pGCL for Isabelle

David Cock

September 19, 2015
Contents

1 Overview 1

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

3 Semantic Structures 17
   3.1 Expectations 17
      3.1.1 Bounded Functions 18
      3.1.2 Non-Negative Functions 20
      3.1.3 Sound Expectations 21
      3.1.4 Unitary expectations 25
      3.1.5 Standard Expectations 25
      3.1.6 Entailment 27
      3.1.7 Expectation Conjunction 28
      3.1.8 Rules Involving Conjunction 29
      3.1.9 Rules Involving Entailment and Conjunction Together 31
   3.2 Expectation Transformers 33
      3.2.1 Comparing Transformers 36
      3.2.2 Healthy Transformers 39
      3.2.3 Sublinearity 47
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.4</td>
<td>Determinism</td>
<td>57</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Modular Reasoning</td>
<td>60</td>
</tr>
<tr>
<td>3.2.6</td>
<td>Transforming Standard Expectations</td>
<td>61</td>
</tr>
<tr>
<td>3.3</td>
<td>Induction</td>
<td>62</td>
</tr>
<tr>
<td>3.3.1</td>
<td>The Lattice of Expectations</td>
<td>62</td>
</tr>
<tr>
<td>3.3.2</td>
<td>The Lattice of Transformers</td>
<td>66</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Tail Recursion</td>
<td>72</td>
</tr>
<tr>
<td>4</td>
<td>The pGCL Language</td>
<td>75</td>
</tr>
<tr>
<td>4.1</td>
<td>A Shallow Embedding of pGCL in HOL</td>
<td>75</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Core Primitives and Syntax</td>
<td>75</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Unfolding rules for non-recursive primitives</td>
<td>79</td>
</tr>
<tr>
<td>4.2</td>
<td>Healthiness</td>
<td>81</td>
</tr>
<tr>
<td>4.2.1</td>
<td>The Healthiness of the Embedding</td>
<td>81</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Healthiness for Loops</td>
<td>94</td>
</tr>
<tr>
<td>4.3</td>
<td>Continuity</td>
<td>104</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Continuity of Primitives</td>
<td>105</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Continuity of a Single Loop Step</td>
<td>122</td>
</tr>
<tr>
<td>4.4</td>
<td>Continuity and Induction for Loops</td>
<td>126</td>
</tr>
<tr>
<td>4.4.1</td>
<td>The Limit of Iterates</td>
<td>134</td>
</tr>
<tr>
<td>4.5</td>
<td>Sublinearity</td>
<td>144</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Nonrecursive Primitives</td>
<td>144</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Sublinearity for Loops</td>
<td>150</td>
</tr>
<tr>
<td>4.6</td>
<td>Determinism</td>
<td>154</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Additivity</td>
<td>154</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Maximality</td>
<td>155</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Determinism</td>
<td>157</td>
</tr>
<tr>
<td>4.7</td>
<td>Well-Defined Programs</td>
<td>158</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Strict Implies Liberal</td>
<td>158</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Sub-Distributivity of Conjunction</td>
<td>162</td>
</tr>
<tr>
<td>4.7.3</td>
<td>The Well-Defined Predicate</td>
<td>172</td>
</tr>
<tr>
<td>4.8</td>
<td>The Loop Rules</td>
<td>175</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Liberal and Strict Invariants</td>
<td>175</td>
</tr>
<tr>
<td>4.8.2</td>
<td>Partial Correctness</td>
<td>176</td>
</tr>
<tr>
<td>4.8.3</td>
<td>Total Correctness</td>
<td>176</td>
</tr>
<tr>
<td>4.8.4</td>
<td>Unfolding</td>
<td>177</td>
</tr>
<tr>
<td>4.9</td>
<td>The Algebra of pGCL</td>
<td>178</td>
</tr>
<tr>
<td>4.9.1</td>
<td>Program Refinement</td>
<td>178</td>
</tr>
<tr>
<td>4.9.2</td>
<td>Simple Identities</td>
<td>179</td>
</tr>
<tr>
<td>4.9.3</td>
<td>Deterministic Programs are Maximal</td>
<td>184</td>
</tr>
<tr>
<td>4.9.4</td>
<td>The Algebraic Structure of Refinement</td>
<td>186</td>
</tr>
<tr>
<td>4.9.5</td>
<td>Data Refinement</td>
<td>189</td>
</tr>
<tr>
<td>4.9.6</td>
<td>The Algebra of Data Refinement</td>
<td>193</td>
</tr>
<tr>
<td>4.9.7</td>
<td>Structural Rules for Correspondence</td>
<td>200</td>
</tr>
</tbody>
</table>
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.9.8 Structural Rules for Data Refinement</td>
<td>201</td>
</tr>
<tr>
<td>4.10 Structured Reasoning</td>
<td>203</td>
</tr>
<tr>
<td>4.10.1 Syntactic Decomposition</td>
<td>204</td>
</tr>
<tr>
<td>4.10.2 Algebraic Decomposition</td>
<td>209</td>
</tr>
<tr>
<td>4.10.3 Hoare triples</td>
<td>209</td>
</tr>
<tr>
<td>4.11 Loop Termination</td>
<td>209</td>
</tr>
<tr>
<td>4.11.1 Trivial Termination</td>
<td>210</td>
</tr>
<tr>
<td>4.11.2 Classical Termination</td>
<td>210</td>
</tr>
<tr>
<td>4.11.3 Probabilistic Termination</td>
<td>213</td>
</tr>
<tr>
<td>4.12 Automated Reasoning</td>
<td>217</td>
</tr>
<tr>
<td>4.13 Miscellaneous Mathematics</td>
<td>219</td>
</tr>
<tr>
<td>4.13.1 Truncated Subtraction</td>
<td>219</td>
</tr>
<tr>
<td></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:

\[
\text{datatype} \quad \text{coin} = \text{Heads} | \text{Tails}
\]

\[
\text{record} \quad \text{coins} =
    \begin{array}{l}
    a :: \text{coin} \\
    b :: \text{coin}
    \end{array}
\]

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

\[
\text{lemma}
\]

\[
\text{Apply} \ (\lambda s. s (\mid a := \text{Heads} \mid)) = (a := (\lambda s. \text{Heads}))
\]

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

2.1.2 Assertion and Annotation

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

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

```
lemma \lbrack \lbrack \lceil \lambda s. b = x \rceil \vdash \text{wp} (a := b) \lbrack \lbrack \lceil \lambda s. a = x \rceil \rceil
by (pvcg, simp add: o-def)
```

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

2.1.3 Probability

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

```
lemma «\lambda s::real. s \neq 0 » \vdash \text{wp} (\text{Apply} (\text{op} * 2) ;; \text{Apply} (\lambda s. s / s)) \lbrack \lbrack \lceil \lambda s. s = 1 \rceil \rceil
by (pvcg, simp add: o-def)
```

So far, we haven’t done anything that required probabilities, or expectations other than 0 and 1. As an example of both, we show that a single coin toss is fair. We introduce the syntax $x p \oplus y$ for a probabilistic choice between $x$ and $y$. This program behaves as $x$ with probability $p$, and as $y$ with probability $(1::'a) - p$. 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**
flip-a :: real ⇒ coins prog

**Where**
flip-a p = a := (λ-. Heads) (λs. p) ⊕ a := (λ-. Tails)

**Lemma**
(λs. 1/2) = wp (flip-a (1/2)) «λs. a = Heads»

**Unfolding** flip-a-def

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

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

### 2.1.4 Nondeterminism

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

**Lemma**
λs. 1/3 ⊢ wp (flip-a (2/3) ⊔ flip-a (1/3)) «λs. a = Heads»

**Unfolding** flip-a-def

by (pvcg, simp add: o-def le-funI)

### 2.1.5 Properties of Expectations

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

**Definition**
flip-b :: real ⇒ coins prog

**Where**
flip-b p = b := (λ-. Heads) (λs. p) ⊕ b := (λ-. Tails)

**Lemma**
(λs. 1/4) = wp (flip-a (1/2) ⊔ flip-b (1/2)) «λs. a = Heads ∧ b = Heads»

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

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

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

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

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

lemma Puniform-out:
  x /∈ S ⇒ Puniform S x = 0
by(simp add:Puniform-def)

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

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

lemma
(λs. 7/2) = wp (bind v at (λs. Puniform {1..6} v) in red := (λs. v)) red
by(simp add:wp-eval supp-Puniform setsum-head-Suc Puniform-in real-eq-of-nat)

The expectations of independent variables add:

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

2.2 Loops

theory LoopExamples imports ../pGCL begin

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

2.2.1 Guaranteed Termination

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

```
definition countdown :: int prog
  where
  countdown = do (\x. 0 < x) → Apply (\s. 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 inv-count :: int ⇒ bool
  where
  inv-count = (\x. 0 \leq x)
```

Read \(wp-inv G \text{ body } I\) as: \(I\) is an invariant of the loop \(\mu x. \text{ body } :: x « G » \oplus \text{ Skip}, or « G » & & I \vdash wp \text{ body } I\).

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

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

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

As usual, well-definedness is trivial.

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

A measure of 0 implies termination.

```
  show \forall x. nat (max x 0) = 0 → \neg 0 < x
    by (auto)
```

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

This is the kind of proof that the VCG is good at. It leaves a purely logical goal,
which we can solve with auto.
CHAPTER 2. INTRODUCTION TO PGCL

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

qed

2.2.2 Probabilistic Termination

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

type-synonym \( \text{coin} = \text{bool} \)
definition \( \text{Heads} = \text{True} \)
definition \( \text{Tails} = \text{False} \)

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

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

definition \( \text{wait-for-heads} :: \text{coin prog} \)
where
\( \text{wait-for-heads} = \text{do} (\text{op } \neq \text{Heads}) \rightarrow \text{flip} \text{ od} \)

Nonetheless, we can show termination.

lemma \( \text{wait-for-heads-term} : \)
\( \lambda s. 1 \mapsto \wp \text{ wait-for-heads} (\lambda s. 1) \)
unfolding \( \text{wait-for-heads-def} \)

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 \((\text{rule termination-0-1}) \)
show \( \text{well-def flip} \)
unfolding \( \text{flip-def} \)
by \((\text{auto intro: wd-intros}) \)

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

show \( \text{maximal} (\wp \text{ flip}) \)
unfolding \( \text{flip-def} \) by \((\text{auto intro:max-intros}) \)

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

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
CHAPTER 2. INTRODUCTION TO PGCL

Invariants

We prove explicitly that only valid doors are ever chosen.

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

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

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

2.3.2 The Game

Hide the prize behind door $D$.

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

Choose door $D$.

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

Open door $D$ and reveal what’s behind.

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

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

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

Guess uniformly at random.

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

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

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

Switch your guess to the other unopened door.

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

The complete game, either with or without switching guesses.

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

where
  monty switch ≡ hide-prize ;;
         make-guess ;;
         reveal ;;
         (if switch then switch-guess else Skip)

2.3.3 A Brute Force Solution

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

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

lemma eval-loss[simp]:
  p ≠ g ⇒ «player-wins» (s\prize := p, \guess := g, \clue := c \}) = 0
  by(simp add:embed-bool-def player-wins-def)

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

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

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

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

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

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.
CHAPTER 2. INTRODUCTION TO PGCL

separately. While this is not strictly necessary for this program, it will scale to larger examples, as the work in annotation only increases linearly with the length of the program.

Healthiness

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

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

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

lemma wd-reveal:
  well-def reveal
proof

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

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

lemma wd-switch-guess:
  well-def switch-guess
proof

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

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

Annotations

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

hide-prize chooses a valid door.

**Lemma wp-hide-prize:**

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

**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:**

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

**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:**

\[«\lambda g. \text{guess } g \neq \text{prize } g \land \text{inv-prize } g \land \text{inv-guess } g» \vdash \text{wp reveal «\lambda g. guess } g \neq \text{prize } g \land \text{clue } g \neq \text{prize } g \land \text{clue } g \neq \text{guess } g \land \text{inv-prize } g \land \text{inv-guess } g \land \text{inv-clue } g»\]

(is ?X \vdash wp reveal ?Y)

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

**Fix** s

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

and inv-prize s

and inv-guess s

**Moreover then obtain** c

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

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 clarsimp ex1E)

**Ultimately show** \(1 \leq wp \text{ reveal } ?Y\ s\)

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

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

**Qed**

Showing that the three doors are all district is a largeish first-order problem, for which sledgehammer gives us a reasonable script.
lemma distinct-game:
[ guess g ≠ prize g; clue g ≠ prize g; clue g ≠ guess g;
  inv-prize g; inv-guess g; inv-clue g ] ⇒
{ 1, 2, 3 } = { guess g, prize g, clue g }

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

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

lemma wp-switch-guess:
« λg. guess g ≠ prize g ∧ clue g ≠ prize g ∧ clue g ≠ guess g ∧
  inv-prize g ∧ inv-guess g ∧ inv-clue g » ⊢ 
wp switch-guess « player-wins »

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

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

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

lemma wp-monty-switch-modular:
(λs. 2 / 3) ⊢ wp (monty True) « player-wins »

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

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

Using the VCG

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

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

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

end
Chapter 3

Semantic Structures

3.1 Expectations

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

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

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a → b</th>
<th>x</th>
<th>y</th>
<th>x ≤ y</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>0</td>
<td>0</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>0</td>
<td>1</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>1</td>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>1</td>
<td>1</td>
<td>T</td>
</tr>
</tbody>
</table>

For probabilistic automata, an expectation gives the current expected value of some expression, if it were to be evaluated in the final state. For example, consider the automaton of Figure 3.1, with transition probabilities affixed to edges. Let $P b = 2.0$ and $P c = 3.0$. Both states b and c are final (accepting) states, and thus the ‘final expected value’ of $P$ in state b is 2.0 and in state

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

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

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

### 3.1.1 Bounded Functions

definition bounded-by :: real $\Rightarrow$ ('a $\Rightarrow$ real) $\Rightarrow$ bool
where
 bounded-by b P $\equiv$ $\forall x. P \; x \leq b$

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

lemma bounded-byI[intro]:
 $[ \forall x. P \; x \leq b ] \Rightarrow$ bounded-by b P
by (simp add:bounded-by-def)

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

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

lemma bounded-byD2[dest]:
 bounded-by b P $\Rightarrow$ $P \leq (\lambda s. b)$
by (blast intro:le-funI)

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

definition bounded :: ('a $\Rightarrow$ real) $\Rightarrow$ bool
where
 bounded P $\equiv$ ($\exists b. \text{ bounded-by } b \; P$)

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

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

lemma bounded-bdd-above[intro]:
 assumes bP: bounded P
 shows bdd-above (range P)
proof
 fix x assume x $\in$ range P
with $b \cdot P$ show $x \leq \inf \{ b, \text{bounded-by } b \cdot P \}$

unfolding bounded-def by (auto intro: cInf-greatest)

qed

The least upper bound has the usual properties:

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

assumes $b \cdot P$: bounded-by $b \cdot P$

shows $\text{bound-of } P \leq b$

unfolding bound-of-def

using $b \cdot P$ by (intro cSup-least, auto)

**lemma** bounded-by-bound-of[!intro!]:

fixes $P$: ‘a ⇒ real

assumes $b \cdot P$: bounded $P$

shows $\text{bounded-by } (\text{bound-of } P) \cdot P$

unfolding bound-of-def

using $b \cdot P$ 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 $a \cdot P$; $a \leq b$ ] $\implies$ bounded-by $b \cdot P$

unfolding bounded-def by (blast intro: order-trans)

