume c ⊑ a
  also assume a ≃ b
  finally have d ⊑ b.
  moreover assume ¬d ⊑ b
  ultimately show False by(auto)
pled

lift-definition
inf-program :: 'a program ⇒ 'a program ⇒ 'a program is DC
proof (safe)
fix a b c d::'s prog
assume a ≼ b and c ≼ d
thus (a ∩ c) ≼ (b ∩ 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, iprover intro:pequiv-antisym)
next
fix x y z::'a program
assume x ≤ y and y ≤ z
thus x ≤ z
  by (transfer, iprover intro:pr-trans)
next
fix x y z::'a program
assume x ≤ y and x ≤ z
thus x ≤ inf y z
  by (transfer, iprover intro:DC-refines)
qed
end

instantiation program :: (type) bot begin
lift-definition
bot-program :: 'a program is Abort
by (auto intro:wd-intros)
4.9. THE ALGEBRA OF PGCL

instance ..
end

lemma eq-det: \( a \simeq b \,\vdash\, \text{determ} (wp a) \implies \text{determ} (wp b) \)
proof(intro determI additiveI maximalI)
  fix a b::'s prog and P::'s \Rightarrow real
  and Q::'s \Rightarrow real and s::'
  assume da: determ (wp a)
  assume sP: sound P and sQ: sound Q
  and eq: a \simeq b
  hence wp b (\lambda s. P s + Q s) s =
  wp a (\lambda s. P s + Q s) s
  by(simp add:sound-intros)
  also from da sP sQ
  have ... = wp a P s + wp a Q s
  by(simp add:additiveD determ-additiveD)
  also from eq sP sQ
  have ... = wp b P s + wp b Q s
  by(simp add:pequivD)
  finally show wp b (\lambda s. P s + Q s) s = wp b P s + wp b Q s .
next
  fix a b::'s prog and c::real
  assume da: determ (wp a)
  assume a \simeq b hence b \simeq a by(simp add:ac-simps)
  moreover assume nn: 0 \leq c
  ultimately have wp b (\lambda-. c) = wp a (\lambda-. c)
  by(simp add:pequivD const-sound)
  also from da nn have ... = (\lambda-. c)
  by(simp add:determ-maximalD maximalD)
  finally show wp b (\lambda-. c) = (\lambda-. c) .
qed

lift-definition
  pdeterm :: 's program \Rightarrow bool
is \lambda a. determ (wp a)
proof(safe)
  fix a b::'s prog
  assume a \simeq b and determ (wp a)
  thus determ (wp b) by(rule eq-det)
next
  fix a b::'s prog
  assume a \simeq b hence b \simeq a by(simp add:ac-simps)
  moreover assume determ (wp b)
  ultimately show determ (wp a) by(rule eq-det)
qed

lemma determ-maximal:
  \[ pdeterm a; a \leq x \implies a = x \]
  by(transfer, auto intro:refines-determ)
4.9.5 Data Refinement

A projective data refinement construction for pGCL. By projective, we mean that the abstract state is always a function ($\varphi$) of the concrete state. Refinement may be predicated ($G$) on the state.

**definition**

drefines :: ('b ⇒ 'a) ⇒ 'b prog ⇒ bool

**where**

drefines $\varphi$ $G$ $A$ $B$ ≡ $\forall$ $P$ $Q$. ($\text{unitary } P \land \text{unitary } Q \land (P \vdash \text{wp } A\ Q)$) $\rightarrow$ ($«G» \&\& (P \circ \varphi) \vdash \text{wp } B\ (Q \circ \varphi)$)

**lemma** drefinesD[$\text{dest}$]:

[ drefines $\varphi$ $G$ $A$ $B$; unitary $P$; unitary $Q$; $P \vdash \text{wp } A\ Q$ ] $\Rightarrow$

«$G» &\& (P o $\varphi$) $\vdash$ wp $B$ (Q o $\varphi$)

**unfolding** drefines-def by(blast)

We can alternatively use $G$ as an assumption:

**lemma** drefinesD2:

assumes dr: drefines $\varphi$ $G$ $A$ $B$
and uP: unitary $P$
and uQ: unitary $Q$
and wpA: $P \vdash \text{wp } A\ Q$
and G: $G$ $s$

shows ($P \circ \varphi$) $s$ $\leq$ wp $B$ (Q o $\varphi$) $s$

**proof** –

from uP have $0 \leq$ ($P \circ \varphi$) $s$ unfolding o-def by(blast)
with G have ($P \circ \varphi$) $s$ = («$G» &\& (P o $\varphi$)) $s$
  by(simp add:exp-conj-def)
also from assms have ... $\leq$ wp $B$ (Q o $\varphi$) $s$ by(blast)
finally show ($P \circ \varphi$) $s$ $\leq$ ...

qed

This additional form is sometimes useful:

**lemma** drefinesD3:

assumes dr: drefines $\varphi$ $G$ $a$ $b$
and G: $G$ $s$
and uQ: unitary $Q$
and wa: well-def $a$

shows wp $a$ $Q$ ($\varphi$ $s$) $\leq$ wp $b$ (Q o $\varphi$) $s$

**proof** –

let $?L\ s' = wp\ a\ Q\ s'$
from uQ wa have sL: sound $?L$ by(blast)
from uQ wa have bL: bounded-by 1 $?L$ by(blast)

have $?L\ \vdash\ $?L$ by(simp)
with sL and bL and assms
show $?\text{thesis}$
  by(blast intro:drefinesD2[OF dr, where P=?L, simplified])
4.9. THE ALGEBRA OF PGCL

qed

lemma drefinesI[intro]:
\[ P \stackrel{\text{unitary}}{\vdash} P;\text{unitary } Q; P \vdash A Q \implies G \&\& (P \circ \varphi) \vdash wp B (Q \circ \varphi) \]
defines \( \varphi \) \( G A B \)
unfolding drefines-def by(blast)

Use \( G \) as an assumption, when showing refinement:

lemma drefinesI2:
fixes \( A: \text{'a prog} \)
and \( B: \text{'b prog} \)
and \( \varphi: \text{'b }\Rightarrow\text{'a} \)
assumes \( wB: \text{well-def } B \)
and \( \text{withAs: } P Q s \Rightarrow G s \Rightarrow wp A Q \)
shows drefines \( \varphi \) \( G A B \)
proof
fix \( P \) and \( Q \)
assume \( uP: \text{unitary } P \)
and \( uQ: \text{unitary } Q \)
and \( \text{wpA: } P \vdash wp A Q \)

hence \( \forall s. G s \Rightarrow (P \circ \varphi) s \leq wp B (Q \circ \varphi) s \)
using \( \text{withAs by(blast)} \)
moreover
from \( uQ \) have unitary \( (Q \circ \varphi) \)
unfolding o-def by(blast)
moreover
from \( uP \) have unitary \( (P \circ \varphi) \)
unfolding o-def by(blast)
ultimately
show \( G \&\& (P \circ \varphi) \vdash wp B (Q \circ \varphi) \)
using \( wB \) by(blast intro:entails-pconj-assumption)

qed

lemma dr-strengthen-guard:
fixes \( a: \text{'s prog and } b: \text{'t prog} \)
assumes \( fg: \forall s. F s \Rightarrow G s \)
and \( \text{drab: drefines } \varphi \) \( G a b \)
shows drefines \( \varphi \) \( F a b \)
proof(intro drefinesI)
fix \( P Q: \text{'s expect} \)
assume \( uP: \text{unitary } P \) and \( uQ: \text{unitary } Q \)
and \( \text{wp: } P \vdash wp a Q \)
from \( fg \) have \( \forall s. F s \leq G s \) by(simp add:embed-bool-def)

hence \( (F \&\& (P \circ \varphi)) \vdash (G \&\& (P \circ \varphi)) \) by(auto intro:pconj-mono le-funI)
Probabilistic correspondence, \textit{pcorres}, is equality on distribution transformers, modulo a guard. It is the analogue, for data refinement, of program equivalence for program refinement.

\textbf{definition}

\texttt{pcorres \:: ('}b \Rightarrow 'a) \Rightarrow \texttt{bool \Rightarrow 'b prog} \Rightarrow \texttt{'a prog}

\textit{where}

\texttt{pcorres} \varphi \ G \ A \ B \leftrightarrow \forall Q. \text{unitary} \ Q \implies ('G' \ & \ wp \ A \ Q \ o \ \varphi) = ('G' \ & \ wp \ B \ (Q \ o \ \varphi))

\textbf{lemma} \texttt{pcorresI}:

\[ \forall Q. \text{unitary} \ Q \implies ('G' \ & \ wp \ A \ Q \ o \ \varphi) = ('G' \ & \ wp \ B \ (Q \ o \ \varphi) \implies pcorres \varphi \ G \ A \ B \]

\textit{by}(simp add:pcorres-def)

Often easier to use, as it allows one to assume the precondition.

\textbf{lemma} \texttt{pcorresI2}[intro]:

\textit{fixes} \ A::'a \texttt{prog} \texttt{and} B::'b \texttt{prog}

\textit{assumes} \ withG: \ \forall Q \ s. \ [\text{unitary} \ Q; \ G \ s \implies wp \ A \ Q \circ \ \varphi = wp \ B \ (Q \circ \ \varphi) \ s]

\textit{and} \ wA: \ \texttt{well-def} \ A

\textit{and} \ wB: \ \texttt{well-def} \ B

\textit{shows} \ pcorres \varphi \ G \ A \ B

\textit{proof}(rule pcorresI, rule ext)

\textit{fix} \ Q::'a \Rightarrow \texttt{real} \texttt{and} s::'b

\textit{assume} \ uQ: \ \texttt{unitary} \ Q

\textit{hence} \ uQ\varphi:: \ \texttt{unitary} \ (Q \circ \ \varphi) \texttt{by(auto)}

\textit{show} \ ('G' \ & \ wp \ A \ Q \circ \ \varphi)) \ s = ('G' \ & \ wp \ B \ (Q \circ \ \varphi)) \ s

\textit{proof}(cases G \ s)

\textit{case} \ True \ \texttt{note} \ this

\textit{moreover}

\textit{from} \ \texttt{well-def-wp-healthy}[OF \ wA] \ uQ \ \texttt{have} \ 0 \leq \ wp \ A \ Q \varphi \ s \texttt{by(blast)}

\textit{moreover}

\textit{from} \ \texttt{well-def-wp-healthy}[OF \ wB] \ uQ\varphi \ \texttt{have} \ 0 \leq \ wp \ B \ (Q \circ \ \varphi) \ s \texttt{by(blast)}

\textit{ultimately show} \ \texttt{thesis}

\textit{using} \ uQ \ \texttt{by}(simp \ add:exp-conj-def \ withG)

\textit{next}

\textit{case} \ False \ \texttt{note} \ this

\textit{moreover}

\textit{from} \ \texttt{well-def-wp-healthy}[OF \ wA] \ uQ \ \texttt{have} \ wp \ A \ Q \varphi \ s \leq 1 \texttt{by(blast)}

\textit{moreover}

\textit{from} \ \texttt{well-def-wp-healthy}[OF \ wB] \ uQ\varphi \ \texttt{have} \ wp \ B \ (Q \circ \ \varphi) \ s \leq 1 \texttt{by(blast dest!:healthy-bounded-byD intro:sound-nexq)}

\textit{ultimately show} \ \texttt{thesis}

\textit{by}(simp \ add:exp-conj-def)

\texttt{qed}

\texttt{qed}
lemma pcorresD:
  \[ \text{pcorres}\,\varphi\,G\,A\,B;\,\text{unitary}\,Q \implies \langle G\rangle & (wp\,A\,Q\,o\,\varphi) = \langle G\rangle & wp\,B\,(Q\,o\,\varphi) \]
  unfolding pcorres-def by(simp)