**lemma** bounded-by-imp-bounded[intro]:

bounded-by $b \cdot P \implies \text{bounded } P$

unfolding bounded-def by (blast)

This is occasionally easier to apply:

**lemma** bounded-by-bound-of-alt:

[ bounded $P$; bound-of $P = a$ ] $\implies$ bounded-by $a \cdot P$

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 $b \cdot Q$; $P \leq Q$ ] $\implies$ bounded-by $b \cdot P$

by (blast intro: order-trans dest: le-funD)

**lemma** bound-of-const[simp, intro]:

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

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

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

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

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

### 3.1.2 Non-Negative Functions.

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

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

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

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

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

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

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

lemma nneg-const[iff]:
  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 soundess, 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 assms by(auto)

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

**lemma sound-sum(simp,intro):**
assumes $sP$: sound $P$ and $sQ$: sound $Q$
shows sound $(\lambda s. P s + Q s)$

**proof**
from $sP$ have $\forall s. P s \leq \text{bound-of } P$ by(blast)
moreover from $sQ$ have $\forall s. Q s \leq \text{bound-of } Q$ by(blast)
ultimately have $\forall s. P s + Q s \leq \text{bound-of } P + \text{bound-of } Q$
by(rule add-mono)
thus bounded-by $(\text{bound-of } P + \text{bound-of } Q) (\lambda s. P s + Q s)$
by(blast)

from $sP$ and $sQ$ have $\forall s. 0 \leq P s$ by(blast)
moreover from $sQ$ have $\forall s. 0 \leq Q s$ by(blast)
ultimately have $\forall s. 0 \leq P s + Q s$ by(simp add: add-mono)
by(blast)

qed

**lemma mult-sound:**
assumes $sP$: sound $P$ and $sQ$: sound $Q$
shows sound $(\lambda s. P s \ast Q s)$

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

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

qed

**lemma div-sound:**
assumes $sP$: sound $P$ and $cpos$: $0 < c$
shows sound $(\lambda s. P s / c)$

**proof**
from $sP$ and $cpos$ have $\forall s. P s / c \leq \text{bound-of } P / c$
by(blast intro: divide-right-mono less-imp-le)
thus bounded-by $(\text{bound-of } P / c) (\lambda s. P s / c)$ by(blast)
from assms show nneg $(\lambda s. P s / c)$
by(blast intro: divide-nonneg-pos)

qed

**lemma tminus-sound:**
assumes $sP$: sound $P$ and $nnc$: $0 \leq c$

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

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

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

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

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

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

lemma sc-sound:
[ sound P; \theta \leq c ] \Longrightarrow sound (\lambda s. c * P s)
by (blast intro: mult-nonneg-nonneg)

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

lemma bounded-by-add:
fixes P::'s \Rightarrow \mathtt{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 setsim-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!: setsim-mono)
from sP show nneg (\lambda s. \sum x \in S. P x s)
by (auto intro!: setsim-nonneg)
qed
3.1.4 Unitary expectations

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

definition unitary :: 's expect ⇒ bool
where unitary P ←→ sound P ∧ bounded-by 1 P

lemma unitaryI[intro]:
  [ sound P; bounded-by 1 P ] ⇒ 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 » ≡ (λ s. if P s then 1 else 0)

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

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

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

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

Standard expectations have a number of convenient properties, which mostly follow from boolean algebra.
**Lemma** *embed-bool-idem*:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Lemma** *expect-1-I*:

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

*by (simp)*

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

sound \langle P \rangle

*by (blast)*

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

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

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

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

**Definition** *negate :: ('s ⇒ bool) ⇒ 's ⇒ bool (N)*

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

**Lemma** *negate1*:

\[
\neg P s \implies N P s
\]

*by (simp add:negate-def)*

**Lemma** *embed-split*:

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

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

lemma negate-embed:
«N P» s = 1 − «P» s
by (simp add:embed-bool-def negate-def)

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

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

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

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

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

3.1.6 Entailment

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

abbreviation entails :: ('s ⇒ real) ⇒ ('s ⇒ real) ⇒ bool (- ⊢ - 50)
where P ⊢ Q ≡ P ≤ Q

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

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

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

lemma entails-trans[trans]:
[ P ⊢ Q; Q ⊢ R ] ⇒ P ⊢ R
by (blast intro:order-trans)

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


**Lemma** implies-entails:

\[ \{ \bigvee s. \ P s \implies Q s \ \} \implies \{ \bigvee s. \ P s \vdash Q s \} \]

by (rule entailsI, case-tac P s, simp-all)

**Lemma** entails-implies:

\[ \bigwedge s. \ \{ \bigvee s. \ P s \vdash Q s \} \iff Q s \]

by (rule ccontr, drule-tac s=s in entailsD, simp)

### 3.1.7 Expectation Conjunction

**Definition**

\( \text{pconj} :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real} \) (infixl \& 71)

where

\[ p \ \& q \equiv p + q - 1 \]

**Definition**

\( \text{exp-conj} :: (\prime s \Rightarrow \text{real}) \Rightarrow (\prime s \Rightarrow \text{real}) \Rightarrow (\prime s \Rightarrow \text{real}) \) (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:

\( \langle a \rangle s \ \& \ \langle b \rangle s = \langle \lambda s. a s \land b s \rangle s \)

**Unfolding** embed-bool-def pconj-def tminus-def by (force)

**Lemma** pconj-comm[ac-simps]:

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

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

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

unfolding pconj-def tminus-def by(simp)

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

unfolding pconj-def tminus-def by(simp)

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

unfolding pconj-def tminus-def by(auto)

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

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

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

unfolding pconj-def by(simp)

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

unfolding pconj-def by(simp)

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

unfolding pconj-def by(auto simp-all)

3.1.8 Rules Involving Conjunction.

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

unfolding exp-conj-def pconj-def
by(auto intro:tminus-left-mono add-right-mono)

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

unfolding exp-conj-def pconj-def
by(auto intro:tminus-left-mono add-left-mono)

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

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

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

**proof** (rule bounded-byI, unfold \( \text{exp-conj-def pconj-def} \))

fix \( x \)

from \( bP \) have \( P x \leq 1 \) by (blast)

moreover from \( bQ \) have \( Q x \leq 1 \) by (blast)

ultimately have \( P x + Q x \leq 2 \) by (auto)

thus \( P x + Q x \odot 1 \leq 1 \)

unfolding \( \text{tminus-def} \) by (simp)

qed

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

\[
(P \&\& Q) \circ f = (P \circ f) \&\& (Q \circ f)
\]

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

**lemma** \( \text{exp-conj-assoc}: \)

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

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

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

**proof** (rule ext)

fix \( s \)

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

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

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

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

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

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

ultimately

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

by (simp add: pconj-assoc)

qed

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

sound \( P \implies "\lambda -. \ True" \&\& P = P \)

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

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

sound \( P \implies P \&\& "\lambda -. \ True" = P \)

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

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

\( "P" \&\& "P" = "P" \)

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

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

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

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

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

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

lemma \( \text{exp-conj-sound} \)\([\text{intro}, \text{simp}]\):
assumes \( s-P; \text{sound } P \)
and \( s-Q; \text{sound } Q \)
shows \( \text{sound } (P \&\& Q) \)
unfolding \( \text{exp-conj-def} \)
proof\( (\text{rule soundI}) \)
from \( s-P \) and \( s-Q \) have \( \forall s. 0 \leq P\ s + Q\ s \) by\( \text{blast \ intro: add-nonneg-nonneg} \)
hence \( \forall s. P\ s \&\& Q\ s \leq P\ s + Q\ s \)
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 \( \text{nng } (\lambda s. P\ s \&\& Q\ s) \)
unfolding \( \text{pconj-def \ tminus-def} \) by\( \text{force} \)
qed

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

lemma \( \text{exp-conj-1-right} \)\([\text{simp}]\):
assumes \( \text{nn: nneg } A \)
shows \( A \&\& (\lambda s. 1) = A \)
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 \( \text{exp-conj-std-split} \):
\( \langle \lambda s. P\ s \&\& Q\ s \rangle = \langle P\ \rangle \&\& \langle Q\ \rangle \)
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 expectation entailment, becoming expectation conjunction:

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

\textbf{thus} $(P \, \&\& \, Q) \, s \leq (R \, \&\& \, S) \, s$

unfolding \textit{exp-conj-def \ pconj-def}.

\textbf{qed}

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

\textbf{lemma pentails-cases}:
\hspace{1em}\hspace{1em} assumes $PQe$: \hspace{1em} $P x \, s \vdash \, Q x$
\hspace{1em} and exhaust: \hspace{1em} $\forall s. \, \exists x. \, P \, s \, x = 1$
\hspace{1em} and framed: \hspace{1em} $\forall x. \, P \, x \, \&\& \, R \, s \vdash \, Q \, x \, \&\& \, S$
\hspace{1em} and $sR$: \hspace{1em} sound $R$
\hspace{1em} and $sS$: \hspace{1em} sound $S$
\hspace{1em} and $bQ$: \hspace{1em} bounded-by 1 ($Q \, x$

shows $R \, s \vdash \, S$

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

\textbf{qed}

\textbf{lemma unitary-bot}[iff]:
\hspace{1em} unitary ($\lambda s. \, 0::real$
\hspace{1em} by (auto)

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

\textbf{lemma unitary-embed}[iff]:
\hspace{1em} unitary $\ll P \rr$
\hspace{1em} by (auto)

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

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

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

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

3.2. Expectation Transformers

theory Transformers imports Expectations begin
  type-synonym 's trans = 's expect $\Rightarrow$ 's expect

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

The set of healthy transformers is the universe into which we place our semantic interpretation of pGCL programs. In its standard presentation, the healthiness condition for pGCL programs is sublinearity, for demonic programs, and superlinearity for angelic programs. We extract a minimal core property, consisting of monotonicity, feasibility and scaling to form our healthiness property, which holds across all programs. The additional components of sublinearity are broken out separately, and shown later. The two reasons for this are firstly to avoid the effort of establishing sub-(super-)linearity globally, and to allow us to define primitives whose sublinearity, and indeed healthiness, depend on context.
Consider again the automaton of Figure 3.1. Here, the effect of executing the automaton from its initial state (a) until it reaches some final state (b or c) is to transform the expectation on final states ($P$), into one on initial states, giving the expected value of the function on termination. Here, the transformation is linear: $P_{\text{prior}}(a) = 0.7 * P_{\text{post}}(b) + 0.3 * P_{\text{post}}(c)$, but this need not be the case.

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

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

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

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

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

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

**Where**

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

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

**Definition**

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

**Where**

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

**Lemma** \( \text{le-transI[\text{intro}]} \):

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

by (simp add: le-trans-def)

**Lemma** \( \text{le-utransI[\text{intro}]} \):

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

by (simp add: le-utrans-def)

**Lemma** \( \text{le-transD[\text{dest}]} \):

\[ \left[ \text{le-trans } t u \right] \Rightarrow t P \leq u P \]

by (simp add: le-trans-def)

**Lemma** \( \text{le-utransD[\text{dest}]} \):

\[ \left[ \text{le-utrans } t u \right] \Rightarrow t P \leq u P \]

by (simp add: le-utrans-def)

**Lemma** \( \text{le-trans-trans[\text{trans}]} \):

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

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

**Lemma** \( \text{le-utrans-trans[\text{trans}]} \):

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

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

**Lemma** \( \text{le-trans-refl[\text{iff}]} \):

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

by (simp add: le-trans-def)

**Lemma** \( \text{le-utrans-refl[\text{iff}]} \):

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

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

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

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

Transformer equivalence is induced by comparison:

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

\textbf{definition}
\[ \text{equiv-utrans } :: \ 's \ \text{trans} \Rightarrow 's \ \text{trans} \Rightarrow \text{bool} \]
\textbf{where}
\[ \text{equiv-utrans } t \ u \longleftrightarrow \text{le-utrans } t \ u \land \text{le-utrans } u \ t \]

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

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

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

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

\textbf{lemma} equiv-trans-refl[\text{iff}]:
\text{equiv-trans } t \ t \quad \text{by}(blast)

\textbf{lemma} equiv-utrans-refl[\text{iff}]:
\text{equiv-utrans } t \ t \quad \text{by}(blast)

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

\textbf{lemma} le-utrans-antisym:
\[
[ \text{le-utrans } x \ y; \text{le-utrans } y \ x ] \implies \text{equiv-utrans } x \ y
\]
unfolding \text{equiv-utrans-def} by (simp)

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

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

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

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

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

\text{lemma} \text{equiv-uimp-le-alt}:
\text{equiv-utrans } t \ u \implies \text{le-utrans } u \ t
by (force simp: ac-simps)

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

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

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

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

\text{lemma} \text{le-trans-equiv-rsp-right[simp]}:
\text{equiv-trans } t \ u \implies \text{le-trans } v \ t \iff \text{le-trans } v \ u
unfolding \text{equiv-trans-def} by (blast intro: le-trans-trans)

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

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

lemma le-trans-equiv-trans [trans]:
\[
\begin{align*}
\text{[ le-trans } t \, \text{ u ; equiv-trans } u \, \text{ v ] } \Rightarrow \text{ le-trans } t \, \text{ v}
\end{align*}
\]
by (simp)

lemma le-utrans-equiv-utrans [trans]:
\[
\begin{align*}
\text{[ le-utrans } t \, \text{ u ; equiv-utrans } u \, \text{ v ] } \Rightarrow \text{ le-utrans } t \, \text{ v}
\end{align*}
\]
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 xy 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-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 \Rightarrow equiv-utrans t u
by (auto)

3.2.2 Healthy Transformers

Feasibility

definition feasible :: (('a ⇒ real) ⇒ ('a ⇒ real)) ⇒ bool
where feasible t \leftarrow\ (\forall P \, b. \, \text{ bounded-by } b \, P \, \land \, \text{ nneg } P \, \rightarrow
\text{ bounded-by } b \, (t \, P) \, \land \, \text{ nneg } (t \, P))

A feasible transformer preserves non-negativity, and bounds. A feasible transformer always takes its argument ‘closer to 0’ (or leaves it where it
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 feasible I[intro]:
\[ \bigwedge b P. \left[ \begin{array}{l} \text{bounded-by } b P; \text{nneg } P \end{array} \right] \implies \text{bounded-by } b \left(t P\right); \]
by(force simp; feasible-def)

lemma feasible-boundedD[dest]:
\[ \left[ \begin{array}{l} \text{feasible } t; \text{bounded-by } b P; \text{nneg } P \end{array} \right] \implies \text{bounded-by } b \left(t P\right) \]
by(simp add: feasible-def)

lemma feasible-nnegD[dest]:
\[ \left[ \begin{array}{l} \text{feasible } t; \text{bounded-by } b P; \text{nneg } P \end{array} \right] \implies \text{nneg } \left(t P\right) \]
by(simp add: feasible-def)

lemma feasible-sound[dest]:
\[ \left[ \begin{array}{l} \text{feasible } t; \text{sound } P \end{array} \right] \implies \text{sound } \left(t P\right) \]
by(rule soundI, unfold sound-def, (blast)+)

lemma feasible-pr-0[simp]:
fixes t::('s => real) => ('s => real)
assumes ft: \text{feasible } t
shows \(t \left(\lambda x. 0\right) = \left(\lambda x. 0\right)\)
proof(rule ext, rule antisym)
fix s

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

have \text{nneg } \left(\lambda ::'s, 0::real\right) by(blast)
with ft have \text{nneg } \left(t \left(\lambda ::0\right)\right) by(blast)
thus 0 \leq t \left(\lambda ::0\right) s by(blast)
qed

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

lemma feasible-bounded-by[dest]:
\[ \left[ \begin{array}{l} \text{feasible } t; \text{sound } P; \text{bounded-by } b P \end{array} \right] \implies \text{bounded-by } b \left(t P\right) \]
by(auto)

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

lemma feasible-fixes-bot:
3.2. EXPECTATION TRANSFORMERS

assumes \( ft: \text{feasible} \ t \)
shows \( t \ (\lambda s. 0) = (\lambda s. 0) \)
proof (rule antisym)
have \( \text{sb: sound} \ (\lambda s. 0) \) by (auto)
with \( ft \) show \( (\lambda s. 0) \leq t \ (\lambda s. 0) \) by (auto)
thm \( \text{bound-of-const} \)
from \( \text{sb have bounded-by} \ (\text{bound-of} \ (\lambda s. 0::\text{real})) \ (\lambda s. 0) \) by (auto)
hence \( \text{bounded-by} \ 0 \ (\lambda s. 0::\text{real}) \) by (simp add \( \text{bound-of-const} \))
with \( ft \) have \( \text{bounded-by} \ 0 \ (t \ (\lambda s. 0)) \) by (auto)
thus \( t \ (\lambda s. 0) \leq (\lambda s. 0) \) by (auto)
qed

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

Monotonicity

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

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

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

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

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

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

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

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

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

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

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

Healthiness

Healthy transformers are feasible and monotonic, and respect scaling

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

lemma healthyI[intro]:
[ feasible t; mono-trans t; scaling t ] =⇒ healthy t
by(simp add:healthy-def)
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[dest]:
  [ 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 $\text{healthy-fixes-bot} = \text{feasible-fixes-bot}(\text{OF healthy-feasibleD})$

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

lemma $\text{le-trans-bot}\{\text{intro},\text{simp}\}$:

\[ \text{healthy } t \implies \text{le-trans } (\lambda P \ s \ . \ 0) \ t \]
\[ \text{by } (\text{blast intro:le-funI}) \]

lemma $\text{le-trans-top}\{\text{intro},\text{simp}\}$:

\[ \text{healthy } t \implies \text{le-trans } t (\lambda P \ s \ . \ \text{bound-of } P) \]
\[ \text{by } (\text{blast intro:le-transI}(\text{OF le-funI})) \]

lemma $\text{healthy-pr-bot}\{\text{simp}\}$:

\[ \text{healthy } t \implies t (\lambda s \ . \ 0) = (\lambda s \ . \ 0) \]
\[ \text{by } (\text{blast intro:feasible-pr-0}) \]

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

lemma $\text{healthy-equivI}$:

\[ \text{fixes } t::(s \Rightarrow \text{real}) \Rightarrow s \Rightarrow \text{real and } u \]
\[ \text{assumes equiv: equiv-trans } t \ u \]
\[ \text{and healthy: healthy } t \]
\[ \text{shows healthy } u \]

proof

have $\text{le-t-u: le-trans } t \ u$ by$(\text{blast intro:equiv})$

have $\text{le-u-t: le-trans } u \ t$ by$(\text{simp add:equiv-imp-le ac-simps equiv})$

from equiv have eq-u-t: equiv-trans $u \ t$ by$(\text{simp add:ac-simps})$

show $\text{feasible } u$

proof

  fix $b$ and $P::s \Rightarrow \text{real}$
  assume $bP$: bounded-by $b \ P$ and $nP$: nneg $P$
  hence $sP$: sound $P$ by$(\text{blast})$
  with healthy have $\forall s. \ 0 \leq t \ P \ s$ by$(\text{blast})$
  also from $sP$ and \text{le-t-u} have $\forall s. \ ... \ s \leq u \ P \ s$ by$(\text{blast})$
  finally show $\text{nneg } (u \ P)$ by$(\text{blast})$

from $sP$ and \text{le-u-t} have $\forall s. \ u \ P \ s \leq t \ P \ s$ by$(\text{blast})$

also from $\text{healthy and } sP$ and $bP$ have $\forall s. \ t \ P \ s \leq b$ by$(\text{blast})$

finally show $\text{bounded-by } b \ (u \ P)$ by$(\text{blast})$

qed

show $\text{mono-trans } u$

proof

  fix $P::s \Rightarrow \text{real and } Q::s \Rightarrow \text{real}$
  assume $sP$: sound $P$ and $sQ$: sound $Q$
  and $\leq$: $P \vdash Q$
  from $sP$ and \text{le-u-t} have $u \ P \vdash t \ P$ by$(\text{blast})$
  also from $sP$ and $sQ$ and $\leq$ and healthy have $t \ P \vdash t \ Q$ by$(\text{blast})$
  also from $sQ$ and \text{le-t-u} have $t \ Q \vdash u \ Q$ by$(\text{blast})$
  finally show $u \ P \vdash u \ Q$.
3.2. EXPECTATION TRANSFORMERS

qed

show scaling u

proof

fix P::'s ⇒ real and c::real and x:'s

assume sound: sound P
and pos: 0 ≤ c

hence bounded-by (c * bound-of P) (λx. c * P x)
by(blast intro:mult-left-mono dest!:less-imp-le)

hence sc-bounded: bounded (λx. c * P x)
by(blast)

moreover from sound and pos have sc-nneg: nneg (λx. c * P x)
by(blast intro:mult-nonneg-nonneg less-imp-le)

ultimately have sc-sound: sound (λx. c * P x) by(blast)

show c * u P x = u (λx. c * P x) x

proof –

from sound have c * u P x = c * t P x
by(simp add:equiv-transD[OF eq-u-t])

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

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

finally show thesis .

qed

qed

lemma healthy-equiv:

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

lemma healthy-scale:

fixes t::('s ⇒ real) ⇒ 's ⇒ real

assumes ht: healthy t and nc: 0 ≤ c and bc: c ≤ 1

shows healthy (λP s. c * t P s)

proof

show feasible (λP s. c * t P s)

proof

fix b and P::'s ⇒ real

assume nnP: nneg P and bP: bounded-by b P

from ht nnP bP have ∨s. t P s ≤ b by(blast)
with nc have $\forall s. c \ast t P s \leq c \ast b$ by (blast intro: mult-left-mono)
also 
  from $nnP$ and $bP$ have $0 \leq b$ by (auto)
  with $bc$ have $c \ast b \leq 1 \ast b$ by (blast intro: mult-right-mono)
  hence $c \ast b \leq b$ by (simp)
} 
finally show bounded-by $b$ ($\lambda s. c \ast t P s$) by (blast)
from $ht nnP bP$ have $\forall s. 0 \leq t P s$ by (auto intro: le-funD)
with $nc$ have $\forall s. 0 \leq c \ast t P s$ by (rule mult-nonneg-nonneg)
thus $nneg$ ($\lambda s. c \ast t P s$) by (blast)
qed 
show mono-trans ($\lambda P s. c \ast t P s$) 
proof 
  fix $P :: \, \forall s := \mathbb{R}$ and $Q$
  assume $sP$ : sound $P$ and $sQ$ : sound $Q$ and $le$ : $P \vdash Q$
  with $ht$ have $\forall s. t P s \leq t Q s$ by (auto intro: le-funD)
  with $nc$ have $\forall s. c \ast t P s \leq c \ast t Q s$
  by (blast intro: mult-left-mono)
  thus $\forall s. c \ast t P s \vdash \forall s. c \ast t Q s$ by (blast)
qed 
from $ht$ show scaling ($\lambda P s. c \ast t P s$)
  by (auto simp: scalingD healthy-scalingD ht)
qed 

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

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

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

definition nearly-healthy :: ($\forall s : \mathbb{R}$) $\Rightarrow$ ($\forall s : \mathbb{R}$) $\Rightarrow$ bool where
  nearly-healthy $t$ $\leftrightarrow$ ($\forall P. \text{unitary } P \Rightarrow \text{unitary } (t P)$) $\land$
  ($\forall P. Q. \text{unitary } P \Rightarrow \text{unitary } Q \Rightarrow P \vdash Q \Rightarrow t P \vdash t Q$)

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

lemma nearly-healthy-monoD[dest]:


3.2. EXPECTATION TRANSFORMERS

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

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

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

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

3.2.3 Sublinearity

As already mentioned, the core healthiness property (aside from feasibility and continuity) for transformers is \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 \( x \ominus y \) represents \textit{truncated subtraction} i.e. \( \max(x - y) (0 ::'a) \) (see Section 4.13.1).

definition sublinear ::
\((('s ⇒ real) ⇒ ('s ⇒ real)) ⇒ \text{bool}\)
where
sublinear t ⇔ \( \forall a b c P Q s. \ (\text{sound } P ∧ \text{sound } Q ∧ 0 ≤ a ∧ 0 ≤ b ∧ 0 ≤ c) \) \( \Rightarrow a \ast t P s + b \ast t Q s \ominus c ≤ t (\lambda s'. a \ast P s' + b \ast Q s' \ominus c) s) \)

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

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

It is easier to see the relevance of sublinearity by breaking it into several component properties, as in the following sections.
Figure 3.4: A graphical depiction of sub-additivity as convexity.

Sub-additivity

**definition** sub-add ::

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

**where**

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

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.
3.2. EXPECTATION TRANSFORMERS

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

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

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

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

Can be rewritten, using scaling, to:

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

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

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

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

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

by (simp add: sub-add-def)

**lemma** sub-addI2:

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

sub-add \( t \)

by (auto)

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

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

by (simp add: sub-add-def)

**lemma** equiv-sub-add:

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

assumes eq: \( \text{equiv-trans } t \ u \)

and sa: \( \text{sub-add } t \)

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

**proof**

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

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

with \( \text{eq } \)

have \( t \ P s + u \ Q s = t \ P s + t \ Q s \)

by (simp add: equiv-transD)

also from \( \text{sP } sQ \) sa

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

by (auto)

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

Sublinearity and feasibility imply sub-additivity.

lemma sublinear-subadd:
  fixes t::('s⇒real)⇒'s⇒real
  assumes slt: sublinear t
  and ft: feasible t
  shows sub-add t
proof
  fix P::'s⇒real and Q::'s⇒real and s::'s
  assume sP: sound P and sQ: sound Q
  with ft have sound (t P) sound (t Q) by (auto)
  hence 0 ≤ t P s and 0 ≤ t Q s by (auto)
  hence 0 ≤ t P s + t Q s by (auto)
  hence ... = ... ⊖ 0 by (simp)
  also from sP sQ have ... ≤ t (λs. P s + Q s ⊖ 0) s
    by (rule sublinearD[OF slt, where a=1 and b=1 and c=0, simplified])
  also {
    from sP sQ have ∃s. 0 ≤ P s and ∃s. 0 ≤ Q s by (auto)
    hence ∃s. 0 ≤ P s + Q s by (auto)
    hence t (λs. P s + Q s ⊖ 0) s = t (λs. P s + Q s) s
      by (simp)
  }
finally show t P s + t Q s ≤ t (λs. P s + Q s) s.
qed

A few properties following from sub-additivity:

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

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

fix F::'a set and x::'a
assume IH: \lambda a. \sum y\in F. t (P y) a \vdash t (\lambda x. \sum y\in F. P y x)
hence t (P x) s + (\sum y\in F. t (P y) s) \leq t (P x) s + t (\lambda x. \sum y\in F. P y x) s
by(auto intro:add-left-mono)
also from sat sP
have ... \leq t (\lambda xa. P x xa + (\sum y\in F. P y xa)) s
by(auto intro!:sub-addD[OF sat setsum-sound])
finally
show t (P x) s + (\sum y\in F. t (P y) s) \leq t (\lambda xa. P x xa + (\sum y\in F. P y xa)) s
qed

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 * t « \lambda z. z = y » s) +
(\sum y\in\{s. \neg G s\}. P y * t « \lambda z. z = y » s) \leq t P s
proof
have \{s. G s\} \cap \{s. \neg G s\} = {} by(blast)
hence (\sum y\in\{s. G s\}. P y * t « \lambda z. z = y » s) +
(\sum y\in\{s. \neg G s\}. P y * t « \lambda z. z = y » s) =
(\sum y\in\{s. G s\} \cup \{s. \neg G s\}. P y * t « \lambda z. z = y » s)
by(auto intro: setsum.union_disjoint[symmetric])
also
have \{s. G s\} \cup \{s. \neg G s\} = UNIV by (blast)
hence (\sum y\in\{s. G s\} \cup \{s. \neg G s\}. P y * t « \lambda z. z = y » s) =
(\lambda x. \sum y\in UNIV. P y * t (\lambda x. «\lambda z. z = y») x) s
by(simp)
}
also {
from \( sP \) have \( \forall y. \ 0 \leq P \ y \) by(auto)

with \( \text{healthy-scalingD[OF \ ht]} \)

have \( (\lambda x. \ \sum y \in \text{UNIV}. \ P \ y \cdot t \ (\lambda x. \ « \lambda z. \ z = y » \ x) \) s = \\
(\lambda x. \ \sum y \in \text{UNIV}. \ t \ (\lambda x. \ P \ y \cdot « \lambda z. \ z = y » \ x) \) s \\
by(\text{simp \ add.scalingD})

} 

also \{ 

from \( \text{sat \ ht \ sP} \)

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

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

} 

also \{ 

have \( \text{rwl}: \ (\lambda x. \ \sum y \in \text{UNIV}. \ P \ y \cdot « \lambda z. \ z = y » \ x) = \\
(\lambda x. \ \sum y \in \text{UNIV}. \ \text{if} \ y = x \ \text{then} \ P \ y \ \text{else} \ 0) \) 
by(auto intro!:setsum.cong)

also from \( sP \) have \( \ldots \vdash P \) 
by(cases finite \( (\text{UNIV::}'s \ \text{set}) \), \text{auto simp:setsum.delta})

finally have \( \text{leP}: \ (\lambda x. \ \sum y \in \text{UNIV}. \ P \ y \cdot « \lambda z. \ z = y » \ x) \leq P \).

moreover have \( \text{sound} \ (\lambda x. \ \sum y \in \text{UNIV}. \ P \ y \cdot « \lambda z. \ z = y » \ x) \)

\textbf{proof}\( (\text{intro soundI2 bounded-byI nnegI setsum-nonneg ballI}) \)

fix \( x \)

from \( \text{leP} \) have \( (\sum y \in \text{UNIV}. \ P \ y \cdot « \lambda z. \ z = y » \ x) \leq P \ x \) by(auto)

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

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

fix \( y \)

from \( sP \) show \( 0 \leq P \ y \cdot « \lambda z. \ z = y » \ x \) 
by(auto intro!:mult-nonneg-nonneg)

qed 

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

} 

finally show \( \text{thesis} \).

qed

\textbf{Sub-distributivity}

definition sub-distrib ::

\( (('s \Rightarrow \text{real}) \Rightarrow ('s \Rightarrow \text{real})) \Rightarrow \text{bool} \)

where

\( \text{sub-distrib} \ t \leftarrow \ (\forall P \ s. \ \text{sound} \ P \longrightarrow t \ P \ s \oplus 1 \ \leq t \ (\lambda s'. \ P \ s' \ominus 1) \ s) \)