Again, easier to use if the precondition is known to hold.

lemma pcorresD2:
  assumes \(\text{pc}\): \(\text{pcorres}\,\varphi\,G\,A\,B\)
  and \(uQ\): \(\text{unitary}\,Q\)
  and \(wA\): \(\text{well-def}\,\,A\) and \(wB\): \(\text{well-def}\,\,B\)
  and \(G\): \(G\,s\)
  shows \(wp\,A\,Q\,(\varphi\,s) = wp\,B\,(Q\,o\,\varphi)\,s\)

proof
  from \(uQ\) \(\text{well-def-wp-healthy}\,[OF\,wA]\) have \(\theta \leq wp\,A\,Q\,(\varphi\,s)\) by(auto)
  with \(G\) have \(wp\,A\,Q\,(\varphi\,s) = \langle G\rangle\,s & \text{wp}\,A\,Q\,(\varphi\,s)\) by(simp)
  also { from \(pe\,uQ\) have \(\langle G\rangle\,& (wp\,A\,Q\,o\,\varphi) = \langle G\rangle\,& wp\,B\,(Q\,o\,\varphi)\) by(pcorresD)
    hence \(\langle G\rangle\,s & \text{wp}\,A\,Q\,(\varphi\,s) = \langle G\rangle\,s & \text{wp}\,B\,(Q\,o\,\varphi)\,s\)
    unfolding exp-conj-def o-def by(rule fun-cong)
  }
  also { from \(uQ\) have \(\text{sound}\,Q\) by(auto)
    hence \(\text{sound}\,(Q\,o\,\varphi)\) by(auto \text{ intro:sound-intros})
    with \(\text{well-def-wp-healthy}\,[OF\,wB]\) have \(\theta \leq wp\,B\,(Q\,o\,\varphi)\,s\) by(auto)
    with \(G\) have \(\langle G\rangle\,s & \text{wp}\,B\,(Q\,o\,\varphi)\,s = wp\,B\,(Q\,o\,\varphi)\,s\) by(simp)
  }
  finally show \(\theta\)thesis .
qed

4.9.6 The Algebra of Data Refinement

Program refinement implies a trivial data refinement:

lemma refines-drefines:
  fixes \(a::'s\) \(\text{prog}\)
  assumes \(\text{rab}\): \(a \subseteq b\) and \(\text{wb}\): \(\text{well-def}\,b\)
  shows \(\text{drefines}\,(\lambda\,s\,.\,\text{real})\,G\,a\,b\)
proof(intro drefinesI2 \(\text{wb}\), simp add:o-def)
  fix \(P::'s\Rightarrow\text{real}\) and \(Q::'s\Rightarrow\text{real}\) and \(s::'s\)
  assume \(sQ\): \(\text{unitary}\,Q\)
  assume \(P\,\vdash\,wp\,a\,Q\) hence \(P\,s \leq wp\,a\,Q\,s\) by(auto)
  also from \(\text{rab}\,s\,Q\) 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 $A::\text{\textquoteleft a prog and B::\textquoteleft b prog and C::\textquoteleft c prog}$
assumes $\text{drAB}: \text{drefines } \varphi \ G\ A\ B$
and $\text{drBC}: \text{drefines } \varphi' \ G'\ B\ C$
and $\text{Gimp}: \land s. \ G'\ s \Rightarrow \ G\ (\varphi'\ s)$
shows $\text{drefines } (\varphi\ o\ \varphi')\ G'\ A\ C$

proof (rule drefinesI)
fix $P::\text{\textquoteleft a} \Rightarrow \text{real}$ and $Q::\text{\textquoteleft a} \Rightarrow \text{real}$ and $s::\text{\textquoteleft a}$
assume $uP$: unitary $P$ and $uQ$: unitary $Q$
and $wpA$: $P \vdash wp\ A\ Q$

have $\langle G'\rangle\ &\& \langle G\ o\ \varphi'\rangle = \langle G'\rangle$
proof (rule ext, unfold exp-conj-def)
fix $x$
show $\langle G'\rangle\ x\ &\& \langle G\ o\ \varphi'\rangle\ x = \langle G'\rangle\ x$ (is $?X$)
proof (cases $G'\ x$)
case False then show $?X$ by (simp)
next
case True
moreover
with $\text{Gimp}$ have $(G\ o\ \varphi')\ x$ by (simp add: o-def)
ultimately
show $?X$ by (simp)
qed

with $uP$
have $\langle G'\rangle\ &\& (P\ o\ (\varphi\ o\ \varphi')) = \langle G'\rangle\ &\& ((\langle G'\rangle\ &\& (P\ o\ \varphi))\ o\ \varphi')$
by (simp add: exp-conj-assoc o-assoc)
also {
from $uP\ uQ\ wpA\ and\ \text{drAB}$
have $\langle G'\rangle\ &\& (P\ o\ \varphi) \vdash wp\ B\ (Q\ o\ \varphi)$
by (blast intro: drefinesD)
with $\text{drBC\ and\ uP\ uQ}$
have $\langle G'\rangle\ &\& ((\langle G'\rangle\ &\& (P\ o\ \varphi))\ o\ \varphi') \vdash wp\ C\ ((Q\ o\ \varphi)\ o\ \varphi')$
by (blast intro: unitary-intros drefinesD)
}
finally
show $\langle G'\rangle\ &\& (P\ o\ (\varphi\ o\ \varphi')) \vdash wp\ C\ (Q\ o\ (\varphi\ o\ \varphi'))$
by (simp add: o-assoc)
qed

Data refinement composes with program refinement:

lemma pr-dr-trans[trans]:
assumes $\text{prAB}: A \sqsubseteq B$
and $\text{drBC}: \text{drefines \varphi \ G B C}$
shows $\text{drefines \varphi \ G A C}$
proof (rule drefinesI)
fix $P$ and $Q$
assume $uP$: unitary $P$
and $uQ$: unitary $Q$
and $wpA$: $P \vdash wp A Q$

note $wpA$
also from $uQ$ and $prAB$ have $wp A Q \vdash wp B Q$ by (blast)
finally have $P \vdash wp B Q$.

with $uP$ $uQ$ $drBC$
show $«G» \&\& (P o \varphi) \vdash wp C (Q o \varphi)$ by (blast intro: drefinesD)
qed

lemma dr-pr-trans [trans]:
assumes $drAB$: drefines $\varphi$ $G A B$
assumes $prBC$: $B \subseteq C$
shows drefines $\varphi$ $G A C$
proof (rule drefinesI)
fix $P$ and $Q$
and $uQ$: unitary $Q$
and $wpA$: $P \vdash wp A Q$

with $drAB$ have $«G» \&\& (P o \varphi) \vdash wp B (Q o \varphi)$ by (blast intro: drefinesD)
also from $uQ$ $prBC$ have $... \vdash wp C (Q o \varphi)$ by (blast)
finally show $«G» \&\& (P o \varphi) \vdash ...$.
qed

If the projection $\varphi$ commutes with the transformer, then data refinement is reflexive:

lemma dr-refl:
assumes $wa$: well-def $a$
and $comm$: $\forall Q. unitary Q \Rightarrow wp a Q o \varphi \vdash wp a (Q o \varphi)$
shows drefines $\varphi$ $G a a$
proof (intro drefinesI2 $wa$)
fix $P$ and $Q$ and $s$
assume $wp$: $P \vdash wp a Q$
assume $uQ$: unitary $Q$

have $(P o \varphi) s = P (\varphi s)$ by (simp)
also from $wp$ have $... \leq wp a Q (\varphi s)$ by (blast)
also {
from $comm$ $uQ$ have $wp a Q o \varphi \vdash wp a (Q o \varphi)$ by (blast)
  hence $(wp a Q o \varphi) s \leq wp a (Q o \varphi) s$ by (blast)
  hence $wp a Q (\varphi s) \leq ...$ by (simp)
}
finally show $(P o \varphi) s \leq wp a (Q o \varphi) s$.
qed
Correspondence implies data refinement

**lemma pcorres-drefine:**
- **assumes**
  - $\text{corres}: \text{pcorres } \varphi \ G \ A \ C$
  - $\text{wC}: \text{well-def } C$
- **shows**
  - $\text{drefines } \varphi \ G \ A \ C$

**proof**

1. Fix $P$ and $Q$
2. Assume $uP$: unitary $P$ and $uQ$: unitary $Q$
3. From $\text{wpA}$ have $P \circ \varphi \vdash wp A Q \circ \varphi$ by $(\text{simp add: o-def le-fun-def})$
4. Hence $\{G\} \& \& (P \circ \varphi) \vdash \{G\} \& \& (wp A Q \circ \varphi)$
5. By $(\text{rule exp-conj-mono-right})$
6. Also from $\text{corres } uQ$
   - Have ... $= \{G\} \& \& (wp C (Q o \varphi))$ by $(\text{rule pcorresD})$
7. Also
   - Have ...
   - $\vdash wp C (Q o \varphi)$
8. Proof $(\text{rule le-funI})$
   - Fix $s$
   - From $uQ$ have unitary $(Q o \varphi)$ by $(\text{rule unitary-intros})$
   - With $\text{well-def-wp-healthy}(\text{OF } wC)$ have $nn-wpC: 0 \leq wp C (Q o \varphi) s \leq wp C (Q o \varphi) s$
   - By $(\text{auto})$
   - Case $\text{True}$
   - With $nn-wpC$ show ?thesis by $(\text{simp add: exp-conj-def})$
9. Next
   - Case $\text{False}$ note this
   - Moreover {
     - From $uQ$ have unitary $(Q o \varphi)$ by $(\text{simp})$
     - With $\text{well-def-wp-healthy}(\text{OF } wC)$ have $wp C (Q o \varphi) s \leq 1$
   }
   - Moreover note $nn-wpC$
10. Ultimately show ?thesis by $(\text{simp add: exp-conj-def})$
11. Qed
12. Qed
13. Finally show $\{G\} \& \& (P o \varphi) \vdash wp C (Q o \varphi)$.
14. Qed

Any data refinement of a deterministic program is correspondence. This is the analogous result to that relating program refinement and equivalence.

**lemma drefines-determ:**
- **fixes** $a::'a \ prog$ and $b::'b \ prog$
- **assumes**
  - $da: \text{determ } (wp a)$
  - $wa: \text{well-def } a$
  - $wb: \text{well-def } b$
  - $dr: \text{drefines } \varphi \ G \ a \ b$
- **shows**
  - $\text{pcorres } \varphi \ G \ a \ b$

The proof follows exactly the same form as that for program refinement: Assuming that correspondence doesn’t hold, we show that $wp b$ is not feasible, and thus not
healthy, contradicting the assumption.

**proof (rule pcorresI, rule contrapos-pp)**

```isar
from \( wb \) have feasible \((wp\ b )\) by(auto)
```

**note** \( ha = \text{well-def-wp-healthy}[OF \ wo]\)

**note** \( hv = \text{well-def-wp-healthy}[OF \ wb]\)

From refinement, \( \langle G \rangle \) \&\& \((wp\ a \ Q \ o \ \varphi )\) lies below \( \langle G \rangle \) \&\& \((wp\ b \ (Q \ o \ \varphi )\).