\textbf{lemma} \ sub-distrI[intro]:

\[ \bigwedge P \ s. \ \text{sound} \ P \Longrightarrow t \ P \ s \oplus 1 \ \leq t \ (\lambda s'. \ P \ s' \ominus 1) \ s \mapsto \text{sub-distrib} \ t \]

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

lemma sub-distrib12:

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

by(auto)

lemma sub-distribD[dest]:

\[ \left( \text{sub-distrib } t; \text{sound } P \right) \implies t \ P s \ominus 1 \leq t (\lambda s'. \ P s' \ominus 1) \]

by(simp add:sub-distrib-def)

lemma equiv-sub-distrib:

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

assumes eq: equiv-trans t u

and sd: sub-distrib t

shows sub-distrib u

proof

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

assume sP: sound P

moreover have sound (\lambda s. 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} and s::'s

assume sP: sound P

moreover have sound (\lambda s. 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 $ht \ sP \ sQ \ nna \ nmb$

have $saP$: sound $(\lambda s. \ a \ast \ P \ s)$ and $staP$: sound $(\lambda s. \ a \ast \ t \ P \ s)$
and $sbQ$: sound $(\lambda s. \ b \ast \ Q \ s)$ and $stbQ$: sound $(\lambda s. \ b \ast \ t \ Q \ s)$
by(auto intro:sc-sound)

**hence** $sabPQ$: sound $(\lambda s. \ a \ast \ P \ s \ast b \ast \ Q \ s)$
and $stabPQ$: sound $(\lambda s. \ a \ast \ t \ P \ s \ast b \ast \ t \ Q \ s)$
by(auto intro:sound-sum)

from $ht \ sP \ sQ \ nna \ nmb$

have $a \ast \ t \ P \ s \ast b \ast \ t \ Q \ s = \ t \ (\lambda s. \ a \ast \ P \ s) \ast t \ (\lambda s. \ b \ast \ Q \ s)$
s by(simp add:scalingD healthy-scalingD)

also from $saP \ sbQ \ sat$

have $t \ (\lambda s. \ a \ast \ P \ s) \ast t \ (\lambda s. \ b \ast \ Q \ s) \ast s \leq$
$t \ (\lambda s. \ a \ast \ P \ s \ast b \ast \ Q \ s) \ast s$ by(blast)

**finally**

have $notm$: $a \ast \ t \ P \ s \ast b \ast \ t \ Q \ s \leq t \ (\lambda s. \ a \ast \ P \ s \ast b \ast \ Q \ s)$

show $a \ast \ t \ P \ s \ast b \ast \ t \ Q \ s \ast c \leq t \ (\lambda s'. \ a \ast \ P \ s' \ast b \ast \ Q \ s' \ast c)$

**proof** (cases $c = 0$)

case True

have $\forall s. \ 0 \leq a \ast \ t \ P \ s \ast b \ast \ t \ Q \ s$ by(auto)

moreover from $sabPQ$

have $\forall s. \ 0 \leq a \ast \ P \ s \ast b \ast \ Q \ s$ by(auto)

ultimately show $\forall \theta$:thesis by(simp add:$z \ notm$)

**next**

case False

have $\forall s. \ (inverse c \ast a) \ast P \ s \ast (inverse c \ast b) \ast Q \ s =$

$inverse c \ast (a \ast P \ s \ast b \ast Q \ s)$

by(simp add: divide-simps)

with $sabPQ$ and $nni$

have $si$: sound $(\lambda s. \ (inverse c \ast a) \ast P \ s \ast (inverse c \ast b) \ast Q \ s)$

by(auto intro:sc-sound)

**hence** $sim$: sound $(\lambda s. \ (inverse c \ast a) \ast P \ s \ast (inverse c \ast b) \ast Q \ s \ast 1)$

by(auto intro!:tminus-sound)

from $nn$

have $a \ast \ t \ P \ s \ast b \ast \ t \ Q \ s \ast c =$

$(c \ast inverse c) \ast a \ast \ t \ P \ s +$

$(c \ast inverse c) \ast b \ast \ t \ Q \ s \ast c$

by(simp)

also

have $\ldots = c \ast (inverse c \ast a \ast \ t \ P \ s) +$

$c \ast (inverse c \ast b \ast \ t \ Q \ s) \ast c$

by(simp add:field-simps)

also from $nn$

have $\ldots = c \ast (inverse c \ast a \ast \ t \ P \ s \ast inverse c \ast b \ast \ t \ Q \ s \ast 1)$

by(simp add:distrib-left tminus-left-distrib)
3.2. EXPECTATION TRANSFORMERS

also \{ 
  have \( X : \bigwedge s. (inverse c \ast a) \ast t P s + (inverse c \ast b) \ast t Q s = \)
  \( inverse c \ast (a \ast t P s + b \ast t Q s) \) by(simp add: divide-simps)
also from \( \text{nmi and notm} \)
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 \( \text{nmi ht subPQ} \)
have \( t (\lambda s. (inverse c \ast a) \ast P s + (inverse c \ast b) \ast Q s) s = \)
\( (inverse c \ast a) \ast t P s + (inverse c \ast b) \ast t Q s \) by(simp add:scalingD(OF healthy-scalingD, OF ht) algebra-simps)
finally
have \( (inverse c \ast a) \ast t P s + (inverse c \ast b) \ast t Q s \leq \)
\( t (\lambda s. (inverse c \ast a) \ast P s + (inverse c \ast b) \ast Q s) s \leq 1 \)
by(rule tminus-left-mono)
also \{ 
  from \( \text{sdt si} \)
  have \( t (\lambda s. (inverse c \ast a) \ast P s + (inverse c \ast b) \ast Q s) s \leq 1 \leq \)
  \( t (\lambda s. (inverse c \ast a) \ast P s + (inverse c \ast b) \ast Q s \leq 1) s \)
  by(blast)
\}
finally
have \( c \ast ((inverse c \ast a) \ast t P s + inverse c \ast b \ast t Q s \leq 1) \leq \)
\( c \ast t (\lambda s. inverse c \ast a \ast P s + inverse c \ast b \ast Q s \leq 1) s \)
using \( nnc \) by(blast intro:mult-left-mono)
also from \( \text{nnc ht sim} \)
have \( c \ast t (\lambda s. inverse c \ast a \ast P s + inverse c \ast b \ast Q s \leq 1) s = \)
\( t (\lambda s. (c \ast inverse c) \ast a \ast P s + \)
\( (c \ast inverse c) \ast b \ast Q s \leq 1) s \)
by(simp add:distrib-left tminus-left-distrib)
also have \( \ldots = t (\lambda s. (c \ast inverse c) \ast a \ast P s + \)
\( (c \ast inverse c) \ast b \ast Q s \leq 1) s \)
by(simp add:field-simps)
finally
show \( a \ast t P s + b \ast t Q s \leq t (\lambda s'. a \ast P s' + b \ast Q s' \leq 1) s \)
using \( nz \) by(simp)
qed
qed

Sub-conjunctivity

definition
  \( \text{sub-conj} :: (('s \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real}) \Rightarrow \text{bool} \)
where
  \( \text{sub-conj} t \equiv \forall P Q. (\text{sound } P \land \text{sound } Q) \rightarrow \)
  \( t P \land t Q \vdash t (P \land Q) \)
CHAPTER 3. SEMANTIC STRUCTURES

lemma sub-conjI[intro]:
\[ \land P \, Q, \land \text{sound } P; \land \text{sound } Q \implies t \, P \land t \, Q \implies t \, (P \land Q) \implies \text{sub-conj } t \]

unfolding sub-conj-def by(simp)

lemma sub-conjD[dest]:
\[ \text{sub-conj } t; \land \text{sound } P; \land \text{sound } Q \implies t \, P \land t \, Q \implies t \, (P \land Q) \]

unfolding sub-conj-def by(simp)

lemma sub-conj-wp-twice:
fixes f :: \( \land \text{s} \implies \text{real} \) \implies \land \text{s} \implies \text{real} 
assumes all: \( \forall s. \text{sub-conj (f s)} \)
shows sub-conj (\( \lambda s. f s \, P \, s \))
proof (rule sub-conjI, rule le-funI)
fix P: \( \land s \Rightarrow \text{real} \) and Q: \( \land s \Rightarrow \text{real} \) and s
assume sP: sound P and sQ: sound Q
have (\( \lambda s. f s \, P \, s \)) \land (\( \lambda s. f s \, Q \, s \)) \leq (\( f s \, P \land f s \, Q \)) s
by(simp add:exp-conj-def)
also {
  from all have sub-conj (f s) by(blast)
  with sP and sQ have (\( f s \, P \land f s \, Q \)) s \leq (\( f s \, P \land f s \, Q \)) s
  by(blast)
}
finally show (\( \lambda s. f s \, P \, s \)) \land (\( \lambda s. f s \, Q \, s \)) \leq (\( f s \, P \land f s \, Q \)) s
qed

Sublinearity implies sub-conjunctivity:

lemma sublinear-sub-conj:
fixes t::\( \land s \Rightarrow \text{real} \Rightarrow \land s \Rightarrow \text{real} \)
assumes slt: sublinear t
shows sub-conj t
proof (rule sub-conjI, rule le-funI, unfold exp-conj-def pconj-def)
fix P::\( \land s \Rightarrow \text{real} \) and Q::\( \land s \Rightarrow \text{real} \) and s::\( \land s \)
assume sP: sound P and sQ: sound Q
thus t \, P \, s + t \, Q \, s \ominus 1 \leq t \, (\( \lambda s. P \, s + Q \, s \ominus 1 \)) s
by(rule sublinearD[OF slt, where a=1 and b=1 and c=1, simplified])
qed

Sublinearity under equivalence

Sublinearity is preserved by equivalence.

lemma equiv-sublinear:
\[ \text{equiv-trans } t \, w; \text{sublinear } t; \text{healthy } t \] \implies \text{sublinear } u
by (iprover intro:sd-sa-sublinear healthy-equivI)
  dest:equiv-sub-distrib equiv-sub-add
  sublinear-sub-distrib sublinear-subadd
  healthy-feasibleD)
3.2. 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**

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

**where**

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

**lemma** **additiveD:**

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

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

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

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

by **(blast)**

Additivity is strictly stronger than sub-additivity.

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

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

by **(simp add:sub-addI additiveD)**

The additivity property extends to finite summation.

**lemma** **additive-setsum:**

**fixes** \( S :: 's set \)

**assumes** **additive**: additive \( t \)

**and** **healthy**: healthy \( t \)

**and** **finite**: finite \( S \)

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

**shows** \( t (\lambda x. \sum y \in S. P y x) = (\lambda x. \sum y \in S. t (P y) x) \)

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

**fix** \( z :: 's \) 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!:setsum-sound additiveD)**

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

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

**Lemma additive-delta-split:**

fixes t::('s::finite ⇒ real) ⇒ 's ⇒ real
assumes additive: additive t
and ht: healthy t
and sP: sound P
shows t P x = (∑ y∈UNIV. P y * t «λz. z = y» x)
proof –
  have (∀x. (∑ y∈UNIV. P y * «λz. z = y» x) =
    (∑ y∈UNIV. if y = x then P y else 0))
    by(auto intro:setsum.cong)
  also have (∀x. ... x = P x)
    by(simp add:setsum.delta)
finally
  have t P x = t (λx. ∑ y∈UNIV. P y * «λz. z = y» x) x
    by(simp)
  also { from sP have (∀z. sound (λa. P z * «λza. za = z » a))
    by(auto intro:mult-sound)
  hence t (λx. ∑ y∈UNIV. P y * «λz. z = y» x) x =
    (∑ y∈UNIV. t (λx. P y * «λz. z = y» x) x)
    by(subst additive-setsum, simp-all add:assms)
}
also from sP
  have (∑ y∈UNIV. t (λx. P y * «λz. z = y» x) x) =
    (∑ y∈UNIV. P y * t «λz. z = y» x)
    by(subst scalingD[OF healthy-scalingD, OF ht], auto)
finally show t P x = (∑ y∈UNIV. P y * t «λz. z = y» x).
qed

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

**Lemma additive-guard-split:**

fixes t::('s::finite ⇒ real) ⇒ 's ⇒ real
assumes additive: additive t
and ht: healthy t
and sP: sound P
shows t P x = (∑ y∈{s. G s}. P y * t «λz. z = y» x) +
  (∑ y∈{s. ¬G s}. P y * t «λz. z = y» x)
proof –
from assms
have $t \cdot P \cdot x = \left( \sum_{y \in \text{UNIV}.} P \cdot y \cdot t \lambda z\cdot z = y\cdot x \right)$
by(rule additive-delta-split)
also 

have $\text{UNIV} = \{s. G \cdot s\} \cup \{s. \neg G \cdot s\}$
by(auto)

hence 
$\left( \sum_{y \in \text{UNIV}.} P \cdot y \cdot t \lambda z\cdot z = y\cdot x \right) = \left( \sum_{y \in \{s. G \cdot s\}} P \cdot y \cdot t \lambda z\cdot z = y\cdot x \right)$
by(simp)

also 

have $\left( \sum_{y \in \{s. G \cdot s\}} P \cdot y \cdot t \lambda z\cdot z = y\cdot x \right) = \left( \sum_{y \in \{s. \neg G \cdot s\}} P \cdot y \cdot t \lambda z\cdot z = y\cdot x \right)$
by(auto intro:setsum.union-disjoint)

finally show \thesis.
qed

Maximality

definition

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

where

$\text{maximal} \cdot t \equiv \forall c. \ 0 \leq c \Rightarrow t \lambda -. \ c = (\lambda -. \ c)$

lemma maximalI[intro]:

$[ \forall c. \ 0 \leq c \Rightarrow t \lambda -. \ c = (\lambda -. \ c)] \Rightarrow \text{maximal} \cdot t$

by(simp add:maximal-def)

lemma maximalD[dest]:

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

by(simp add:maximal-def)

A transformer that is both additive and maximal is deterministic:

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

where

$\text{determ} \cdot t \equiv \text{additive} \cdot t \land \text{maximal} \cdot t$

lemma determI[intro]:

$[ \text{additive} \cdot t; \text{maximal} \cdot t ] \Rightarrow \text{determ} \cdot t$

by(simp add:determ-def)

lemma determ-additiveD[intro]:

$\text{determ} \cdot t \Rightarrow \text{additive} \cdot t$

by(simp add:determ-def)

lemma determ-maximalD[intro]:

$\text{determ} \cdot t \Rightarrow \text{maximal} \cdot t$

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** t (λs. « N P » s) s = 1

by (simp)

finally show ?thesis .

**Qed**

### 3.2.5 Modular Reasoning

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

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

**Lemma entails-combine:**

**Assumes** wp1: P ⊢ t R

and wp2: Q ⊢ t S

and sc: sub-conj t

and sR: sound R

and sS: sound S

**Shows** P & & Q ⊢ t (R & & S)

**Proof**

from wp1 and wp2 have P & & Q ⊢ t R & & t S

by (blast intro: entails-frame)

also from sc and sR and sS have ... ≤ t (R & & S)

by (rule sub-conjD)

finally show ?thesis .

**Qed**

These allow mismatched results to be composed

**Lemma entails-strengthen-post:**

[ P ⊢ t Q; healthy t; sound R; Q ⊢ R; sound Q ] ⇒ P ⊢ t R

by (blast intro: entails-trans)
### 3.2. EXPECTATION TRANSFORMERS

**Lemma** \textit{entails-weaken-pre}:
\[
\left[ Q \vdash t ~R; ~P \vdash Q \right] \Rightarrow \ P \vdash t ~R
\]
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** \textit{entails-scale}:

\begin{itemize}
  \item \text{assumes} wp: \ P \vdash t \ Q \text{ and } h: \text{healthy } t
  \item \text{and } sQ: \text{sound } Q \text{ and } \text{pos}: \ 0 \leq c
  \item \text{shows} (\lambda s. \ c \ast P \ s) \vdash t \ (\lambda s. \ c \ast Q \ s)
\end{itemize}

\textbf{proof} (rule le-funI)

\begin{itemize}
  \item \text{fix } s
  \item \text{from } \text{pos and } \text{wp have } \ c \ast P \ s \leq c \ast t \ Q \ s
    \text{ by (auto intro: mult-left-mono)}
  \item \text{with } sQ \text{ pos } h \text{ show } \ c \ast P \ s \leq t \ (\lambda s. \ c \ast Q \ s) \ s
    \text{ by (simp add: scalingD healthy-scalingD)}
\end{itemize}

\textbf{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** \textit{use-premise}:

\begin{itemize}
  \item \text{assumes} h: \text{healthy } t \text{ and } wP: \bigwedge s. \ P \ s \implies 1 \leq t \ «Q» \ s
  \item \text{shows} «P» \vdash t \ «Q»
\end{itemize}

\textbf{proof} (rule entailsI)

\begin{itemize}
  \item \text{fix } s \text{ show } «P» \ s \leq t \ «Q» \ s
    \text{ by (cases P s)}
  \item \text{case True with } wP \text{ show } \ ?\text{thesis by (auto)}
  \item \text{next}
    \item \text{case False with } h \text{ show } \ ?\text{thesis by (auto)}
\end{itemize}

\textbf{qed}

The other direction works too.

**Lemma** \textit{fold-premise}:

\begin{itemize}
  \item \text{assumes} ht: \text{healthy } t
  \item \text{and wp: } «P» \vdash t \ «Q»
  \item \text{shows} \ \forall s. \ P \ s \implies 1 \leq t \ «Q» \ s
\end{itemize}

\textbf{proof} (clarify)

\begin{itemize}
  \item \text{fix } s \text{ assume } P \ s
    \item \text{hence } 1 = «P» \ s \text{ by (simp)}
    \item \text{also from } \text{wp have } \ ... \leq t \ «Q» \ s \text{ by (auto)}
    \item \text{finally show } 1 \leq t \ «Q» \ s \text{ .}
\end{itemize}

\textbf{qed}
Predicate conjunction behaves as expected:

**Lemma** \( \text{conj-post} \):

\[
P \vdash t \downarrow \lambda s. \ Q s \land R s; \ \text{healthy } t \quad \implies \quad P \vdash t \downarrow Q
\]

by (\text{blast intro: entails-strengthen-post implies-entails})

Similar to \([\text{healthy } ?t; \ \land s. \ ?P s \implies \ 1 \leq ?t \downarrow \ ?Q s] \implies \ ?P \vdash ?t \downarrow ?Q\), but more general.

**Lemma** \( \text{entails-pconj-assumption} \):

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

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

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

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

**Proof** (rule \( \text{entailsI} \))

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

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

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

- next
  
  case \( \text{False} \)
  
  moreover from \( uQ \) have \( Q s \leq 1 \) by (auto)
  
  ultimately show \( \theta \text{thesis} \) using assms by (simp, blast)

qed

qed

end

### 3.3 Induction

**Theory** \( \text{Induction} \)

**Imports** \( \text{Expectations Transformers Conditionally-Complete-Lattices} \)

**Begin**

#### 3.3.1 The Lattice of Expectations

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

Finding a lower bound is easy (it’s \( \lambda\. \ 0::'b \)), 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 \( \lambda s. \text{ bound-of } P \).

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

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

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

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

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

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

**lemma** Sup-exp-sound:
assumes sS: \( \bigwedge P. P \in S \Rightarrow \text{ sound } P \)
and bS: \( \bigwedge P. P \in S \Rightarrow \text{ bounded-by } b P \)
shows sound (Sup-exp \( S \))
**proof**
(cases \( S=\{\}, \text{ simp add:Sup-exp-def, blast, intro sound12 bounded-byI2 nnegI2} \))
assume neS: \( S \neq \{\} \)
then obtain \( P \) where Pin: \( P \in S \) by(auto)
with sS bS have \( nP: \text{ nneg } P \text{ bounded-by } b P \) by(auto)
hence nb: \( 0 \leq b \) by(auto)

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

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

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

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

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

lemma $\text{feasible-lfp-exp-sound}$:
feasible $t$ = $\Rightarrow \text{sound } (\text{lfp-exp } t)$
by(intro soundI2 bounded-byI2 nnegI2, auto intro:!:(lfp-exp-lowerbound lfp-exp-greatest))

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

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

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

lemma $\text{lfp-exp-lemma2}$:
fixes $t$: $'s$ trans
assumes $st$: $\bigwedge P. \text{sound } P \Rightarrow \text{sound } (t P)$
3.3. INDUCTION

\texttt{and \textit{mt}: \textit{mono-trans \textit{t}}} \\
\texttt{and \textit{fr}: \textit{t \overset{R}{\Rightarrow} R} \textit{and \textit{sR}: sound \textit{R}}} \\
\texttt{shows \textit{t} (lfp-exp \textit{t}) \leq lfp-exp \textit{t}} \\
\texttt{proof} (\texttt{rule lfp-exp-greatest})

\texttt{from \textit{fr} \textit{sR} show sound \textit{(t (lfp-exp \textit{t})}) by(auto intro: lfp-exp-sound \textit{st})}

\texttt{fix \textit{P}::\textit{\{trans \textit{t} \Rightarrow \textit{w expect \textit{t}\}}}} \\
\texttt{assume \textit{ff}: \textit{t \overset{P}{\Rightarrow} P} \textit{and \textit{sP: sound \textit{P}}} \\
\texttt{hence \textit{lfp-exp \textit{t} \overset{P}{\Rightarrow} \textit{P}} by(\texttt{rule lfp-exp-lowerbound})} \\
\texttt{with \textit{fp \textit{sp}} have \textit{t (lfp-exp \textit{t}) \overset{P}{\Rightarrow} t \textit{P} by(auto intro: mono-trans \textit{mt \textit{fr \textit{sR}}})} \\
\texttt{also note \textit{fp}}} \\
\texttt{finally show \textit{t (lfp-exp \textit{t}) \overset{P}{\Rightarrow}}} \\
\texttt{qed}

\texttt{lemma lfp-exp-lemma3:} \\
\texttt{assumes \textit{st}: \forall \textit{P}. sound \textit{P} \Rightarrow sound \textit{(t \textit{P})}} \\
\texttt{and \textit{mt}: \textit{mono-trans \textit{t}}} \\
\texttt{and \textit{fr}: \textit{t \overset{R}{\Rightarrow} R} \textit{and \textit{sR: sound \textit{R}}} \\
\texttt{shows \textit{lfp-exp \textit{t} \leq t \textit{of \textit{t}}} by(\texttt{iprover intro: lfp-exp-lowerbound \textit{assms \textit{mt \textit{fr \textit{sR}}}}})}

\texttt{lemma lfp-exp-unfold:} \\
\texttt{assumes \textit{nt}: \forall \textit{P}. sound \textit{P} \Rightarrow sound \textit{(t \textit{P})}} \\
\texttt{and \textit{mt}: \textit{mono-trans \textit{t}}} \\
\texttt{and \textit{fr}: \textit{t \overset{R}{\Rightarrow} R} \textit{and \textit{sR: sound \textit{R}}} \\
\texttt{shows \textit{lfp-exp \textit{t} = t \textit{(lfp-exp \textit{t})}} by(\texttt{iprover intro: antisym \textit{lfp-exp-lemma2 \textit{assms \textit{mt \textit{fr \textit{sR}}}}})}

\texttt{definition gfp-exp :: \textit{\{trans \textit{t} \Rightarrow \textit{w expect \textit{t}\}}}} \\
\texttt{where \textit{gfp-exp \textit{t} = Sup-exp \{P. unitary \textit{P} \Rightarrow \textit{P \leq t \textit{P}}\}}} \\
\texttt{lemma gfp-exp-upperbound:} \\
\texttt{[ \textit{P \leq t \textit{P}; unitary \textit{P} \Rightarrow P \leq gfp-exp \textit{t}}} \\
\texttt{by(auto simp: gfp-exp-def intro: Sup-exp-upper})

\texttt{lemma gfp-exp-least:} \\
\texttt{[ \textit{\forall \textit{P}. \textit{unitary \textit{P} \Rightarrow P \leq Q}; unitary \textit{Q} \Rightarrow gfp-exp \textit{t} \leq Q}\]} \\
\texttt{unfolding gfp-exp-def \texttt{by(auto intro: Sup-exp-least})}

\texttt{lemma gfp-exp-bound:} \\
\texttt{(\textit{\forall \textit{P}. unitary \textit{P} \Rightarrow unitary \textit{(t \textit{P})}) \Rightarrow bounded-by 1 \textit{(gfp-exp \textit{t})}} \\
\texttt{unfolding gfp-exp-def \texttt{by(rule bounded-by12 \textit{OF Sup-exp-least})}} \\
\texttt{auto}

\texttt{lemma gfp-exp-nneg[iff]:} \\
\texttt{nneg \textit{(gfp-exp \textit{t})}} \\
\texttt{proof(intro nnegI2, simp add: gfp-exp-def, cases})} \\
\texttt{assume empty: \textit{\{P. unitary \textit{P} \Rightarrow P \Rightarrow t \textit{P}\} = \{\}}}
CHAPTER 3. SEMANTIC STRUCTURES

show λs. 0 ⊢ Sup-exp {P. unitary P ∧ P ⊢ t P}
   by(simp only:empty Sup-exp-def, auto)
next
assume {P. unitary P ∧ P ⊢ t P} ≠ {}
then obtain Q where Qin: Q ∈ {P. unitary P ∧ P ⊢ t P} by(auto)
hence λs. 0 ⊢ Q by(auto)
also from Qin have Q ⊢ Sup-exp {P. unitary P ∧ P ⊢ t P}
   by(auto intro:Sup-exp-upper)
finally show λs. 0 ⊢ Sup-exp {P. unitary P ∧ P ⊢ t P}.
qed

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

lemma gfp-exp-lemma2:
assumes ft: ∀P. unitary P ⇒ unitary (t P)
   and mt: ∀P Q. [ unitary P; unitary Q; P ⊢ ⊢ Q ] ⇒ t P ⊢ t Q
shows gfp-exp t ≤ t (gfp-exp t)
proof(rule gfp-exp-least)
show unitary (t (gfp-exp t)) by(auto intro:gfp-exp-lemma1 ft)
fix P
assume fp: P ≤ t P and uP: unitary P
with ft have P ≤ gfp-exp t by(auto intro:gfp-exp-upperbound)
with uP gfp-exp-unitary ft
have t P ≤ t (gfp-exp t) by(blast intro: mt)
with fp show P ≤ t (gfp-exp t) by(auto)
qed

lemma gfp-exp-lemma3:
assumes ft: ∀P. unitary P ⇒ unitary (t P)
   and mt: ∀P Q. [ unitary P; unitary Q; P ⊢ ⊢ Q ] ⇒ t P ⊢ t Q
shows t (gfp-exp t) ≤ gfp-exp t
by(iprover intro:gfp-exp-upperbound unitary-sound
   gfp-exp-unitary gfp-exp-lemma2 assms)

lemma gfp-exp-unfold:
(∀P. unitary P ⇒ unitary (t P)) ⇒ (∀P Q. [ unitary P; unitary Q; P ⊢ ⊢ Q ] ⇒ t P ⊢ t Q) ⇒
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 λ- -, 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** $\text{Inf-trans } S = (\lambda P. \text{Inf-exp \{ } t P \mid t \in S \})$

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

**unfolding** $\text{Inf-trans-def}$  
by (rule $\text{le-transI[OF Inf-exp-lower]}$, 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!: $\text{le-transI[OF Inf-exp-greatest]}$)

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

**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 $\text{le-utransI[OF Sup-exp-upper]}$, auto intro!: $\text{unitary-bound}$)

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

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

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

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

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

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

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

**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 $nS$ obtain $t$ where (in: $t \in S$) by(auto)
with $fS$ have $ft$: 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 $fS$) 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 $fS \ bP \ nP$
show bounded-by $b$ ($\text{Sup-trans} \ S \ P$)
  by(auto intro!:bounded-byI2[OF Sup-trans-least2])
qed

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

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

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

lemma $\text{lfp-trans-sound}$:
fixes $P \ Q$: $\forall s \ \text{expect}$
assumes $sP$: $\text{sound} \ P$
  and $fv$: $\text{le-trans} \ (T \ v) \ v$
  and $sv$: $\forall P. \ \text{sound} \ P \Rightarrow \text{sound} \ (v \ P)$
shows $\text{sound} \ (\text{lfp-trans} \ T \ P)$
proof(intro soundI2 bounded-byI2 nnegI2)
from $fv \ sv$ have $\text{le-trans} \ (\text{lfp-trans} \ T) \ v$
  by(intprover intro:!lfp-trans-lowerbound)
with $sP$ have $\text{lfp-trans} \ T \ P \vdash v \ P$ by(auto)
also {
  from $sv \ sP$ have $\text{sound} \ (v \ P)$ by(intprover)
    hence $v \ P \vdash \lambda s. \ \text{bound-of} \ (v \ P)$ by(auto)
}
finally show $\text{lfp-trans} \ T \ P \vdash \lambda s. \ \text{bound-of} \ (v \ P)$.

have $\text{le-trans} \ (\lambda P \ s. \ 0) \ (\text{lfp-trans} \ T)$
proof(intro !lfp-trans-greatest)
3.3. INDUCTION

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

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

\[ \text{qed} \]

\textbf{lemma \( lfp-trans-lemma3 \):}
fixes \( v \) : \( \text{\`s trans} \)
assumes mono: \( \forall t \ u. \ [ \text{le-trans } t \ u; \ \forall P. \ \text{sound } P \implies \text{sound } (t \ P); \ \\forall P. \ \text{sound } P \implies \text{sound } (u \ P) ] \implies \text{le-trans } (T \ t) (T \ u) \)
and \( sT \): \( \forall t \ P. \ [ [ \forall Q. \ \text{sound } Q \implies \text{sound } (t \ Q); \ \text{sound } P ] \implies \text{sound } (T \ t \ P) \)
and \( fu \): \( \text{le-trans } (T \ v) \ v \)
and \( sv \): \( \forall P. \ \text{sound } P \implies \text{sound } (v \ P) \)
shows le-trans \( (lfp-trans T) (T (lfp-trans T)) \)
\textbf{proof (rule \( lfp-trans-lowerbound \))}
fix \( P \) : \( \text{\`s expect} \)
assume \( sP \): \( \text{sound } P \)
have \( n1 : \forall P. \ \text{sound } P \implies \text{sound } (lfp-trans T \ P) \)
by (iprover intro \( lfp-trans-sound \) \( fv \) \( sv \))
with \( sP \) have \( n2 : \text{sound } (lfp-trans T \ P) \)
by (iprover intro \( lfp-trans-sound \) \( fv \) \( sv \) \( sT \))
with \( n1 \ sP \) show \( n3 : \text{sound } (T (lfp-trans T) \ P) \)
by (iprover intro \( sT \))
next
show le-trans \( (T (T (lfp-trans T))) (T (lfp-trans T)) \)
by (rule mono [OF \( lfp-trans-lemma2 \), OF mono],
(iprover intro assms \( lfp-trans-sound \))+)
\[ \text{qed} \]