From ha \( uQ\)

**have** : \( \langle G \rangle \) \&\& \((wp\ a \ Q \ o \ \varphi \) \( \vdash \) \( wp\ b \ (Q \ o \ \varphi )\) by(blast intro!:drefinesD[OF dr])

**have** \( le : \langle G \rangle \) \&\& \((wp\ a \ Q \ o \ \varphi \) \( \vdash \) \( \langle G \rangle \) \&\& \((wp\ b \ (Q \ o \ \varphi )\)

**unfolding** exp-conj-def

**proof (rule le-funI)**

```isar
fix \( s \)

from gle have \( \langle G \rangle \ s \ &\ & (wp\ a \ Q \ o \ \varphi ) \ s \leq \ wp\ b \ (Q \ o \ \varphi ) \ s\)

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

hence \( \langle G \rangle \ s \ &\ & (wp\ a \ Q \ o \ \varphi ) \ s \leq \langle G \rangle \ s \ &\ & wp\ b \ (Q \ o \ \varphi ) \ s\)

by(auto intro!:pconj- mono)

moreover from \( uQ \) ha have \( wp\ a \ Q \ (\varphi \ s) \leq 1\)

by(auto dest: healthy-bounded-byD)

moreover from \( uQ \) ha have \( 0 \leq wp\ a \ Q \ (\varphi \ s)\)

by(auto)

ultimately

show \( \langle G \rangle \ s \ &\ & wp\ a \ Q \ (\varphi \ s) \ s \leq \langle G \rangle \ s \ &\ & wp\ b \ (Q \ o \ \varphi ) \ s\)

by(simp add:pconj- assoc)

qed
```

If the programs do not correspond, the terms must differ somewhere, and given the previous result, the second must be somewhere strictly larger than the first:

**have** \( \text{rule: } \exists s. \ (\langle G \rangle \ &\ & (wp\ a \ Q \ o \ \varphi )\ s < (\langle G \rangle \ &\ & wp\ b \ (Q \ o \ \varphi )\ s)\)

**proof (rule contrapos-np[OF ne], rule ext, rule antisym)**

```isar
fix \( s \)

from le show \( (\langle G \rangle \ &\ & (wp\ a \ Q \ o \ \varphi )\ s \leq (\langle G \rangle \ &\ & wp\ b \ (Q \ o \ \varphi )\ s)\)

by(blast)
```

**next**

```isar
fix \( s \)

assume \( \neg (\exists s. \ (\langle G \rangle \ &\ & (wp\ a \ Q \ o \ \varphi )\ s < (\langle G \rangle \ &\ & wp\ b \ (Q \ o \ \varphi )\ s)\)

thus \( (\langle G \rangle \ &\ & (wp\ b \ (Q \ o \ \varphi ))\ s \leq (\langle G \rangle \ &\ & (wp\ a \ Q \ o \ \varphi )\ s)
```
by\((simp\ add:not-less)\)

qed

from\ this\ obtain\ \(s\)\ where\ less-s;
\((\langle G\rangle\ &\ wp\ a\ Q\circ\ \varphi\rangle)\ s\ <\ (\langle G\rangle\ &\ wp\ b\ (Q\circ\ \varphi)\rangle)\ s\)
by\(\text{blast}\)

The\ transformers\ themselves\ must\ differ\ at\ this\ point:

hence\ larger:\ wp\ a\ Q\ (\varphi\ s)\ <\ wp\ b\ (Q\circ\ \varphi)\ s
proof(cases \(G\ s\))

case \(True\)
moreover\ from\ ha\ uQ\ have\ \(0\ \leq\ wp\ a\ Q\ (\varphi\ s)\)
by\(\text{blast}\)
moreover\ from\ hb\ uQ\ \varphi\ have\ \(0\ \leq\ wp\ b\ (Q\circ\ \varphi)\ s\)
by\(\text{blast}\)
moreover\ note\ less-s
ultimately\ show\ \?thesis\ by\(\text{simp\ add:exp-conj-def}\)

next

case \(False\)
moreover\ from\ ha\ uQ\ have\ wp\ a\ Q\ (\varphi\ s)\ \leq\ 1
by\(\text{blast}\)
moreover\ {\}
from\ uQ\ have\ bounded-by\ \(1\ (Q\circ\ \varphi)\)
by\(\text{blast}\)
moreover\ from\ unitary-sound\(\text{OF uQ}\)
have\ sound\ (Q\circ\ \varphi)\ by\(\text{auto}\)
ultimately\ have\ wp\ b\ (Q\circ\ \varphi)\ s\ \leq\ 1
using\ hb\ by\(\text{auto}\)

}\
moreover\ note\ less-s
ultimately\ show\ \?thesis\ by\(\text{simp\ add:exp-conj-def}\)
qed

from\ less-s\ have\ (\langle G\rangle\ &\ wp\ a\ Q\circ\ \varphi\rangle)\ s\ \neq\ (\langle G\rangle\ &\ wp\ b\ (Q\circ\ \varphi)\rangle)\ s\)
by\(\text{force}\)

\(G\)\ must\ also\ hold,\ as\ otherwise\ both\ would\ be\ zero.

hence\ \(G\)-s: \(G\ s\)
proof\(\text{rule contrapos-np}\)
assume\ \(nG:\ \neg\ G\ s\)
moreover\ from\ ha\ uQ\ have\ wp\ a\ Q\ (\varphi\ s)\ \leq\ 1
by\(\text{blast}\)
moreover\ {\}
from\ uQ\ have\ bounded-by\ \(1\ (Q\circ\ \varphi)\)
by\(\text{blast}\)
moreover\ from\ unitary-sound\(\text{OF uQ}\)
have\ sound\ (Q\circ\ \varphi)\ by\(\text{auto}\)
ultimately\ have\ wp\ b\ (Q\circ\ \varphi)\ s\ \leq\ 1
using\ hb\ by\(\text{auto}\)

}\
alternatively
4.9. THE ALGEBRA OF PGCL

\[ \text{show} \ (\langle G \rangle \ & \& (\text{wp } a \ Q \circ \varphi)) \ s = (\langle G \rangle \ & \& \text{wp } b \ (Q \circ \varphi)) \ s \]
\[ \text{by(simp add:exp-conj-def)} \]
\[ \text{qed} \]

Take a carefully constructed expectation:

\begin{verbatim}
let ?Qc = \lambda s. bound-of Q - Q s
have bQc: bounded-by 1 ?Qc
  proof (rule bounded-byI)
    fix s
    from uQ have bound-of Q ≤ 1 and 0 ≤ Q s by(auto)
    thus bound-of Q - Q s ≤ 1 by(auto)
  qed
have sQc: sound ?Qc
  proof (rule soundI)
    from bQc show bounded ?Qc by(auto)
    show nneg ?Qc
      proof (rule nnegI)
        fix s
        from uQ have Q s ≤ bound-of Q by(auto)
        thus 0 ≤ bound-of Q - Q s by(auto)
      qed
  qed
qed
\end{verbatim}

By the maximality of wp a, wp b must violate feasibility, by mapping s to something strictly greater than bound-of Q.

\begin{verbatim}
from uQ have 0 ≤ bound-of Q by(auto)
with da have bound-of Q = wp a (\lambda s. bound-of Q) (\varphi s)
  by(simp add:maximalD determ-maximalD)
also have wp a (\lambda s. bound-of Q) (\varphi s) = wp a (\lambda s. Q s + ?Qc s) (\varphi s)
  by(simp)
also {
  from da have additive (wp a) by(blast)
    with uQ sQc
    have wp a (\lambda s. Q s + ?Qc s) (\varphi s) =
      wp a Q (\varphi s) + wp a ?Qc (\varphi s) by(subst additiveD, blast+)
}
also {
  from ha and sQc and bQc
  have \langle G \rangle \ & \& (wp a ?Qc o \varphi) \vdash wp b (\varphi s)
    by(blast intro!:drefinesD[OF dr])
  hence \langle G \rangle \ & \& (wp a ?Qc o \varphi) ≤ wp b (\varphi s)
    by(blast)
  moreover from sQc and ha
  have 0 ≤ wp a (\lambda s. bound-of Q - Q s) (\varphi s)
    by(blast)
  ultimately
  have wp a ?Qc (\varphi s) ≤ wp b (\varphi s)
    using G-s by(simp add:exp-conj-def)
\end{verbatim}
hence \( wp a Q (\varphi s) + wp a ?Qc (\varphi s) \leq wp a Q (\varphi s) + wp b (?Qc o \varphi) s \)
by (rule add-left-mono)
also with larger
have \( wp a Q (\varphi s) + wp b (?Qc o \varphi) s < \)
\( wp b (Q o \varphi) s + wp b (?Qc o \varphi) s \)
by (auto)
finally
have \( wp a Q (\varphi s) + wp a ?Qc (\varphi s) < \)
\( wp b (Q o \varphi) s + wp b (?Qc o \varphi) s . \)
\}
also from \( s Qc \) and \( \text{unitary-sound}[OF uQ] \) and \( s Qc \)
have \( wp b (Q o \varphi) s + wp b (?Qc o \varphi) s \leq \)
\( wp b (\lambda s. (Q o \varphi) s + (?Qc o \varphi) s) s \)
by (blast)
also have \( ... = wp b (\lambda s. \text{bound-of } Q) s \)
by (simp)
finally
show \( \neg \text{feasible} (wp b) \)
proof (rule contrapos-pn)
assume \( fb: \text{feasible} (wp b) \)
have \( \text{bounded-by} (\text{bound-of } Q) (\lambda s. \text{bound-of } Q) \) by (blast)
hence \( \text{bounded-by} (\text{bound-of } Q) (wp b (\lambda s. \text{bound-of } Q)) \)
using \( uQ \) by (blast intro: feasible-boundedD [OF fb])
hence \( wp b (\lambda s. \text{bound-of } Q) s \leq \text{bound-of } Q \) by (blast)
thus \( \neg \text{bound-of } Q < wp b (\lambda s. \text{bound-of } Q) s \) by (simp)
qed
qed

4.9.7 Structural Rules for Correspondence

lemma pcorres-Skip:
\[ \text{pcorres } \varphi \ G \ Skip \ Skip \]
by (simp add: pcorres-def wp-eval)

Correspondence composes over sequential composition.

lemma pcorres-Seq:
fixes \( A::'b \ prog \) and \( B::'c \ prog \)
and \( C::'b \ prog \) and \( D::'c \ prog \)
and \( \varphi::'c \Rightarrow 'b \)
assumes \( \text{pcAB: pcorres } \varphi \ G \ A \ B \)
and \( \text{pcCD: pcorres } \varphi \ H \ C \ D \)
and \( \text{waA: well-def } A \) and \( \text{wbB: well-def } B \)
and \( \text{wcC: well-def } C \) and \( \text{wdD: well-def } D \)
and \( \text{ps2p2: } Q. \ \text{unitary } Q \Rightarrow \langle I \rangle \ \& \ \& wp B Q = wp B (\langle H \rangle \ \& \ \& Q) \)
and \( \text{p1p3: } \langle s. G s \Rightarrow I s \rangle \)
shows \( \text{pcorres } \varphi \ G (A;;C) (B;;D) \)
proof (rule pcorresI)
fix \( Q::'b \Rightarrow \text{real} \)
assume \( uQ:: \text{unitary } Q \)
4.9. THE ALGEBRA OF PGCL

with well-def-wp-healthy[OF wC] have uCQ: unitary (wp C Q) by(auto)
from uQ well-def-wp-healthy[OF wD] have uDQ: unitary (wp D (Q o ϕ))
by(auto dest:unitary-comp)

have p3p1: \( \forall R S. \ [ \text{unitary } R; \text{unitary } S; \langle I \rangle \& \& R = \langle I \rangle \& \& S ] \implies \langle G \rangle \& \& R = \langle G \rangle \& \& S \)

proof(rule ext)
fix \( R::'c \Rightarrow \text{real} \) and \( S::'c \Rightarrow \text{real} \) and \( s::'c \)
assume a3: \( \langle I \rangle \& \& R = \langle I \rangle \& \& S \)
and uR: \text{unitary } R and uS: \text{unitary } S
show \( \langle G \rangle \& \& R \) s = \( \langle G \rangle \& \& S \) s

proof(simp add:exp-conj-def, cases G s)
case False note this
moreover from uR have R s \( \leq 1 \) by(blast)
moreover from uS have S s \( \leq 1 \) by(blast)
ultimately show \( \langle G \rangle \& \& R \) s = \( \langle G \rangle \& \& S \) s
by(simp)
next
case True note p1 = this
with p1p3 have I s by(blast)
with fun-cong[OF a3, where \( x=s \)] have \( 1 \& \& R \) s = \( 1 \& \& S \) s
by(simp add:exp-conj-def)
with p1 show \( \langle G \rangle \& \& R \) s = \( \langle G \rangle \& \& S \) s
by(simp)
qed

show \( \langle G \rangle \& \& (wp (A;;C) Q o ϕ) = \langle G \rangle \& \& wp (B;;D) (Q o ϕ) \)

proof(simp add:wp-eval)
from uCQ pcAB have \( \langle G \rangle \& \& (wp A (wp C Q) o ϕ) = \langle G \rangle \& \& wp B ((wp C Q) o ϕ) \)
by(auto dest:pcorresD)
also have \( \langle G \rangle \& \& wp B ((wp C Q) o ϕ) = \langle G \rangle \& \& wp B (wp D (Q o ϕ)) \)

proof(rule p3p1)
from uCQ well-def-wp-healthy[OF wB] show unitary (wp B (wp C Q o ϕ))
by(auto intro:unitary-comp)
from uDQ well-def-wp-healthy[OF wB] show unitary (wp B (wp D (Q o ϕ)))
by(auto)

from uQ have \( \langle H \rangle \& \& (wp C Q o ϕ) = \langle H \rangle \& \& wp D (Q o ϕ) \)
by(blast intro:pcorresD[OF pcCD])
thus \( \langle I \rangle \& \& wp B (wp C Q o ϕ) = \langle I \rangle \& \& wp B (wp D (Q o ϕ)) \)
by(simp add:p3p2 uCQ uDQ)
qed

finally show \( \langle G \rangle \& \& (wp A (wp C Q) o ϕ) = \langle G \rangle \& \& wp B (wp D (Q o ϕ)) \)

qed

qed
4.9.8 Structural Rules for Data Refinement

**Lemma dr-Skip:**

fixes \( \varphi : c \Rightarrow 'b \)

shows drefines \( \varphi \) G Skip Skip

**Proof:**

\[
\text{proof (intro drefinesI2 wd-intros)} \\
\text{fix } P : 'b \Rightarrow \text{real and } Q : 'b \Rightarrow \text{real and } s : 'c \\
\text{assume } P \vdash \vdash \text{wp Skip } Q \\
\text{hence } (P o \varphi) \ s \leq \text{wp Skip } (Q o \varphi) \ s \text{ by(simp, blast)} \\
\text{thus } (P o \varphi) \ s \leq \text{wp Skip } (Q o \varphi) \ s \text{ by(simp add:wp-eval)}
\]

**Qed**

**Lemma dr-Abort:**

fixes \( \varphi : c \Rightarrow 'b \)

shows drefines \( \varphi \) G Abort Abort

**Proof:**

\[
\text{proof (intro drefinesI2 wd-intros)} \\
\text{fix } P : 'b \Rightarrow \text{real and } Q : 'b \Rightarrow \text{real and } s : 'c \\
\text{assume } P \vdash \vdash \text{wp Abort } Q \\
\text{hence } (P o \varphi) \ s \leq \text{wp Abort } (Q o \varphi) \ s \text{ by(auto)} \\
\text{thus } (P o \varphi) \ s \leq \text{wp Abort } (Q o \varphi) \ s \text{ by(simp add:wp-eval)}
\]

**Qed**

**Lemma dr-Apply:**

fixes \( \varphi : c \Rightarrow 'b \)

assumes commutes: \( f o \varphi = \varphi o g \)

shows drefines \( \varphi \) G (Apply f) (Apply g)

**Proof:**

\[
\text{proof (intro drefinesI2 wd-intros)} \\
\text{fix } P : 'b \Rightarrow \text{real and } Q : 'b \Rightarrow \text{real and } s : 'c \\
\text{assume } \text{wp: } P \vdash \text{wp } (\text{Apply f} ) \ Q \\
\text{hence } P \vdash (Q o f) \text{ by(simp add:wp-eval)} \\
\text{hence } P \ (\varphi \ s) \leq (Q o f) \ (\varphi \ s) \text{ by(blast)} \\
\text{also have } ... = Q ((f o \varphi) \ s) \text{ by(simp)} \\
\text{also with commutes} \\
\text{have } ... = ((Q o \varphi) o g) \ s \text{ by(simp)} \\
\text{also have } ... = \text{wp } (\text{Apply g} ) \ (Q o \varphi) \ s \\
\text{by(simp add:wp-eval)} \\
\text{finally show } (P o \varphi) \ s \leq \text{wp } (\text{Apply g} ) \ (Q o \varphi) \ s \text{ by(simp)}
\]

**Qed**

**Lemma dr-Seq:**

assumes drAB: drefines \( \varphi \) P A B

and drBC: drefines \( \varphi \) Q C D

and wpB: «\( P \) ⇒ wp B «Q»

and wB: well-def B

and wC: well-def C

and wD: well-def D

shows drefines \( \varphi \) P (A;C) (B;D)

**Proof:**

\[
\text{fix } R \text{ and } S
\]
assume \( u \text{R: unitary R} \) and \( u \text{S: unitary S} \)
and \( \text{wpAC}: R \vdash \text{wp} \ (A_1; C) \ S \)

from \( u \text{R} \)
have \( \langle P \rangle \land (R \ o \ \varphi) = \langle P \rangle \land (\langle P \rangle \land (R \ o \ \varphi)) \)
by\((\text{simp add: exp-conj-assoc})\)

also \{ 
from \( \text{well-def-wp-healthy}[\text{OF \ wC}] \ \ u\text{R \ uS} \)
and \( \text{wpAC}[\text{unfolded eval-wp-Seq o-def}] \)
have \( \langle P \rangle \land (R \ o \ \varphi) \vdash \text{wp} \ B \ (wp \ C \ o \ \varphi) \)
by\((\text{auto intro:drefinesD[OF \ drAB]})\)
with \( \text{wpB \ well-def-wp-healthy}[\text{OF \ wC}] \ \ u\text{S} \)
sublinear-sub-conj\([\text{OF \ well-def-wp-sublinear, \ OF \ wB}] \)
have \( \langle P \rangle \land (\langle P \rangle \land (R \ o \ \varphi)) \vdash \text{wp} \ B \ (\langle Q \rangle \land (wp \ C \ o \ \varphi)) \)
by\((\text{auto intro!: entails-combine dest!: unitary-sound})\)
\}

also \{ 
from \( \text{uS \ well-def-wp-healthy}[\text{OF \ wC}] \)
have \( \langle Q \rangle \land (wp \ C \ o \ \varphi) \vdash \text{wp} \ D \ (S \ o \ \varphi) \)
by\((\text{auto intro!: drefinesD[OF \ drBC]})\)
with \( \text{well-def-wp-healthy}[\text{OF \ wB}] \ \ \text{well-def-wp-healthy}[\text{OF \ wC}] \)
well-def-wp-healthy\([\text{OF \ wD}] \ \ \text{and \ unitary-sound}[\text{OF \ uS}] \)
have \( \text{wp} \ B \ (\langle Q \rangle \land (wp \ C \ o \ \varphi)) \nvdash \text{wp} \ B \ (wp \ D \ (S \ o \ \varphi)) \)
by\((\text{blast intro!: mono-transD})\)
\}

finally
show \( \langle P \rangle \land (R \ o \ \varphi) \vdash \text{wp} \ (B_1; D) \ (S \ o \ \varphi) \)
unfolding \( \text{wp-eval o-def} \).

qed

lemma \( \text{dr-repeat} \):
fixes \( \varphi: \ 'a \Rightarrow \ 'b \)
assumes \( \text{dr-ab: drefines \varphi \ G \ a \ b} \)
and \( \text{Gpr: \ 'G \ \vdash \ wp} \ b \ 'G \)
and \( \text{wa: \ well-def \ a} \)
and \( \text{wb: \ well-def \ b} \)
shows \( \text{drefines \varphi \ G \ (repeat \ n \ a) \ (repeat \ n \ b) \ (is \ \?X \ n)} \)
proof\((\text{induct \ n})\)
show \( \?X \ 0 \ \text{by}(\text{simp add: dr-Skip}) \)
fix \( n \)
assume \( \text{IH: \ ?X \ n} \)
thus \( \?X \ (\text{Suc} \ n) \ \text{by}(\text{auto intro!: dr-Seq Gpr assms \ wd-intros}) \)
qed

end
4.10 Structured Reasoning

By linking the algebraic, the syntactic, and the semantic views of computation, we derive a set of rules for decomposing expectation entailment proofs, firstly over the syntactic structure of a program, and secondly over the refinement relation. These rules also form the basis for automated reasoning.

4.10.1 Syntactic Decomposition

lemma wp-Abort:
  \( (\lambda s. 0) \vdash wp \text{Abort} Q \)
  unfolding wp-eval by(simp)

lemma wlp-Abort:
  \( (\lambda s. 1) \vdash wlp \text{Abort} Q \)
  unfolding wp-eval by(simp)

lemma wp-Skip:
  \( P \vdash wp \text{Skip} P \)
  unfolding wp-eval by(blast)

lemma wlp-Skip:
  \( P \vdash wlp \text{Skip} P \)
  unfolding wp-eval by(blast)

lemma wp-Apply:
  \( Q \circ f \vdash wp \text{(Apply f)} Q \)
  unfolding wp-eval by(simp)

lemma wlp-Apply:
  \( Q \circ f \vdash wlp \text{(Apply f)} Q \)
  unfolding wp-eval by(simp)

lemma wp-Seq:
  assumes ent-a: \( P \vdash wp \text{ a} Q \)
  and ent-b: \( Q \vdash wp \text{ b} R \)
  and wa: well-def a
  and wb: well-def b
  and s-Q: sound Q
  and s-R: sound R
  shows \( P \vdash wp \text{ (a ;; b) R} \)
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 \(\vdash wp \text{ a} (wp \text{ b} R) \)
  by(blast intro:healthy-monoD2)
finally show \(?thesis by\)(simp add:wp-eval)

qed

lemma wlp-Seq:
assumes ent-a: \(P \vdash wlp a Q\)
and ent-b: \(Q \vdash wlp b R\)
and wa: well-def a
and wb: well-def b
and u-Q: unitary Q
and u-R: unitary R
shows \(P \vdash wlp (a ;; b) R\)
proof
  note ha = well-def-wlp-nearly-healthy[OF wa]
  note hb = well-def-wlp-nearly-healthy[OF wb]
  note ent-a also from ent-b ha hb u-Q u-R have wlp a Q \(\vdash wlp a (wlp b R)\)
  by(blast intro:nearly-healthy-monoD[OF ha])
finally show \(?thesis by\)(simp add:wp-eval)
qed

lemma wp-PC:
\((\lambda s. P s * wp a Q s + (1 - P s) * wp b Q s) \vdash wp (a \oplus b) Q\)
by(simp add:wp-eval)

lemma wlp-PC:
\((\lambda s. P s * wlp a Q s + (1 - P s) * wlp b Q s) \vdash wlp (a \oplus b) Q\)
by(simp add:wp-eval)