\textbf{lemma \( lfp-trans-unfold \):}
fixes \( P \) : \( \text{\`s expect} \)
assumes mono: \( \forall t \ u. \ [ \text{le-trans } t \ u; \ \forall P. \ \text{sound } P \implies \text{sound } (t \ P); \ \\forall P. \ \text{sound } P \implies \text{sound } (u \ P) ] \implies \text{le-trans } (T \ t) (T \ u) \)
and \( fu \): \( \text{le-trans } (T \ v) \ v \)
and \( sv \): \( \forall P. \ \text{sound } P \implies \text{sound } (v \ P) \)
shows equiv-trans \( (lfp-trans T) (T (lfp-trans T)) \)
by (rule le-trans-antisym,
rule \( lfp-trans-lemma2 \) [OF mono], (iprover intro assms)+,
rule \( lfp-trans-lemma2 \) [OF mono], (iprover intro assms)+)
\textbf{definition \( gfp-trans :: (\text{\`s trans } \implies \text{\`s trans}) \implies \text{\`s trans} \)}
\textbf{where \( gfp-trans T = \text{Sup-trans } \{ t. (\forall P. \text{unitary } P \implies \text{unitary } (t \ P)) \land \text{le-utrans } t \ (T \ t) \} \)}
\textbf{lemma \( gfp-trans-upperbound \):}
\[ \forall \text{le-utrans } t \ (T \ t); \ \forall P. \ \text{unitary } P \implies \text{unitary } (t \ P) \] \implies \text{le-utrans } t \ (gfp-trans T)
\textbf{unfolding \( gfp-trans-def \) by (auto intro: Sup-trans-upper)}
3.3. INDUCTION

**Lemma 3.3.1 (gfp-trans-least):**

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

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

**Lemma 3.3.2 (gfp-trans-unitary):**

fixes P :: 's
assumes uP :: unitary P
shows unitary (gfp-trans T P)
proof (intro unitaryI2 nnegI2 bounded-byI2)
show \( \forall t. \forall P. \text{unitary } P \implies \text{unitary } (t P) \)
by (auto)
thus \( t P \vdash \lambda s. 1 \)
by (auto)
next
show \( \text{neginj } (\lambda s. 1) \)
by (auto)
qed

let \( ?S = \{ t P \mid \forall P. \text{unitary } P \implies \text{unitary } (t P) \} \)
show \( \lambda s. 0 \vdash \text{gfp-trans } T P \)
unfolding gfp-trans-def Sup-trans-def
proof (cases)
assume \( ?S = \{ \} \)
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 unitary Q
by (auto)
hence \( \lambda s. 0 \vdash Q \)
by (auto)
also with uP Qin have \( Q \vdash \text{Sup-exp } ?S \)
proof (intro Sup-exp-upper, blast, clarify)
fix t :: 's
assume \( \forall Q. \text{unitary } Q \implies \text{unitary } (t Q) \)
with uP show bounded-by 1 (t P)
by (auto)
qed
finally show \( \lambda s. 0 \vdash \text{Sup-exp } ?S \)
qed
qed

**Lemma 3.3.3 (gfp-trans-lema2):**

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

proof\((\text{rule gfp-trans-least, simp-all add:} hT \ gfp-trans-unitary)\)

fix \( t \)

assume \( fp: \text{le-utrans} \ t \ (T \ t) \) and \( ht: \bigwedge P. \text{unitary} \ P \implies \text{unitary} \ (t \ P) \)

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

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

qed

lemma \( gfp-trans-lemma3: \)

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

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

shows \( \text{le-utrans} \ (T \ (gfp-trans T)) \) (\( gfp-trans T \)) by\((\text{blast intro!: mono gfp-trans-unitary gfp-trans-upperbound gfp-trans-lemma2 mono hT})\)

lemma \( gfp-trans-unfold: \)

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

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

shows \( \text{equiv-utrans} \ (gfp-trans T) \) (\( T \ (gfp-trans T) \)) using\((\text{assms by}(\text{auto intro!: le-utrans-antisym gfp-trans-lemma2 gfp-trans-lemma3}))\)

3.3.3 Tail Recursion

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

lemma \( gfp-pulldown: \)

fixes \( P::\text{’s expect} \)

assumes \( \text{tailcall}: \bigwedge u P. \text{unitary} \ P \implies \text{unitary} \ (t \ P \ u) \)

and \( ft: \bigwedge t P. \left[ \bigwedge Q. \text{unitary} \ Q \implies \text{unitary} \ (t \ Q); \text{unitary} \ P \right] \implies \text{unitary} \ (T \ t \ P) \)

and \( ft: \bigwedge P Q. \text{unitary} \ P \implies \text{unitary} \ Q \implies \text{unitary} \ (t \ P \ Q) \)

and \( mt: \bigwedge P Q R. \left[ \text{unitary} \ P; \text{unitary} \ Q; \text{unitary} \ R; Q \vdash R \right] \implies t \ P \)

\( Q \vdash t P R \)

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

and \( \text{monoT}: \bigwedge t u. \left[ \text{le-utrans} \ t \ u; \bigwedge P. \text{unitary} \ P \implies \text{unitary} \ (t \ P); \bigwedge P. \text{unitary} \ P \implies \text{unitary} \ (u \ P) \right] \implies \text{le-utrans} \ (T \ t) \ (T \ u) \)

shows \( \text{gfp-trans} \ T \ P = \text{gfp-exp} \ (t \ P) \) (\( \text{is} ?X P = ?Y P \))
proof\(\) (rule antisym)
show ?X P ≤ ?Y P
proof\(\) (rule gfp-exp-upperbound)
from mono T \( T\) \( uP\) have \( (gfp\text{-}trans \ T \ P) \leq (T \ (gfp\text{-}trans \ T \ P)) \ P\)
  by(auto intro!: le-utransD[OF gfp-trans-lemma2])
also from \( uP\) have \( (T \ (gfp\text{-}trans \ T \ P)) = t \ P \ (gfp\text{-}trans \ T \ P) \) by(rule tailcall)
finally show gfp-trans \( T \ P \vdash t \ P \ (gfp\text{-}trans \ T \ P) \).
from \( uP\) gfp-trans-unitary show unitary \( (gfp\text{-}trans \ T \ P) \) by(auto)
qed
show ?Y P ≤ ?X P
proof\(\) (rule le-utrans[simp-all add:assms], simp-all add:assms)
  show le-utrans \( (\lambda. \ gfp\text{-}exp \ (t \ a)) \) \( (T \ (\lambda. \ gfp\text{-}exp \ (t \ a))) \)
proof\(\) (rule le-utransI)
  fix \( Q\):'s expect assume \( uQ\): unitary \( Q\)
  with \( ft\) have \( R. \ unitary \ R \implies unitary \ (t \ Q \ R) \) by(auto)
  with \( mt\) have \( OF \ uQ\) have \( gfp\text{-}exp \ (t \ Q) = t \ Q \ (gfp\text{-}exp \ (t \ Q)) \) by(blast intro: gfp-exp-unfold)
also from \( uQ\) have \( ... = T \ (\lambda. \ gfp\text{-}exp \ (t \ a)) \) \( Q\) by(rule tailcall[symmetric])
finally show \( gfp\text{-}exp \ (t \ Q) \leq T \ (\lambda. \ gfp\text{-}exp \ (t \ a)) \) \( Q\) by(simp)
qed
fix \( Q\):'s expect assume unitary \( Q\)
with \( ft\) have \( R. \ unitary \ R \implies unitary \ (t \ Q \ R) \) by(auto)
thus unitary \( (gfp\text{-}exp \ (t \ Q)) \) by(rule gfp-exp-unitary)
qed
qed

lemma lfp-pulldown:
  fixes \( P\):'s expect and \( t\):'s expect \Rightarrow 's trans
  and \( T\):'s trans \Rightarrow 's trans
assumes tailcall: \( \lambda u P. \ sound \ P \implies T \ u \ P = t \ P \ (u \ P) \)
  and st: \( \lambda P \ Q. \ sound \ P \implies sound \ Q \implies sound \ (t \ P \ Q) \)
  and mt: \( \lambda P. \ sound \ P \implies mono\text{-}trans \ (t \ P) \)
  and monoT: \( \lambda u. \ [ \ le\text{-}trans \ t \ u; \ \lambda P. \ sound \ P \implies sound \ (t \ P); \)
    \( \lambda P. \ sound \ P \implies sound \ (u \ P) \ ] \implies le\text{-}trans \ (T \ t) \ (T \ u) \)
  and aT: \( \lambda t P. \ [ \ \lambda Q. \ sound \ Q \implies sound \ (t \ Q); \ sound \ P \ ] \implies sound \ (T \ t \ P) \)
  and fv: \( le\text{-}trans \ (T \ v) \ v \)
  and sv: \( \lambda P. \ sound \ P \implies sound \ (v \ P) \)
  and sp: \( sound \ P \)
shows lfp-trans \( T \ P = lfp\text{-}exp \ (t \ P) \) \( (\isasymis \ ?X P = ?Y P) \)
proof\(\) (rule antisym)
show ?Y P ≤ ?X P
proof\(\) (rule lfp-exp-lowerbound)
  from \( sp\) have \( t \ P \ (lfp\text{-}trans \ T \ P) = (T \ (lfp\text{-}trans \ T) \ P\) by(rule tailcall[symmetric])
  also have \( (T \ (lfp\text{-}trans \ T) \ P) \leq (lfp\text{-}trans \ T \ P) \)
    by(rule le-utransD[OF lfp-trans-lemma2[OF monoT]], (iprover intro:assms)+)
finally show \( t \ P \ (lfp\text{-}trans \ T \ P) \leq lfp\text{-}trans \ T \ P \).
from \( sp\) show sound \( (lfp\text{-}trans \ T \ P) \)
qed
CHAPTER 3. SEMANTIC STRUCTURES

by (iprover intro: lfp-trans-sound assms)
qed

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

show \( ?X \ P \leq ?Y \ P \)
proof (rule le-transD [OF lfp-trans-lowerbound, OF - sP])
show \( \text{le-trans} (T (\lambda a. \text{lfp-exp} (t \ a))) (\lambda a. \text{lfp-exp} (t \ a)) \)
proof (rule le-transI)
fix \( P::'s \) expect
assume \( \text{fuP} \ \text{svP} \)

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 \( \text{sound} \ P \)
with \( \text{fuP} \ \text{svP} \) show \( \text{sound} (\text{lfp-exp} (t \ P)) \)
by (blast intro: lfp-exp-sound)
qed

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

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

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

end
Chapter 4

The pGCL Language

4.1 A Shallow Embedding of pGCL in HOL

theory Embedding imports Misc Induction begin

4.1.1 Core Primitives and Syntax

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

type-synonym 's prog = bool ⇒ ('s ⇒ real) ⇒ ('s ⇒ real)

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

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

Skip does nothing at all.

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

Apply lifts a state transformer into the space of programs.

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

Seq is sequential composition.

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

PC is probabilistic choice between programs.

definition PC :: 's prog ⇒ ('s ⇒ real) ⇒ 's prog ⇒ 's prog
(- _⊕ - [58,57,57] 57)
where PC a P b ≡ λab Q s. P s * a ab Q s + (1 - P s) * b ab Q s

75
DC is demonic choice between programs.

**Definition:**

\[
\text{DC} :: \text{'s prog } \Rightarrow \text{'s prog } \Rightarrow \text{'s prog } (- \prod - [58,57] 57)
\]

**Where:**

\[
\text{DC } a \ b \equiv \lambda \text{ab } Q \ s . \ min (a \ ab \ Q \ s) (b \ ab \ Q \ s)
\]

AC is angelic choice between programs.

**Definition:**

\[
\text{AC} :: \text{'s prog } \Rightarrow \text{'s prog } \Rightarrow \text{'s prog } (- \bigcup - [58,57] 57)
\]

**Where:**

\[
\text{AC } a \ b \equiv \lambda \text{ab } Q \ s . \ max (a \ ab \ Q \ s) (b \ ab \ Q \ s)
\]

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

**Definition:**

\[
\text{Embed} :: \text{'s trans } \Rightarrow \text{'s prog }
\]

**Where:**

\[
\text{Embed } t = (\lambda \text{ab } t)
\]

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

**Definition:**

\[
\text{Mu} :: (\text{'s prog } \Rightarrow \text{'s prog }) \Rightarrow \text{'s prog }
\]

**Where:**

\[
\text{Mu} (T) \equiv (\lambda \text{ab } \text{if ab then lfp-trans } (\lambda t. T (\text{Embed } t) ab) \ else \ gfp-trans (\lambda t. T (\text{Embed } t) ab))
\]

**Repeat** expresses finite repetition

**Primrec**

\[
\text{repeat} :: \text{nat } \Rightarrow \text{'a prog } \Rightarrow \text{'a prog }
\]

**Where:**

\[
\text{repeat } 0 \ p = \text{Skip } \mid \text{repeat } (\text{Suc } n) \ p = p ;; \text{repeat } n \ p
\]

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

**Definition:**

\[
\text{SetDC} :: (\text{'a } \Rightarrow \text{'s prog }) \Rightarrow (\text{'s } \Rightarrow \text{'a set }) \Rightarrow \text{'s prog }
\]

**Where:**

\[
\text{SetDC } f \ S \equiv \lambda \text{ab } P \ s . \ Inf ((\lambda a. f \ a \ ab \ P \ s) \cdot S \ s)
\]

**Syntax**

\[
\text{-SetDC} :: \text{pttrn } => (\text{'s } => \text{'a set } ) => \text{'s prog } => \text{'s prog } (- \prod - 100)
\]

**Translations**

\[
\prod x \in S . \ p = \text{CONST SetDC } (% x. \ p) \ S
\]

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**

\[
\text{SetPC} :: (\text{'a } \Rightarrow \text{'s prog }) \Rightarrow (\text{'s } \Rightarrow \text{'a } \Rightarrow \text{real } ) \Rightarrow \text{'s prog }
\]

**Where:**

\[
\text{SetPC } f \ p \equiv \lambda \text{ab } P \ s . \ \sum a \in \text{supp } (p \ s) . \ p \ s \ a * f \ a \ ab \ P \ s
\]

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

**Definition**

\[
\text{Bind} :: (\text{'s } \Rightarrow \text{'a } ) \Rightarrow (\text{'a } \Rightarrow \text{'s prog }) \Rightarrow \text{'s prog }
\]
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 : \texttt{pttrn} \mapsto (\texttt{\textquotesingle s} \mapsto \texttt{\textquotesingle a}) \mapsto \texttt{\textquotesingle s prog} \mapsto \texttt{\textquotesingle s prog}

**translations** \texttt{x is f in a} \mapsto \texttt{CONST Bind f (%x a)}

**definition** \texttt{flip} : (\texttt{\textquotesingle a} \Rightarrow \texttt{\textquotesingle b} \Rightarrow \texttt{\textquotesingle c}) \Rightarrow \texttt{\textquotesingle b} \Rightarrow \texttt{\textquotesingle a} \Rightarrow \texttt{\textquotesingle c}

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

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

**syntax** -PBind : \texttt{pttrn} \mapsto (\texttt{\textquotesingle s} \mapsto \texttt{\textquotesingle a}) \mapsto \texttt{\textquotesingle s prog} \mapsto \texttt{\textquotesingle s prog}

**translations** \texttt{bind x at p in a} \mapsto \texttt{CONST SetPC (%x a) (CONST flip (%x p))}

**syntax** -DBind : \texttt{pttrn} \mapsto (\texttt{\textquotesingle s} \mapsto \texttt{\textquotesingle a set}) \Rightarrow \texttt{\textquotesingle s prog} \mapsto \texttt{\textquotesingle s prog}

**translations** \texttt{bind x from S in a} \mapsto \texttt{CONST SetDC (%x a) S}

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

**syntax** -assign : \texttt{ident} \mapsto \texttt{\textquotesingle a} \mapsto \texttt{\textquotesingle s prog} (\texttt{- := } [1000,900]900)

**ML**

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

\]]

**parse-translation**

\[
	ext{fun set-pc-tr - [Const (f,-), P] = }
\text{Const (SetPC, dummyT) }$
\text{Abs (v, dummyT,}
\text{(Const (Embedding.Apply, dummyT) }$
\text{Abs (s, dummyT, Syntax.const (suffix Record.updateN f) }$
\text{Abs (Name.uu-, dummyT, Bound 2) }$	ext{ Bound 0))} )$
\]

\[P\]
| set-pc-tr - ts = raise TERM (set-pc-tr, ts)
These definitions instantiate the embedding as either weakest precondition (True) or weakest liberal precondition (False).