A simpler rule for when the probability does not depend on the state.

lemma PC-fixed:
assumes wpa: \(P \vdash a ab R\)
and wpb: \(Q \vdash b ab R\)
and np: \(0 \leq p\) and bp: \(p \leq 1\)
shows \((\lambda s. p * P s + (1 - p) * Q s) \vdash (a (\lambda s. p) \oplus b) ab R\)
unfolding PC-def
proof(rule le-funI)
  fix s
  from wpa and np have \(p * P s \leq p * a ab R s\)
    by(auto intro:mult-left-mono)
  moreover {
    from bp have \(0 \leq 1 - p\) by(simp)
    with wpb have \((1 - p) * Q s \leq (1 - p) * b ab R s\)
      by(auto intro:mult-left-mono)
  }
  ultimately show \(p * P s + (1 - p) * Q s \leq p * a ab R s + (1 - p) * b ab R s\)
    by(rule add-mono)
qed
lemma \texttt{wp-PC-fixed}: \\
\[
[ P \vdash wp a R; Q \vdash wp b R; 0 \leq p; p \leq 1 ] \implies \\
\lambda s. p * P s + (1 - p) * Q s \vdash wp (\lambda s. p \oplus b) R \\
\text{by (simp add: wp-def PC-fixed)}
\]

lemma \texttt{wlp-PC-fixed}: \\
\[
[ P \vdash wlp a R; Q \vdash wlp b R; 0 \leq p; p \leq 1 ] \implies \\
\lambda s. p * P s + (1 - p) * Q s \vdash wlp (\lambda s. p \oplus b) R \\
\text{by (simp add: wlp-def PC-fixed)}
\]

lemma \texttt{wp-DC}: \\
\[
(\lambda s. \min (wp a Q s) (wp b Q s)) \vdash wp (a \sqcap b) Q \\
\text{unfolding wp-eval by (simp)}
\]

lemma \texttt{wlp-DC}: \\
\[
(\lambda s. \min (wlp a Q s) (wlp b Q s)) \vdash wlp (a \sqcap b) Q \\
\text{unfolding wp-eval by (simp)}
\]

Combining annotations for both branches:

\textbf{lemma \textit{DC-split}}: \\
\textit{fixes a::'s prog and b} \\
\textit{assumes \texttt{wp}a: \texttt{P} \vdash a ab R} \\
\textit{and \texttt{wp}b: \texttt{Q} \vdash b ab R} \\
\textit{shows (\lambda s. \min (P s) (Q s)) \vdash (a \sqcap b) ab R} \\
\textit{unfolding DC-def} \\
\textit{proof (rule le-funI)} \\
\textit{fix s} \\
\textit{from \texttt{wp}a \texttt{wp}b} \\
\textit{have P s \leq a ab R s and Q s \leq b ab R s by (auto)} \\
\textit{thus \min (P s) (Q s) \leq \min (a ab R s) (b ab R s) by (auto)} \\
\textit{qed}

\textbf{lemma \textit{wp-DC-split}}: \\
\[
[ P \vdash wp prog R; Q \vdash wp prog' R ] \implies \\
(\lambda s. \min (P s) (Q s)) \vdash wp (prog \sqcap prog') R \\
\text{by (simp add: wp-def DC-split)}
\]

\textbf{lemma \textit{wlp-DC-split}}: \\
\[
[ P \vdash wlp prog R; Q \vdash wlp prog' R ] \implies \\
(\lambda s. \min (P s) (Q s)) \vdash wlp (prog \sqcap prog') R \\
\text{by (simp add: wlp-def DC-split)}
\]

\textbf{lemma \textit{wp-DC-split-same}}: \\
\[
[ P \vdash wp prog Q; P \vdash wp prog' Q ] \implies P \vdash wp (prog \sqcap prog') Q \\
\text{unfolding wp-eval by (blast intro:min.boundedI)}
\]

\textbf{lemma \textit{wlp-DC-split-same}}: \\
\[
[ P \vdash wlp prog Q; P \vdash wlp prog' Q ] \implies P \vdash wlp (prog \sqcap prog') Q \\
\text{unfolding wp-eval by (blast intro:min.boundedI)}
\]
4.10. STRUCTURED REASONING

lemma SetPC-split:
  fixes f::'a ⇒ 'y prog
  and p::'y ⇒ 'x ⇒ real
  assumes rec: ∀x s. x ∈ supp (p s) ⇒ P x ⊨ f x ab Q
  and nnp: ∀s. nneg (p s)
  shows (∀s. ∑ x ∈ supp (p s). p s x * P x s) ⊨ SetPC f p ab Q
unfolding SetPC-def
proof (rule le-funI)
  fix s
  from rec have ∀x. x ∈ supp (p s) ⇒ P x s ≤ f x ab Q s by (blast)
  moreover from nnp have ∀x. 0 ≤ p s x by (blast)
  ultimately have ∀x. x ∈ supp (p s) ⇒ p s x * P x s ≤ p s x * f x ab Q s
    by (blast intro:mult-left-mono)
  thus (∑ x ∈ supp (p s). p s x * P x s) ≤ (∑ x ∈ supp (p s). p s x * f x ab Q s)
    by (rule sum-mono)
qed

lemma wp-SetPC-split:
[ ∀x s. x ∈ supp (p s) ⇒ P x ⊨ wp (f x) Q; ∀s. nneg (p s) ] ⇒
(∀s. ∑ x ∈ supp (p s). p s x * P x s) ⊨ wp (SetPC f p) Q
by (simp add:wp-def SetPC-split)

lemma wlp-SetPC-split:
[ ∀x s. x ∈ supp (p s) ⇒ P x ⊨ wlp (f x) Q; ∀s. nneg (p s) ] ⇒
(∀s. ∑ x ∈ supp (p s). p s x * P x s) ⊨ wlp (SetPC f p) Q
by (simp add:wlp-def SetPC-split)

lemma wp-SetDC-split:
[ ∀s x. x ∈ S s ⇒ P x ⊨ wp (f x) Q; ∀s. S s ≠ {} ] ⇒
P ⊨ wp (SetDC f S) Q
by (rule le-funI, unfold wp-eval, blast intro!:cInf-greatest)

lemma wlp-SetDC-split:
[ ∀s x. x ∈ S s ⇒ P x ⊨ wlp (f x) Q; ∀s. S s ≠ {} ] ⇒
P ⊨ wlp (SetDC f S) Q
by (rule le-funI, unfold wp-eval, blast intro!:cInf-greatest)

lemma wp-SetDC:
  assumes wp: ∀s x. x ∈ S s ⇒ P x ⊨ wp (f x) Q
  and nc: ∀s. S s ≠ {} 
  and sP: ∀x. sound (P x)
  shows (∀s. Inf ((∀x. P x s) ∩ S s)) ⊨ wp (SetDC f S) Q
using assms by (intro le-funI, simp add:wp-eval, blast intro!:cInf-mono)

lemma wlp-SetDC:
  assumes wp: ∀s x. x ∈ S s ⇒ P x ⊨ wlp (f x) Q 
  and nc: ∀s. S s ≠ {} 
  and sP: ∀x. sound (P x)
shows \( (\lambda x. \text{Inf} ((\lambda x. P x s) \cdot S s)) \vdash \text{wlp} (\text{SetDC} f S) Q \)
using assms by (intro le-funI, simp add: wp-eval, blast intro!: cInf-mono)

**Lemma wp-Embed:**
\[
P \vdash t Q \implies P \vdash \text{wp} (\text{Embed} t) Q
\]
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:**
\[
[ \forall s. P s \leq \text{wp} (a (f s)) Q s ] \implies P \vdash \text{wp} (\text{Bind} f a) Q
\]
by (auto simp: wp-def Bind-def)

**Lemma wlp-Bind:**
\[
[ \forall s. P s \leq \text{wlp} (a (f s)) Q s ] \implies P \vdash \text{wlp} (\text{Bind} f a) Q
\]
by (auto simp: wlp-def Bind-def)

**Lemma wp-repeat:**
\[
[ P \vdash \text{wp} a Q; Q \vdash \text{wp} (\text{repeat} n a) R; \text{well-def} a; \text{sound} Q; \text{sound} R ] \implies P \vdash \text{wp} (\text{repeat} (\text{Suc} n) a) R
\]
by (auto intro!: wp-Seq wd-intros)

**Lemma wlp-repeat:**
\[
[ P \vdash \text{wlp} a Q; Q \vdash \text{wlp} (\text{repeat} n a) R; \text{well-def} a; \text{unitary} Q; \text{unitary} R ] \implies P \vdash \text{wlp} (\text{repeat} (\text{Suc} n) a) R
\]
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:**
\[
\text{entails-strengthen-post[where } t=\text{wp a for } a]\]

**Lemma wp-strengthen-post:**
\[
P \vdash \text{wp} a Q \implies \text{nearly-healthy} (\text{wp} a) \implies \text{unitary} R \implies Q \vdash R \implies \text{unitary} Q \implies P \vdash \text{wp} a R
\]
by (blast intro: entails-trans)

**Lemmas wp-weaken-pre:**
\[
\text{entails-weaken-pre[where } t=\text{wp a for } a]\]

**Lemmas wlp-weaken-pre:**
\[
\text{entails-weaken-pre[where } t=\text{wlp a for } a]\]

**Lemmas wp-scale:**
4.11. LOOP TERMINATION

entails-scale[where t=wp a for a, OF - well-def-wp-healthy]

4.10.2 Algebraic Decomposition

Refinement is a powerful tool for decomposition, belied by the simplicity of the rule. This is an axiomatic formulation of refinement (all annotations of the a are annotations of b), rather than an operational version (all traces of b are traces of a).

lemma wp-refines:
\[ a \sqsubseteq b; P \Vdash wp a Q; sound Q \implies P \vdash wp b Q \]
by(auto intro:entails-trans)

lemmas wp-drefines = drefinesD

4.10.3 Hoare triples

The Hoare triple, or validity predicate, is logically equivalent to the weakest-precondition entailment form. The benefit is that it allows us to define transitivity rules for computational (also/finally) reasoning.

definition wp-valid :: (′a ⇒ real) ⇒ ′a prog ⇒ (′a ⇒ real) ⇒ bool (⟦-⟧ - ⟦-⟧p)
where
wp-valid P prog Q ≡ P \vdash wp prog Q

lemma wp-validI:
\[ P \vdash wp prog Q \implies ⟦P⟧ prog ⟦Q⟧p \]
unfolding wp-valid-def by(assumption)

lemma wp-validD:
\[ ⟦P⟧ prog ⟦Q⟧p \implies P \vdash wp prog Q \]
unfolding wp-valid-def by(assumption)

lemma valid-Seq:
\[ ⟦P⟧ a ⟦Q⟧p; ⟦Q⟧ b ⟦R⟧p; well-def a; well-def b; sound Q; sound R \implies ⟦P⟧ a :: b ⟦R⟧p \]
unfolding wp-valid-def by(rule wp-Seq)

We make it available to the computational reasoner:

declare valid-Seq[trans]

end

4.11 Loop Termination

theory Termination imports Embedding StructuredReasoning Loops begin
Termination for loops can be shown by classical means (using a variant, or a measure function), or by probabilistic means: We only need that the loop terminates \textit{with probability one}.

\subsection{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).

\textbf{lemma} \texttt{maximal-Seq-term}: 
\begin{verbatim}
fixes r::'s prog and s::'s prog
assumes mr: maximal (wp r)
    and ws: well-def s
    and ts: (λs. 1) ⊢ wp s (λs. 1)
shows (λs. 1) ⊢ wp (r ;; s) (λs. 1)
proof

- note hs = well-def-wp-healthy[OF ws]
have wp s (λs. 1) = (λs. 1)
proof (rule antisym)

- show (λs. 1) ⊢ wp s (λs. 1) by (rule ts)
- have bounded-by 1 (wp s (λs. 1)) by (auto intro!:healthy-bounded-byD[OF hs])
- thus wp s (λs. 1) ⊢ (λs. 1) by (auto intro!:le-funI)

qed

with mr show ?thesis
by (simp add: wp-eval embed-bool-def maximalD)
qed
\end{verbatim}

From any state where the guard does not hold, a loop terminates in a single step.

\textbf{lemma} \texttt{term-onestep}: 
\begin{verbatim}
assumes wb: well-def body
shows «N G» ⊢ wp do G → body od (λs. 1)
proof (rule le-funI)

- note hb = well-def-wp-healthy[OF wb]
fix s
show «N G» s ≤ wp do G → body od (λs. 1) s
proof (cases G s, simp-all add:wp-loop-nguard hb)
    from hb have sound (wp do G → body od (λs. 1))
    by (auto intro!:healthy-sound[OF healthy-wp-loop])
    thus 0 ≤ wp do G → body od (λs. 1) s by (auto)

qed

\end{verbatim}

\subsection{4.11.2 Classical Termination}

The first non-trivial termination result is quite standard: If we can provide a natural-number-valued measure, that decreases on every iteration, and implies termination on reaching zero, the loop terminates.
Lemma loop-term-nat-measure-noinv:

Fixes m :: 's ⇒ nat and body :: 's prog

Assumes wb: well-def body

And guard: ∃s, m ∗ s = 0 → ¬ G s

And variant: ∃n. (λs. m ∗ s = Suc n) ⊢ wp body (λs. m ∗ s = n)

Shows λs. 1 ⊢ wp do G → body od (λs. 1)

Proof

Note hb = well-def-wp-healthy[OF wb]

Have ∃n. (∀s. m ∗ s = n → 1 ≤ wp do G → body od (λs. 1) s)

Proof (clarify)

Fix n

Show ∀s. m ∗ s = 0 → 1 ≤ wp do G → body od (λs. 1) s

Proof (clarify)

Fix s

Assume m ∗ s = 0

With guard have ¬ G s by (blast)

With hb show 1 ≤ wp do G → body od (λs. 1) s

By (simp add: wp-loop-nguard)

Qed

Assume IH: ∀s. m ∗ s = n → 1 ≤ wp do G → body od (λs. 1) s

Hence IH': ∀s. m ∗ s = n → 1 ≤ wp do G → body od (λs. True) s

By (simp add: embed-bool-def)

Have ∀s. m ∗ Suc n → 1 ≤ wp do G → body od (λs. True) s

Proof (intro fold-premise healthy-intros hb, rule le-funI)

Fix s

Show (λs. m ∗ Suc n) s ≤ wp do G → body od (λs. True) s

Proof (cases G s)

Case False

Hence 1 = (λs. True) s by (auto)

Also from wb have ... ≤ wp do G → body od (λs. 1) s

By (rule le-funD[OF term-onestep])

Finally show ?thesis by (simp add: embed-bool-def)

Next

Case True note G = this

From IH' have (λs. m ∗ Suc n) ⊢ wp do G → body od (λs. True)

By (blast intro:use-premise healthy-intros hb)

With variant wb

Have (λs. m ∗ Suc n) ⊢ wp (body ;; do G → body od) (λs. True)

By (blast intro: wp-Seq wp-intros)

Hence (λs. m ∗ Suc n) s ≤ wp (body ;; do G → body od) (λs. True) s

By (auto)

Also from hb G have ... = wp do G → body od (λs. True) s

By (simp add: wp-loop-guard)

Finally show ?thesis .

Qed

Qed

Thus ∀s. m ∗ Suc n → 1 ≤ wp do G → body od (λs. 1) s

By (simp add: embed-bool-def)

Qed
thus \( \text{thesis by (auto)} \)
\[\text{qed}\]

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** \( m :: 's \Rightarrow \texttt{nat} \) and \( \text{body :: 's prog} \)
- **Assumes** \( \text{wb: well-def body} \)
- **Guard**: \( \forall s. m s = 0 \rightarrow \neg G s \)
- **Variant**: \( \forall n. \exists s. m s = \text{Suc } n \wedge \text{hit} \)
- **Inv**: \( \text{wp-inv } G \text{ body } I \)

**Proof**

- **Note** \( hh = \text{well-def-wp-healthy} [\text{OF } \text{wb}] \)
- **Note** \( scb = \text{sublinear-sub-conj} [\text{OF } \text{well-def-wp-sublinear}, \text{OF } \text{wb}] \)
- **Have** \( \text{I } \vdash \text{wp do } G \rightarrow \	ext{body od } \lambda s. \text{True} \)

**Proof (rule use-premise, intro healthy-intros \( hh \))**

- **Fix** \( s \)
- **Have** \( \forall n. \forall s. m s = n \wedge I s \rightarrow 1 \leq \text{wp do } G \rightarrow \text{body od } \lambda s. \text{True} \)

**Proof (clarify)**

- **Fix** \( s \)
- **Assume** \( m s = 0 \)
- **With guard** \( \text{have } \neg G s \text{ by (blast)} \)
- **With \( hh \)** \( \text{show } 1 \leq \text{wp do } G \rightarrow \text{body od } \lambda s. \text{True} \)
  - **By (simp add: wp-loop-nguard)**

**Qed**

**Assume** \( \forall s. m s = n \wedge I s \rightarrow 1 \leq \text{wp do } G \rightarrow \text{body od } \lambda s. \text{True} \)

**Show** \( \forall s. m s = \text{Suc } n \wedge I s \rightarrow 1 \leq \text{wp do } G \rightarrow \text{body od } \lambda s. \text{True} \)

**Proof (intro fold-premise healthy-intros \( hh \)). \( \text{le-funI} \)**

- **Fix** \( s \)
- **Show** \( \lambda s. m s = \text{Suc } n \wedge I s \rightarrow s \leq \text{wp do } G \rightarrow \text{body od } \lambda s. \text{True} \)

**Proof (cases \( G s \))**

**Case** False with \( hh \) \( \text{show } \text{thesis} \)
  - **By (simp add: wp-loop-nguard)**

**Next**

**Case** True note \( G = \text{this} \)

- **Have** \( \lambda s. m s = \text{Suc } n \wedge \text{hit} \)
  - **By (simp)**

**Also Have** \( \ldots = (\lambda s. m s = \text{Suc } n \wedge \text{hit}) \)
  - **By (simp add: exp-conj-assoc exp-conj-unitary del: exp-conj-idem)**

**Also Have** \( \ldots = (\lambda s. m s = \text{Suc } n \wedge \text{hit}) \)
  - **By (simp only: exp-conj-comm)**

**Also** \{ \( \text{from inv } hh \) \( \text{have } G \wedge \text{hit} \) \( \text{wp do } I \) \( \text{by (rule wp-inv-stdD)} \) \}
with variant

have (λs. m s = Suc n) & & «I» & & «G» & & «I») ⊨
    wp body (λs. m s = n) & & wp body «I»
    by (rule entails-frame)

} also from scb

have wp body (λs. m s = n) & & wp body «I» ⊨
    wp body (λs. m s = n) & & «I»
    by (blast)

finally have (λs. m s = Suc n) & & «I» & & «G» ⊨
    wp body (λs. m s = n) & & «I».

moreover {
    from IH have (λs. m s = n & & I s) ⊨ wp do G −→ body od «λs. True»
        by (blast intro: use-premise healthy-intros hh)
    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 hh 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 \(\lambda s. 1 \not\vdash wp do G \to body od (\lambda s. 1)\)

**proof**
- note hb = well-def-wp-healthy[OF wb]
- note sb = healthy-scalingD[OF hb]
- note sab = sublinear-subadd[OF well-def-wp-sublinear, OF wb, OF healthy-feasibleD, OF hb]

from hb have hloop: healthy (wp do G \to body od)
by(rule healthy-intros)
hence swp: sound (wp do G \to body od (\lambda s. 1)) by(blast)
p is no greater than 1, by feasibility.

from onestep have onestep': \(\forall s. p \leq wp body \langle N \ G \rangle s\) by(auto)
also {
  from hb have unitary (wp body \langle N \ G \rangle) by(auto)
  hence \(\forall s. wp body \langle N \ G \rangle s \leq 1\) 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: \(\forall k. 0 \leq k \implies k \leq 1 \implies (\lambda s. k) \not\vdash wp do G \to body od (\lambda s. 1)\) \implies
\(\lambda s. p * (1-k) + k \not\vdash wp do G \to body od (\lambda s. 1)\)

**proof**(rule le-funI)
fix k s
assume X: \(\lambda s. k \not\vdash wp do G \to body od (\lambda s. 1)\)
and k0: \(0 \leq k\) and k1: \(k \leq 1\)

from k1 have nz1k: \(0 \leq 1 - k\) by(auto)
with p1 have \(p * (1-k) + k \leq 1 * (1-k) + k\)
by(blast intro:mult-right-mono add-mono)
hence \(p * (1-k) + k \leq 1\)
by(simp)

The new bound is \(p * (1-k) + k\).

hence \(p * (1-k) + k \leq \langle N \ G \rangle s + \langle G \rangle s * (p * (1-k) + k)\)
by(cases G s, simp-all)

By the one-step termination assumption:
also from onestep' nz1k
have ... \(\leq \langle N \ G \rangle s + \langle G \rangle s * (wp body \langle N \ G \rangle s * (1-k) + k)\)
by (simp add: mult-right-mono ordered-comm-semiring-class.comm-mult-left-mono)

By scaling:
also from nz1k
Lastly, by folding two loop iterations:

\[
\text{have } \ldots = \langle \mathcal{N} \ G \rangle \ s + \langle G \rangle \ s \ast (wp \ \text{body} \ (\lambda s. \langle \mathcal{N} \ G \rangle \ s \ast (1-k)) \ s + k) \\
\quad \text{by} (\text{simp add:right-scaling mod[OF sh]})
\]

By the maximality (termination) of the loop body:

\[
\text{also from } \text{mb k0} \\
\text{have } \ldots = \langle \mathcal{N} \ G \rangle \ s + \langle G \rangle \ s \ast (wp \ \text{body} \ (\lambda s. \langle \mathcal{N} \ G \rangle \ s \ast (1-k)) \ s + wp \ \text{body} \\
(\lambda s. \ k) \ s) \\
\quad \text{by} (\text{simp add:maximalD})
\]

By sub-additivity of the loop body:

\[
\text{also from } k0 \text{nizk} \\
\text{have } \ldots = \langle \mathcal{N} \ G \rangle \ s + \langle G \rangle \ s \ast (wp \ \text{body} \ (\lambda s. \langle \mathcal{N} \ G \rangle \ s + \langle G \rangle \ s \ast k) \ s) \\
\quad \text{by} (\text{auto intro!}:\text{add-left-mono mult-left-mono sub-addD mod[OF sab] sound-intros})
\]