**ML**

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

**Parse-Translation**

```plaintext
\[ \text{parse-translation} \ \| \ [[[@\{\text{syntax-const -SetPC}\}, \ \text{set-pc-tr}] \ \| \]
```

**Syntax**

\[-\text{set-dc} :: \text{ident} \ => \ ('s => 'a set) => 's prog \ (\ - :\in - [66,66]) \]

**Definitions**

\[ \text{wp} :: 's prog \Rightarrow 's trans \]
where
\[ \text{wp pr} \equiv \text{pr True} \]

\[ \text{wlp} :: 's prog \Rightarrow 's trans \]
where
\[ \text{wlp pr} \equiv \text{pr False} \]

If-Then-Else as a degenerate probabilistic choice.

**Abbreviation**

\[ \text{if-then-else} :: ['s \Rightarrow \text{bool}, 's prog, 's prog] \Rightarrow 's prog \]
(If - Then - Else - 58)
where
\[ \text{If P Then a Else b == a \land P \lor b} \]

Syntax for loops

**Abbreviation**

\[ \text{do-while} :: ['s \Rightarrow \text{bool}, 's prog] \Rightarrow 's prog \]
(\text{do - \text{-\text{-\text{-\text{-\text{-}}}} / (4 -) } \ \text{/\text{od})} \]
where
\[ \text{do-while P a \equiv \mu x. If P Then a \ ;; \ x Else \ Skip} \]
4.1. A SHALLOW EMBEDDING OF PGCL IN HOL

4.1.2 Unfolding rules for non-recursive primitives

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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

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

Use these as simp rules to clear out Skips

lemmas skip-simps = Skip-Seq Seq-Skip

end

4.2 Healthiness

theory Healthiness imports Embedding begin

4.2.1 The Healthiness of the Embedding

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

lemma healthy-wp-Abort:
\[
\text{healthy (wp Abort)}
\]
proof(rule healthy-parts)
fix \( b \) and \( P::\alpha \Rightarrow \text{real} \)
assume \( nP::\text{nneg P and bP: bounded-by b P} \)
thus \( \text{bounded-by b (wp Abort P)} \)
unfolding wp-eval by(blast)
show \( \text{nneg (wp Abort P)} \)
unfolding wp-eval by(blast)
next
fix \( P \) and \( Q::\alpha \text{ expect} \)
show \( \text{wp Abort P \vdash wp Abort Q} \)
unfolding wp-eval by(blast)
next
fix \( P \) and \( c \) and \( s::\alpha \)
show \( c * \text{ wp Abort P s} = \text{ wp Abort (\lambda s. c * P s) s} \)
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 ($wlp \ t$) and $hu$: nearly-healthy ($wlp \ u$)
shows nearly-healthy ($wlp \ (t :: u)$)
proof(rule nearly-healthyI, simp-all add:wp-eval)
fix $b$ and $P$: $\Rightarrow \Rightarrow$ real
assume unitary $P$
with $hu$ have unitary ($wlp \ u \ P$) by(auto)
with $ht$ show unitary ($wlp \ t \ (wlp \ u \ P)$) by(auto)
next
fix $P \ Q$: $\Rightarrow \Rightarrow$ real
assume unitary $P$ and unitary $Q$ and $P \vdash Q$
with $hu$ have unitary ($wlp \ u \ P$) and unitary ($wlp \ u \ Q$)
and $wlp \ u \ P \vdash wlp \ u \ Q$ by(auto)
with $ht$ show $wlp \ t \ (wlp \ u \ P) \vdash wlp \ t \ (wlp \ u \ Q)$ by(auto)
qed

lemma healthy-wp-PC:
fixes $f$: $\Rightarrow$ prog
assumes $hf$: healthy ($wp \ f$) and $hg$: healthy ($wp \ g$)
and $uP$: unitary $P$
shows healthy ($wp \ (f \oplus g)$)
proof(intro healthy-parts bounded-byI nnegI le-funI, simp-all add:wp-eval)
fix $b$ and $Q$: $\Rightarrow \Rightarrow$ real and $s$: $\Rightarrow$
assume $nQ$: $\neg\neg Q$ and $bQ$: bounded-by $b \ Q$
Non-negative:
from $nQ$ and $bQ$ and $hf$ have $0 \leq wp \ f \ Q \ s$ by(auto)
with $uP$ have $0 \leq P \ s \ * \ \ldots$ by(auto intro:mult-nonneg-nonneg)
moreover {
from $uP$ have $0 \leq 1 - P \ s$ by(auto simp:sign-simps)
with $nQ$ and $bQ$ and $hg$ have $\ldots \ le \ \ldots \ * \ wp \ g \ Q \ s$
by(auto intro:mult-nonneg-nonneg)
}
ultimately show $0 \leq P \ s \ * \ wp \ f \ Q \ s + (1 - P \ s) * wp \ g \ Q \ s$
by(auto intro:mult-nonneg-nonneg)

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

next

Monotonic:

fix \( Q \Rarrow \bullet \)\( s \Rightarrow \) real and \( s \)
assume \( \text{sound } Q \) and \( \text{sound } R \) and \( \text{lc} : Q \vdash R \)

with \( \text{lf} \) have \( \text{wp } f \quad Q \cdot s \leq \text{wp } f \quad R \cdot s \) by (\text{dest:mono-transD})
with \( \text{uf} \) have \( P \cdot s \ast \text{wp } f \quad Q \cdot s \leq P \cdot s \ast \text{wp } f \quad R \cdot s \)
by (\text{intro:mult-left-mono})
moreover {
from \( \text{sf } Q \cdot s \Rightarrow \text{lc } R \)
have \( \text{wp } g \quad Q \cdot s \leq \text{wp } g \quad R \cdot s \) by (\text{dest:mono-transD})
moreover from \( \text{uf} \) have \( 0 \leq 1 - P \cdot s \) by (\text{simp:sign-simps})
ultimately have \((1 - P \cdot s) \ast \text{wp } g \quad Q \cdot s \leq (1 - P \cdot s) \ast \text{wp } g \quad R \cdot s \)
by (\text{intro:mult-left-mono})
}
ultimately show \( P \cdot s \ast \text{wp } f \quad Q \cdot s + (1 - P \cdot s) \ast \text{wp } g \quad Q \cdot s \leq P \cdot s \ast \text{wp } f \quad R \cdot s + (1 - P \cdot s) \ast \text{wp } g \quad R \cdot s \) by (\text{auto})
next

Scaling:

fix \( Q \Rarrow \bullet \)\( s \Rightarrow \) real and \( c : \text{real} \) and \( s \)
assume \( \text{sound } Q \) and \( \text{pos} \): \( 0 \leq c \)
have \( c \ast (P \cdot s \ast \text{wp } f \quad Q \cdot s + (1 - P \cdot s) \ast \text{wp } g \quad Q \cdot s) = P \cdot s \ast (c \ast \text{wp } f \quad Q \cdot s) + (1 - P \cdot s) \ast (c \ast \text{wp } g \quad Q \cdot s) \)
by (\text{simp add:distrib-left})
also have \( \ldots \ast = P \cdot s \ast \text{wp } f (\lambda s. c \ast Q \cdot s) \ast + (1 - P \cdot s) \ast \text{wp } g (\lambda s. c \ast Q \cdot s) \ast \ast 
using \( \text{lf } Q \cdot s \Rightarrow \text{lc } Q \cdot s \)
by (\text{simp add:scalingD(OF healthy-scalingD)})
finally show \( c \ast (P \cdot s \ast \text{wp } f \quad Q \cdot s + (1 - P \cdot s) \ast \text{wp } g \quad Q \cdot s) = P \cdot s \ast \text{wp } f (\lambda s. c \ast Q \cdot s) \ast + (1 - P \cdot s) \ast \text{wp } g (\lambda s. c \ast Q \cdot s) \ast \ast 
qed

lemma nearly-healthy-wlp-PC:
fixes \( f \Rarrow \bullet \)\( s \Rightarrow \) prog
assumes \( \text{lf}: \) nearly-healthy \( (\text{wlp } f) \) and \( \text{lh}: \) nearly-healthy \( (\text{wlp } g) \) and \( \text{uf}: \) unitary \( P \)
shows nearly-healthy \( (\text{wlp } (f \oplus g)) \)
proof (\text{intro nearly-healthyI unitaryI2 nnegI bounded-byI lc-funI, simp-all add:wp-eval})
fix \( Q \Rarrow \bullet \)\( s \Rightarrow \) expect and \( s \)
assume \( \text{uf}: \) unitary \( Q \)
from \( \text{uf } Q \Rightarrow \text{lf } Q \) have \( \text{uf}: \) unitary \( (\text{wlp } f \quad Q) \) unitary \( (\text{wlp } g \quad Q) \) by (\text{auto})
from \( \text{uf} \) have \( \text{nnP}: \) \( 0 \leq P \cdot s \ast 0 \leq 1 - P \cdot s \) by (\text{simp:sign-simps})
moreover from \( \text{uf } Q \) have \( 0 \leq \text{wp } f \quad Q \cdot s 0 \leq \text{wp } g \quad Q \cdot s \) by (\text{auto})
ultimately show \( 0 \leq P \cdot s \ast \text{wp } f \quad Q \cdot s + (1 - P \cdot s) \ast \text{wp } g \quad Q \cdot s \)
4.2. HEALTHINESS

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

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

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

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

with \(hf\) have bounded-by \(b\) \((\wp f P)\) by(auto)
hence \(\wp f P s \leq b\) by(blast)
thus \(\min (\wp f P s) (\wp g P s) \leq b\) by(auto)

from \(nP\) \(bP\) assms show \(0 \leq \min (\wp f P s) (\wp g P s)\) by(auto)
next
fix \(P\):’s \(\Rightarrow\) real and \(Q\) and \(s\):’s
from assms have \(mf\): mono-trans \((\wp f)\) and \(mg\): mono-trans \((\wp g)\) by(auto)
assume \(sP\): sound \(P\) and \(sQ\): sound \(Q\) and \(le\): \(P \vdash Q\)
hence \(\wp f P s \leq \wp f Q s\) and \(\wp g P s \leq \wp g Q s\)
    by(auto intro:le-funD[OF mono-transD[OF \(mf\)] le-funD[OF mono-transD[OF \(mg\)]]])
thus \(\min (\wp f P s) (\wp g P s) \leq \min (\wp f Q s) (\wp g Q s)\) by(auto)
next
fix \(P\):’s \(\Rightarrow\) real and \(c\):real and \(s\):’s
assume \(sP\): sound \(P\) and \(pos\): \(0 \leq c\)
lemma nearly-healthy-wlp-DC:
 fixes 
 assumes \( hf \): nearly-healthy \((wlp f)\)
 and \( hg \): nearly-healthy \((wlp g)\)
 shows nearly-healthy \((wlp (f \sqcap g))\)

proof
 intro nearly-healthyI bounded-byI nnegI le-funI unitaryI2,
 simp-all add:wp-eval, safe
 fix \( P \):'s ⇒ real and \( s \):'s
 assume \( uP \): unitary \( P \)
 with \( hf \) \( hg \) have \( atP \): unitary \((wlp f P)\) unitary \((wlp g P)\) by(auto)
 thus \( 0 \leq wlp f P s 0 \leq wlp g P s \) by(auto)

have \( min (wlp f P s) (wlp g P s) \leq wlp f P s \) by(auto)
 also from \( atP \) have \( ... \leq 1 \) by(auto)
 finally show \( min (wlp f P s) (wlp g P s) \leq 1 \) .

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

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

qed

lemma healthy-wp-AC:
 fixes 
 assumes \( hf \): healthy \((wp f)\) and \( hg \): healthy \((wp g)\)
 shows healthy \((wp (f \sqcap g))\)

proof
 intro healthy-parts bounded-byI nnegI le-funI, simp-all only:wp-eval
 fix \( b \) and \( P \):'s ⇒ real and \( s \):'s
 assume \( nP \): \( nneg P \) and \( bP \): bounded-by \( b \) \( P \)
 with \( hf \) have bounded-by \( b \) \((wp f P)\) by(auto)
4.2. HEALTHINESS

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

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

lemma nearly-healthy-wlp-AC:
  fixes \( f::'s \Rightarrow \text{prog} \)
  assumes \( \text{hf: nearly-healthy} (wp f) \)
    and \( \text{hg: nearly-healthy} (wp g) \)
  shows \( \text{nearly-healthy} (wp (f \parallel g)) \)
proof(intro nearly-healthyI bounded-byI nnegI unitaryI2 le-funI, simp-all only:wp-eval)
  fix \( b \) and \( P::'s \Rightarrow real \) and \( s::'s \)
  assume \( uP: \text{unitary} P \)
with \( \text{hf} \) have \( wp \ f \ P \ s \leq 1 \) 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 \( uP \ hg \) have \( \text{unitary} (wp f P) \) by(auto)
  hence \( 0 \leq wp \ f \ P \ s \) by(auto)
thus \( 0 \leq \max (wp \ f \ P \ s) (wp \ g \ P \ s) \) by(auto)
next
\textbf{CHAPTER 4. THE PGCL LANGUAGE}

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

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

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

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

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

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

by (intro cInf-lower bdd-below1, auto)

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

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

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

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

next

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

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

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

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

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

moreover

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

proof (rule ballI, clarify, rule bexI)

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

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

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

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

qed

ultimately

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

by (intro cInf-mono, blast+)

next

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

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

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

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

proof (rule antisym)

show \( ?U \leq \ ?V \)

proof (rule cInf-greatest)

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

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

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

* \( y \) by (auto)

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

proof (intro cInf-lower[OF yin] bdd-below1)

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

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

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

qed

with \( \text{pos rwx} \) show \( c * \inf ((\lambda. \ wp \ (\text{prog a}) \ P \ s) \cdot S \ s) \leq x \) by (auto)

intro: mult-left-mono)

qed

show \( ?V \leq ?U \)

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

lemma nearly-healthy-wlp-SetDC:
  fixes \( \text{prog} : \text{'a prog} \) \( \text{and} \ S : \text{a set} \)
  assumes healthy: \( \forall x. x \in S \Longrightarrow \text{nearly-healthy} (\text{wp} \ (\text{prog} x)) \)
  and nonempty: \( \forall s. \exists \ x. x \in S \) 
  shows nearly-healthy \( (\text{wp} \ (\text{SetDC} \ \text{prog} S)) \) \( (\text{is} \ \text{nearly-healthy} \ ?T) \)
proof (intro nearly-healthyI unitaryI2 bounded-byI negI le-funI, simp-all:wp-eval)
fix \(b\) and \(\text{P}::'a \Rightarrow \text{real} \; \text{and} \; s::'a\)
assume \(\text{uP}: \text{unitary} \; P\)

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

bdd_below1, \text{auto})
also from \(\text{xin healthy uP have } x \leq 1 \) \text{ by(blast)}
finally show \(\text{Inf} \; ((\lambda a. \text{wlp} \; \text{prog} \; a) \; P \) \(\cdot S \; s\) \(\leq 1 \).