By monotonicity of the loop body, and that \( k \) is a lower bound:

\[
\text{also from } k0 \text{ le-funD mod[OF X]} \\
\text{have } \ldots \leq \langle \mathcal{N} \ G \rangle \ s + \langle G \rangle \ s \ast (wp \ \text{body} \ (\lambda s. \langle \mathcal{N} \ G \rangle \ s + \langle G \rangle \ s \ast wp \ \text{do} \ G \rightarrow \ \text{body od} \ (\lambda s. \ 1) \ s) \\
\text{s})
\quad \text{by} (\text{iprover intro!}:\text{add-left-mono mult-left-mono le-funI embed-gc-0} \\
\text{le-funD mod[OF mono-transD, mod[healthy-monoD, mod[hb}]
\text{ sound-sum standard-sound sound-intros swp})
\]

Unrolling the loop once and simplifying:

\[
\text{also } \{ \\
\text{have } \bigwedge s. \langle \mathcal{N} \ G \rangle \ s + \langle G \rangle \ s \ast wp \ \text{body} \ (wp \ \text{do} \ G \rightarrow \ \text{body od} \ (\lambda s. \ 1) \ s) = \\
\langle \mathcal{N} \ G \rangle \ s + \langle G \rangle \ s \ast (\lambda s. \ \langle \mathcal{N} \ G \rangle \ s + \langle G \rangle \ s \ast wp \ \text{body} \ (wp \ \text{do} \ G \rightarrow \ \text{body od} \ (\lambda s. \ 1) \ s) \\
(\lambda s. \ 1) \ s)
\quad \text{by} (\text{simp only!}:\text{distrib-left mul assoc mod[symmetric] embed-bool-idem embed-bool-cancel})
\]

\[
\text{also have } \bigwedge s. \ldots = \langle \mathcal{N} \ G \rangle \ s + \langle G \rangle \ s \ast wp \ \text{do} \ G \rightarrow \ \text{body od} \ (\lambda s. \ 1) \ s
\quad \text{by} (\text{simp add:fun-cong mod[wp-loop-unfold mod[symmetric, where \ P=\lambda s. \ 1, simplified, mod[hb]]])}
\]

\[
\text{finally have } X. \bigwedge s. \langle \mathcal{N} \ G \rangle \ s + \langle G \rangle \ s \ast wp \ \text{body} \ (wp \ \text{do} \ G \rightarrow \ \text{body od} \ (\lambda s. \ 1) \ s)
\quad = \\
\langle \mathcal{N} \ G \rangle \ s + \langle G \rangle \ s \ast wp \ \text{do} \ G \rightarrow \ \text{body od} \ (\lambda s. \ 1) \ s
\quad \text{by} (\text{simp only:X})
\]

Lastly, by folding two loop iterations:

\[
\text{also } \\
\text{have } \langle \mathcal{N} \ G \rangle \ s + \langle G \rangle \ s \ast (wp \ \text{body} \ (\lambda s. \langle \mathcal{N} \ G \rangle \ s + \langle G \rangle \ s \ast \\
wp \ \text{body} \ (wp \ \text{do} \ G \rightarrow \ \text{body od} \ (\lambda s. \ 1) \ s) \ s) =
\]

\[
\langle \mathcal{N} \ G \rangle \ s + \langle G \rangle \ s \ast (wp \ \text{body} \ (wp \ \text{do} \ G \rightarrow \ \text{body od} \ (\lambda s. \ 1) \ s) \ s)
\]

\[
\quad \text{by}(\text{simp only:X})
\]
\[ wp \text{ do } G \rightarrow \text{ body od } (\lambda s. 1) \ s \]

\text{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]])}

\text{finally show } p \ast (1-k) + k \leq wp \text{ do } G \rightarrow \text{ body od } (\lambda s. 1) \ s .

\text{qed}

If the previous bound lay in \([0,1)\), the new bound is strictly greater. This is where we appeal to the fact that \(p\) is nonzero.

\text{from nzp have inc: } \exists k. 0 \leq k \Rightarrow k < 1 \Rightarrow k < p \ast (1-k) + k

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

The result follows by contradiction.

\text{show } \text{thesis}

\text{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.

\text{assume } \neg \text{thesis}

\text{hence } \neg (\forall s. 1 \leq wp \text{ do } G \rightarrow \text{ body od } (\lambda s. 1) \ s) \text{ by (auto)}

\text{then obtain } s \text{ where point: } \neg 1 \leq wp \text{ do } G \rightarrow \text{ body od } (\lambda s. 1) \ s \text{ by (auto)}

\text{let } ?k = \text{Inf } (\text{range } (wp \text{ do } G \rightarrow \text{ body od } (\lambda s. 1)))

\text{from bloop}

\text{have Inflb: } \exists s. ?k \leq wp \text{ do } G \rightarrow \text{ body od } (\lambda s. 1) \ s

\text{by (intro cInf-lower bdd-belowI, auto)}

\text{also from point have wp do } G \rightarrow \text{ body od } (\lambda s. 1) \ s < 1 \text{ by (auto)}

Thus the least (infimum) probability of termination is strictly less than one.

\text{finally have k1: } ?k < 1 .

\text{hence } ?k \leq 1 \text{ by (auto)}

\text{moreover from bloop have k0: } 0 \leq ?k

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

The infimum is, naturally, a lower bound.

\text{moreover from Inflb have } (\lambda s. ?k) \vdash wp \text{ do } G \rightarrow \text{ body od } (\lambda s. 1) \text{ by (auto)}

\text{ultimately}

We can therefore use the previous result to find a new bound, . . .

\text{have } \exists s. p \ast (1 - ?k) + ?k \leq wp \text{ do } G \rightarrow \text{ body od } (\lambda s. 1) \ s

\text{by (blast intro: le-funD[OF new-bound])}

. . . which is lower than the infimum, by minimality, . . .

\text{hence } p \ast (1 - ?k) + ?k \leq ?k

\text{by (blast intro: cInf-greatest)}

. . . yet also strictly greater than it.
moreover from \( k0 \leq k1 \) have \( \exists k \leq p \ast (1 - \exists k) + \exists k \) by (rule inc)

We thus have a contradiction.

ultimately show False by (simp)

qed

end

4.12 Automated Reasoning

theory Automation imports StructuredReasoning
begin

This theory serves as a container for automated reasoning tactics for pGCL, implemented in ML. At present, there is a basic verification condition generator (VCG).

named-theorems wd
  theorems to automatically establish well-definedness

named-theorems pwp-core
  core probabilistic wp rules, for evaluating primitive terms

named-theorems pwp
  user-supplied probabilistic wp rules

named-theorems pwp
  user-supplied probabilistic wlp rules

ML-file (pVCG.ML)

method-setup pvcg =
  (Scan.succeed (fn ctxt => SIMPLE-METHOD' (pVCG.pVCG-tac ctxt)));

declare wd-intros[wd]

lemmas core-wp-rules =
  wp-Skip \equiv wlp-Skip
  wp-Abort \equiv wlp-Abort
  wp-Apply \equiv wlp-Apply
  wp-Seq \equiv wlp-Seq
  wp-DC-split \equiv wlp-DC-split
  wp-PC-fixed \equiv wlp-PC-fixed
  wp-SetDC \equiv wlp-SetDC
  wp-SetPC-split \equiv wlp-SetPC-split

declare core-wp-rules[pwp-core]

end
Additional Material

4.13 Miscellaneous Mathematics

theory Misc
imports
  HOL−Analysis.Multivariate-Analysis
begin

lemma sum-UNIV:
  fixes S::′a::finite set
  assumes complete: ∀x. x∉S ⇒ f x = 0
  shows sum f S = sum f UNIV
proof
  from complete have sum f S = sum f (UNIV - S) + sum f S by(simp)
  also have ... = sum f UNIV
    by(auto intro: sum.subset-diff[symmetric])
finally show ?thesis .
qed

lemma cInf-mono:
  fixes A::′a::conditionally-complete-lattice set
  assumes lower: ∀b. b ∈ B ⇒ ∃a∈A. a ≤ b
    and bounded: ∀a. a ∈ A ⇒ c ≤ a
    and ne: B ≠ {}
  shows Inf A ≤ Inf B
proof(rule cInf-greatest[OF ne])
  fix b assume bin: b ∈ B
  with lower obtain a where ain: a ∈ A and le: a ≤ b by(auto)
  from ain bounded have Inf A ≤ a by(intro cInf-lower bdd-belowI , auto)
  also note le
  finally show Inf A ≤ b .
qed

lemma max-distrib:
  fixes c::real
  assumes nn: 0 ≤ c
  shows c * max a b = max (c * a) (c * b)
proof(cases a ≤ b)
  case True
  moreover with nn have c * a ≤ c * b by(auto intro:mult-left-mono)
  ultimately show ?thesis by(simp add:max.absorb2)
next 
case False then have $b \leq a$ by(auto) 
moreover with nn have $c \cdot b \leq c \cdot a$ by(auto intro:mult-left-mono) 
ultimately show ?thesis by(simp add:max.absorb1)
qed 

lemma mult-div-mono-left: 
fixes c::real 
assumes nnc: $0 \leq c$ and nzcnznz: $c \neq 0$ 
and inv: $a \leq \text{inverse} \ c \cdot b$
shows $c \cdot a \leq b$
proof -- 
from nnc inv have $c \cdot a \leq (c \cdot \text{inverse} \ c) \cdot b$
by(auto simp:mult.assoc intro:mult-left-mono) 
also from nzcnznz have ... = 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 nzcnznz: $c \neq 0$ 
and inv: $\text{inverse} \ c \cdot a \leq b$
shows $a \leq c \cdot b$
proof -- 
from nzcnznz have $a = (c \cdot \text{inverse} \ c) \cdot a$ by(simp) 
also from nnc inv have $(c \cdot \text{inverse} \ c) \cdot a \leq c \cdot b$
by(auto simp:mult.assoc intro:mult-left-mono) 
finally show $a \leq c \cdot b$.
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(rule finite-induct, simp-all add:assms)
  fix $x :: 'a$ and $S :: 'a set$
  assume $IH: S = \{\} \lor (\exists x \in S. \forall y \in S. x \leq y)$
  show $(\forall y \in S. x \leq y) \lor (\exists x' \in S. x' \leq x \land (\forall y \in S. x' \leq y))$
  proof(cases $S = \{\}$)
    case True then show $\text{thesis}$ by(auto)
  next
    case False with $IH$ have $\exists x \in S. \forall y \in S. x \leq y$ by(auto)
    then obtain $z$ where $zin: z \in S$ and $zmin: \forall y \in S. z \leq y$ by(auto)
    thus $\text{thesis}$ by(cases $z \leq x$, auto)
qed

with $\text{ne}$ show $\text{thesis}$ by(auto)
qed

lemma $cSup$-add:
  fixes $c :: \text{real}$
  assumes $\text{ne: } S \neq \{\}$
  and $bS: \forall x. x \in S \implies x \leq b$
  shows $\sup S + c = \sup \{ x + c | x \in S \}$
proof(rule antisym)
  from $\text{ne bS}$ show $\sup \{ x + c | x \in S \} \leq \sup S + c$
    by(auto intro:!:$cSup$-least add-right-mono $cSup$-upper bdd-aboveI)$
  have $\sup S \leq \sup \{ x + c | x \in S \} - c$
  proof(intro $cSup$-least $\text{ne}$)
    fix $x$ assume $zin: x \in S$
    from $bS$ have $\forall x. x \in S \implies x + c \leq b + c$ by(auto intro:add-right-mono)
    hence $\text{bdd-aboveI} \{ x + c | x \in S \}$ by(intro bdd-aboveI, blast)
    with $zin$ have $x + c \leq \sup \{ x + c | x \in S \}$ by(auto intro:$cSup$-upper)
    thus $x \leq \sup \{ x + c | x \in S \} - c$ by(auto)
  qed
  thus $\sup S + c \leq \sup \{ x + c | x \in S \}$ by(auto)
qed

lemma $cSup$-mult:
  fixes $c :: \text{real}$
  assumes $\text{ne: } S \neq \{\}$
  and $bS: \forall x. x \in S \implies x \leq b$
  and $\text{nnc: } 0 \leq c$
  shows $c \ast \sup S = \sup \{ c \ast x | x \in S \}$
proof(cases)
  assume $c = 0$
  moreover from $\text{ne}$ have $\exists x. x \in S$ by(auto)
  ultimately show $\text{thesis}$ by(simp)
next
assume cnz: \( c \neq 0 \)

show ?thesis

proof (rule antisym)
  from \( bS \) have \( baS: \text{bdd-above } S \) 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 \[ \text{OF cSup-upper} \])
  have \( \sup S \leq \inverse c \ast \sup \{ c \ast x \mid x \in S \} \)
  proof (intro cSup-least ne)
    fix \( x \) assume \( x : x \in S \)
    moreover from \( bS \) nnc have \( \forall x . x \in S \Rightarrow c \ast x \leq c \ast b \)
      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

qed

lemma closure-contains-Sup:
  fixes \( S : \text{real set} \)
  assumes neS: \( S \neq \{ \} \) and \( bS: \forall x \in S . x \leq B \)
  shows \( \sup S \in \text{closure } S \)
  proof (subgoal_tac)
    from neS have \( \forall x \in \uminus ' S \) \( \inf \uminus ' S \leq \uminus ' x \)
      by (auto)
    from bS have \( \forall x \in \uminus ' S . \exists y \in S . x \leq y \)
      by (auto)
    hence \( \forall x \in S . x \leq \inf \uminus ' S \)
      by (intro mult-left mono)
    with cnz show \( \forall x \in S . x \leq \inf \uminus ' S \)
      by (simp)
  qed

proof (rule antisym)
  from neT bT have \( \forall x \in \uminus ' S \) \( \inf \uminus ' S \leq \uminus ' x \)
    by (auto)
  hence \( \forall x \in \uminus ' S . \exists y \in S . x \leq y \)
    by (intro mult-left mono)
  hence \( \forall x \in S . x \leq \inf \uminus ' S \)
    by (simp)
  with neS bT show \( \sup S \leq \inf \uminus ' S \)
    by (intro cSup-least)
  have \( \sup S \leq \inf \uminus ' S \)

proof (rule cInf-greatest (OF neT))
fix x assume x ∈ uminus ' S
then obtain y where yin: y ∈ S and rwx: x = − y by (auto)
from yin bS have y ≤ Sup S
  by (intro cSup-upper bdd-belowI, auto)
hence − 1 * Sup S ≤ − 1 * y
  by (simp add: mult-left-mono-neg)
with rwx show − Sup S ≤ x by (simp)
qed

hence − 1 * Inf ?T ≤ − 1 * (− Sup S)
  by (simp add: mult-left-mono-neg)
thus − Inf ?T ≤ Sup S by (simp)
qed

also {
  from neT bbT have Inf ?T ∈ closure ?T by (rule closure-contains-Inf)
hence − Inf ?T ∈ uminus ' closure ?T by (auto)
}
also {
  have linear uminus by (auto intro: linearI)
hence uminus ' closure ?T ⊆ closure (uminus ' ?T)
  by (rule closure-linear-image-subset)
}
also {
  have uminus ' ?T ⊆ S by (auto)
hence closure (uminus ' ?T) ⊆ closure S by (rule closure-mono)
}
finally show Sup S ∈ closure S .
qed

lemma tendsto-min:
  fixes x y::real
  assumes ta: a −→ x
      and tb: b −→ y
  shows (λi. min (a i) (b i)) −→ min x y
proof (rule LIMSEQ-I, simp)
  fix e::real assume pe: 0 < e

  from ta pe obtain noa where balla: ∀ n ≥ noa. abs (a n − x) < e
    by (auto dest: LIMSEQ-D)
  from tb pe obtain nob where ballb: ∀ n ≥ nob. abs (b n − y) < e
    by (auto dest: LIMSEQ-D)

  { 
    fix n
    assume ge: max noa nob ≤ n
    hence gea: noa ≤ n and geb: nob ≤ n by (auto)
    have abs (min (a n) (b n) − min x y) < e
      proof cases
        assume le: min (a n) (b n) ≤ min x y
show ?thesis
proof cases
  assume a n ≤ b n
  hence \( \text{rwmin: } \min (a n) (b n) = a n \) by(auto)
  with \( \leq \) have a n ≤ \( \min x y \) by(simp)
  moreover from gea balla have abs (a n - x) < e by(simp)
  moreover have \( \min x y \leq x \) by(auto)
  ultimately have abs (a n - \( \min x y \)) < e by(auto)
  with \( \text{rwmin} \) show abs (\( \min (a n) (b n) - \min x y \)) < e by(simp)
next
  assume \( \neg a n \leq b n \)
  hence \( \neg \text{a n} \leq a n \) by(auto)
  hence \( \text{rwmin: } \min (a n) (b n) = b n \) by(auto)
  with \( \leq \) have b n ≤ \( \min x y \) by(simp)
  moreover from geb ballb have abs (b n - y) < e by(simp)
  moreover have \( \min x y \leq y \) by(auto)
  ultimately have abs (b n - \( \min x y \)) < e by(auto)
  with \( \text{rwmin} \) show 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 \( \leq \) \( \min x y \leq \min (a n) (b n) \) by(auto)
  show ?thesis
proof cases
  assume x ≤ y
  hence \( \text{rwmin: } \min x y = x \) by(auto)
  with \( \leq \) have x ≤ \( \min (a n) (b n) \) by(simp)
  moreover from gea balla have abs (a n - x) < e by(simp)
  moreover have \( \min (a n) (b n) \leq a n \) by(auto)
  ultimately have abs (\( \min (a n) (b n) - x \)) < e by(auto)
  with \( \text{rwmin} \) show abs (\( \min (a n) (b n) - \min x y \)) < e by(simp)
next
  assume \( \neg x \leq y \)
  hence \( \text{rwmin: } \min x y = y \) by(auto)
  with \( \leq \) have y ≤ \( \min (a n) (b n) \) by(simp)
  moreover from geb ballb have abs (b n - y) < e by(simp)
  moreover have \( \min (a n) (b n) \leq b n \) by(auto)
  ultimately have abs (\( \min (a n) (b n) - y \)) < e by(auto)
  with \( \text{rwmin} \) show abs (\( \min (a n) (b n) - \min x y \)) < e by(simp)
qed
qed

thus \( \exists \text{no. } \forall n \geq \text{no. } |\min (a n) (b n) - \min x y| < e \) by(blast)
qed

definition supp :: \( 's \Rightarrow \text{real} \Rightarrow 's \ set \)
where supp f = \{x. f x ≠ 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
lemma tminus-left-mono: 
\[ a \leq b \implies a \ominus c \leq b \ominus c \]
unfolding tminus-def 
by (case-tac a \leq c, simp-all)

lemma tminus-less: 
\[ \[ 0 \leq a; 0 \leq b \] \implies a \ominus b \leq a \]
unfolding tminus-def by (force)

lemma tminus-left-distrib: 
assumes nna: \( 0 \leq a \)
shows \( a \cdot (b \ominus c) = a \cdot b \ominus a \cdot c \)
proof (cases b \leq c)
\begin{cases}
\text{case True} & \text{note le = this} \\
\text{hence } a \cdot \max (b - c) \cdot 0 = 0 & \text{by (simp add: max.absorb2)} \\
\text{also } & \\
\text{from nna le have } a \cdot b \leq a \cdot c & \text{by (blast intro: mult-left-mono)} \\
\text{hence } 0 = \max (a \cdot b - a \cdot c) \cdot 0 & \text{by (simp add: max.absorb1)} \\
\end{cases}
\text{finally show } ?thesis \text{ by (simp add: tminus-def)}
\text{next}
\begin{cases}
\text{case False} & \text{hence le: } c \leq b \text{ by (auto)} \\
\text{hence } a \cdot \max (b - c) \cdot 0 = a \cdot (b - c) & \text{by (simp only: max.absorb1)} \\
\text{also } & \\
\text{from nna le have } a \cdot c \leq a \cdot b & \text{by (blast intro: mult-left-mono)} \\
\text{hence } a \cdot (b - c) = \max (a \cdot b - a \cdot c) \cdot 0 & \text{by (simp add: max.absorb1)} \\
\end{cases}
\text{field-simps}
\text{finally show } ?thesis \text{ by (simp add: tminus-def)}
qed

lemma tminus-le[simp]: 
\[ b \leq a \implies a \ominus b = a - b \]
unfolding tminus-def by (simp)

lemma tminus-le-alt[simp]: 
\[ a \leq b \implies a \ominus b = 0 \]
by (simp add: tminus-def)

lemma tminus-nle[simp]: 
\[ \neg b \leq a \implies a \ominus b = 0 \]
unfolding tminus-def by (simp)

lemma tminus-add-mono: 
\[ (a+b) \ominus (c+d) \leq (a\ominus c) + (b\ominus d) \]
proof (cases \( 0 \leq a - c \))
\begin{cases}
\text{case True} & \text{note pac = this} \\
\end{cases}
show \( \text{thesis} \)
proof(cases \( \theta \leq b - d \))
  case True note pbd = this
  from pac and pbd have \((c + d) \leq (a + b)\) by(simp)
  with pac and pbd show \( \text{thesis} \) by(simp)
next
  case False with pac show \( \text{thesis} \)
  by(cases \( c + d \leq a + b, \text{auto} \))
qed

next
  case False note nac = this
  show \( \text{thesis} \)
  proof(cases \( 0 \leq b - d \))
    case True with nac show \( \text{thesis} \)
    by(cases \( c + d \leq a + b, \text{auto} \))
  next
    case False note nbd = this
    with nac have \( \neg (c + d) \leq (a + b) \) by(simp)
    with nac and nbd show \( \text{thesis} \) by(simp)
  qed

qed

lemma tminus-sum-mono:
assumes fS: finite S
shows \( \text{sum} f \ S \ominus \text{sum} g \ S \leq \text{sum} (\lambda x. f x \ominus g x) \ S \)
(is \( ?X \ S \))
proof(rule finite-induct)
  from fS show finite S .

  show \( ?X \ \{\} \) by(simp)

  fix \( x \) and \( F \)
  assume fF: finite F and xniF: \( x \notin F \)
  and IH: \( ?X \ F \)
  have \( f x + \text{sum} f \ F \ominus \text{sum} g \ F \leq \)
  \( (f x \ominus g x) + (\text{sum} f F \ominus \text{sum} f F) \)
  by(rule tminus-add-mono)
  also from IH have \( \ldots \leq (f x \ominus g x) + (\sum_{x \in F. f x \ominus g x}) \)
  by(rule add-left-mono)
  finally show \( ?X \ (\text{insert} x \ F) \)
  by(simp add:sum.insert[OF fF xniF])
qed

lemma tminus-nneg[simp,intro]:
\( \theta \leq a \ominus b \)
by(cases \( b \leq a, \text{auto} \))

lemma tminus-right-antimono:
assumes clb: \( c \leq b \)
shows $a \odot b \leq a \odot c$

proof (cases $b \leq a$)
  case True
    moreover with $\mathit{clb}$ have $c \leq a$ by (auto)
    moreover note $\mathit{clb}$
    ultimately show ?thesis by (simp)
  next
  case False then show ?thesis by (simp)
qed

lemma min-tminus-distrib:
  $\min a \ b \odot c = \min (a \odot c) \ (b \odot c)$
  unfolding tminus-def by (auto)

end