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

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

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

lemma \text{healthy-wp-SetPC}:
  fixes \(p::s\) \(\Rightarrow \; 'a \Rightarrow \text{real}\)
  and \(f::'a \Rightarrow \; 's \; \text{prog}\)
  assumes \text{healthy: } \lambda a. s. a \in \text{supp} \; (p \; s) \Longrightarrow \text{healthy (wp (f a))}
  and \text{sound}: \lambda s. \text{sound (p s)}
  and \text{sub-dist: } \lambda s. (\sum a \in \text{supp (p s)} \cdot p \; s \; a) \leq 1
shows healthy (wp (SetPC f p)) (is healthy ?X)

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

fix b and P:’s ⇒ real and s:’s

assume bP: bounded-by b P and nP: nneg P

hence sP: sound P by(auto)

from sP and bP and healthy have \( \forall a. a \in \text{supp} (p s) \implies \text{wp} (f a) P s \leq b \)

by(blast dest:healthy-bounded-byD)

with sound have \( ( \sum a \in \text{supp} (p s). p s a * wp (f a) P s) \leq ( \sum a \in \text{supp} (p s). p s a) * b \)

by(simp add:setsum-left-distrib)

also have \( \ldots = ( \sum a \in \text{supp} (p s). p s a) * b \)

also { from bP and nP have \( 0 \leq b \) by(blast)

with sub-dist have \( ( \sum a \in \text{supp} (p s). p s a) * b \leq 1 * b \)

by(rule mult-right-mono) }

also have \( 1 * b = b \) by(simp)

finally show \( ( \sum a \in \text{supp} (p s). p s a * wp (f a) P s) \leq b . \)

show \( 0 \leq ( \sum a \in \text{supp} (p s). p s a * wp (f a) P s) \)

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

fix x

from sound show \( 0 \leq p s x \) by(blast)

assume x ∈ supp (p s) with sP and healthy

show \( 0 \leq \text{wp} (f x) P s \) by(blast)

qed

next

fix P:’s ⇒ real and Q:’s ⇒ real and s

assume s-P: sound P and s-Q: sound Q and ent: P ⊢ Q

with healthy have \( \forall a. a \in \text{supp} (p s) \implies \text{wp} (f a) P s \leq \text{wp} (f a) Q s \)

by(blast)

with sound show \( ( \sum a \in \text{supp} (p s). p s a * wp (f a) P s) \leq ( \sum a \in \text{supp} (p s). p s a) * wp (f a) Q s) \)

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

next

fix P:’s ⇒ real and c:real and s:’s

assume sound: sound P and pos: 0 ≤ c

have c * ( \( \sum a \in \text{supp} (p s). p s a * wp (f a) P s) = ( \sum a \in \text{supp} (p s). p s a) * (c * wp (f a) P s) \)

(is ?A = ?B)

by(simp add:setsum-right-distrib ac-simps)

also from sound and pos and healthy

have \( \ldots = ( \sum a \in \text{supp} (p s). p s a * wp (f a) (\lambda s. c * P s)) \)

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

finally show ?A = ... .

qed
4.2. HEALTHINESS

lemma nearly-healthy-wlp-SetPC:
  fixes p::s ⇒ 'a ⇒ real
  and f::'a ⇒ 's prog
  assumes healthy: ∀ a s. a ∈ supp (p s) ⇒ nearly-healthy (wlp (f a))
  and sound: ∀ s. sound (p s)
  and sub-dist: ∀ s. (∑ a∈supp (p s). p s a) ≤ 1
  shows nearly-healthy (wlp (SetPC f p)) (is nearly-healthy ?X)
proof
  (intro nearly-healthyI unitaryI2 bounded-byI nnegI le-funI, simp-all:wp-eval)
  fix b and P::'s ⇒ real and s::'s
  assume uP: unitary P
  from uP healthy have ∀ a s. a ∈ supp (p s) ⇒ unitary (wlp (f a) P)
  hence ∀ a. a ∈ supp (p s) ⇒ wlp (f a) P s ≤ 1 by(auto)
  with sound have (∑ a∈supp (p s). p s a * wlp (f a) P s) ≤ (∑ a∈supp (p s). p s a * 1)
    by(blast intro:setsum-mono mult-left mono)
  also have ... = (∑ a∈supp (p s). p s a)
    by(simp add:setsum-left-distrib)
  also note sub-dist
  finally show (∑ a∈supp (p s). p s a * wlp (f a) P s) ≤ 1 .
  show 0 ≤ (∑ a∈supp (p s). p s a * wlp (f a) P s)
  proof(rule setsum-nonneg, clarify, rule mult-nonneg nonneg)
  fix x
    from sound show 0 ≤ p s x by(blast)
  assume x ∈ supp (p s) with uP healthy
  show 0 ≤ wlp (f x) P s by(blast)
qed

next
  fix P::'s expect and Q::'s expect and s
  assume uP: unitary P and uQ: unitary Q and le: P ⊢ Q
  hence ∀ a. a ∈ supp (p s) ⇒ wlp (f a) P s ≤ wlp (f a) Q s
    by(blast intro: le-funD[OF nearly-healthy monoD, OF healthyI])
  with sound show (∑ a∈supp (p s). p s a * wlp (f a) P s) ≤
    (∑ a∈supp (p s). p s a * wlp (f a) Q s)
    by(blast intro:setsum-mono mult-left mono)
qed

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

lemma nearly-healthy-wlp-Apply:
  nearly-healthy (wlp (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 hs: healthy (wp (f s))
  shows healthy (wp (Bind f p))
CHAPTER 4. THE PGCL LANGUAGE

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 \(b\) \((wp\ (p\ (f\ s))\) \(P\) \(b\)) by(auto)
thus \(wp\ (p\ (f\ s))\) \(P\) \(s\) \(\leq\) \(b\) by(auto)
from \(bP\ nP\) hsub have \(nneg\ (wp\ (p\ (f\ s))\) \(P\) \(b\)) by(auto)
thus \(0\ \leq\ wp\ (p\ (f\ s))\) \(P\) \(s\) by(auto)

next
fix \(P\) \(Q::'s expect\) and \(s::'s\)
assume sound \(P\) sound \(Q\) \(P\) \(\vdash\) \(Q\)
thus \(0\ \leq\ wp\ (p\ (f\ s))\) \(P\) \(s\) \(\leq\) \(1\) by(rule le-funD[OF mono-transD, OF healthy-monoD, OF hsub])

next
fix \(P::'s expect\) and \(c::\text{real}\) and \(s::'s\)
assume sound \(P\) and \(0\ \leq\ c\)
thus \((\lambda s. c * P\ s)\) \(s\) = \((wp\ (p\ (f\ s)))\) \(s\)
by(simp add:scalingD[OF healthy-scalingD, OF hsub])

qed

lemma nearly-healthy-wlp-Bind:
fixes \(f::'s\) \(\Rightarrow::'a\)
assumes hsub: \(\lambda s.\) nearly-healthy \((wp\ (p\ (f\ s)))\)
shows nearly-healthy \((wp\ (\text{Bind}\ f\ p))\)

proof\(\) (intro nearly-healthyI unitaryI2 nnegI bounded-byI le-funI, simp-all only:wp-eval)
fix \(P::'s expect\) and \(s::'s\)
assume unitary \(P\)
with hsub have unitary \((wp\ (p\ (f\ s)))\) \(P:\)
by(lemma le-funD[OF nearly-healthy-monoD, OF hsub])

qed

4.2.2 Healthiness for Loops

lemma wp-loop-step-mono:
fixes \(t::u::'s\) \(\Rightarrow\)
assumes hb: \(\) healthy \((wp\ \text{body})\)
and le: \(\) le-trans \(t::u::\)
and ht: \(\) \(\lambda P.\) sound \(P\) \(\Rightarrow\) sound \((t\ P)\)
and hu: \(\) \(\lambda P.\) sound \(P\) \(\Rightarrow\) sound \((u\ P)\)
shows le-trans \((wp\ (\text{body}::\text{Embed}\ t::G::\text{Skip}))\)
\((wp\ (\text{body}::\text{Embed}\ u::G::\text{Skip}))\)

proof\(\) (intro le-transI le-funI, simp add:wp-eval)
fix \(P::'s expect\) and \(s::'s\)
assume sp: sound \(P\)
with le have \(\) \(t\ P\) \(\vdash\) \(u\ P\) by(auto)
moreover from sp ht hu have sound \((t\ P)\) sound \((u\ P)\) by(auto)

qed
ultimately have \( \text{wp body } (t \ P) \ s \leq \text{wp body } (u \ P) \ s \)
by(auto intro:le-funD[OF mono-transD, OF healthy-monoD, OF hb])
thus \( \langle \text{G} \rangle \ s \ast \text{wp body } (t \ P) \ s \leq \langle \text{G} \rangle \ s \ast \text{wp body } (u \ P) \ s \)
by(auto intro:mult-left-mono)
qed

**Lemma** \( \text{lfp-loop-step-mono} \):
\begin{align*}
\text{fixes } & t \ u :: 's \ trans \\
\text{assumes } & mb: \text{nearly-healthy } (\text{wp body}) \\
\text{and } & \text{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 } & \text{le-trans } (\text{wp body } :: \text{Embed } t \langle \text{G} \rangle \oplus \text{Skip}) \\
& (\text{wp body } :: \text{Embed } u \langle \text{G} \rangle \oplus \text{Skip})
\end{align*}

**Proof**
\begin{align*}
\text{fix } & P :: 's \ \text{expect} \\
\text{fix } & s :: 's
\end{align*}

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

**Lemma** \( \text{lfp-loop-fp} \):
\begin{align*}
\text{assumes } & hb: \text{healthy } (\text{wp body}) \\
\text{and } & sP: \text{sound } P \\
\text{shows } & \lambda s. \langle \text{G} \rangle \ s \ast \text{wp body } (\lambda s. \text{bound-of } P) \ s + \langle \text{N} \text{G} \rangle \ s \ast P \ s \vdash \lambda s. \text{bound-of } P
\end{align*}

**Proof**
\begin{align*}
\text{fix } & s \\
\text{from } & sP \ \text{have } \text{sound } (\lambda s. \text{bound-of } P) \ \text{by(auto)} \\
\text{moreover hence } & \text{bounded-by } (\text{bound-of } P) (\lambda s. \text{bound-of } P) \text{ by(auto)} \\
\text{ultimately have } & \text{bounded-by } (\text{bound-of } P) (\text{wp body } (\lambda s. \text{bound-of } P)) \\
\text{using } & hb \ \text{by(auto)} \\
\text{hence } & \text{wp 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 } & \langle \text{G} \rangle \ s \ast \text{wp body } (\lambda a. \text{bound-of } P) \ s + (1 - \langle \text{G} \rangle \ s) \ast P \ s \leq \langle \text{G} \rangle \ s \ast \text{bound-of } P + (1 - \langle \text{G} \rangle \ s) \ast \text{bound-of } P \\
\text{by } & (\text{blast intro:add-mono mult-left-mono}) \\
\text{thus } & \langle \text{G} \rangle \ s \ast \text{wp body } (\lambda a. \text{bound-of } P) \ s + \langle \text{N} \text{G} \rangle \ s \ast P \ s \leq \text{bound-of } P \\
\text{by } & (\text{simp add:algebra-simps negate-embed})
\end{align*}

qed

**Lemma** \( \text{lfp-loop-greatest} \):
\begin{align*}
\text{fixes } & P :: 's \ \text{expect}
\end{align*}
\textbf{assumes} lb: \( \forall R. \; \lambda s. \; ('G' \; s \; \ast \; \text{wp body} \; R \; s + \; 'N' \; G' \; s \; \ast \; P \; s \vdash R \implies \text{sound} \; R \)
\( \implies Q \vdash R \)
\; and \; hb: \text{healthy} \; (\text{wp body})
\; and \; sP: \text{sound} \; P
\; and \; sQ: \text{sound} \; Q
\textbf{shows} \( Q \vdash \text{lfp-exp} \; (\lambda Q. \; ('G' \; s \; \ast \; \text{wp body} \; Q \; s + \; 'N' \; G' \; s \; \ast \; P \; s)) \)
\textbf{using} \text{by}(\text{auto intro!:\text{lfp-exp-greatest} \{OF lb sQ\} sP \text{lfp-loop-fp} \; hb)\text{lemma lfp-loop-sound:}\text{fixes} \; \text{t #:}'s \; \text{expect}
\textbf{assumes} hb: \text{healthy} \; (\text{wp body})
\; and \; sP: \text{sound} \; P
\textbf{shows} \text{sound} \; (\text{lfp-exp} \; (\lambda Q. \; ('G' \; s \; \ast \; \text{wp body} \; Q \; s + \; 'N' \; G' \; s \; \ast \; P \; s)) \)
\textbf{using} \text{assms by}(\text{auto intro!:\text{lfp-exp-sound} \text{lfp-loop-fp})\text{lemma wlp-loop-step-unitary:}\text{fixes} \; t #:}'s \; \text{trans}
\textbf{assumes} hb: \text{nearly-healthy} \; (\text{wlp body})
\; and \; \text{ht: } \; \exists P. \; \text{unitary} \; P \implies \text{unitary} \; (t \; P)
\; and \; uP: \text{unitary} \; P
\textbf{shows} \text{unitary} \; (\text{wlp body} \; (\text{body} \; t \; _\ast \; G \; \oplus \; \text{Skip} \; P) \)
\textbf{proof}(\text{intro unitaryI2 \; \text{nexpI} \; \text{bounded-byI}, \; \text{simp-all add:wp-eval})
\text{fix} \; s::'s
\text{from} \; \text{ht} \; uP \; \text{have} \; uP: \text{unitary} \; (t \; P) \; \text{by}(\text{auto})
\text{with} \; \text{hb have} \; \text{unitary} \; (\text{wp body} \; (t \; P)) \; \text{by}(\text{auto})
\text{hence} \; 0 \leq \; \text{wlp body} \; (t \; P) \; s \; \text{by}(\text{auto})
\text{with} \; \text{uP show} \; 0 \leq \; ('G' \; s \; \ast \; \text{wp body} \; (t \; P) \; s + \; (1 - \; 'G' \; s) \; \ast \; P \; s
\; \text{by}(\text{auto intro!:add-nonneg-nonneg} \; \text{mult-nonneg-nonneg})
\text{from} \; \text{ht} \; uP \; \text{have} \; \text{bounded-by} \; 1 \; (t \; P) \; \text{by}(\text{auto})
\text{with} \; \text{uP \; hb have} \; \text{bounded-by} \; 1 \; (\text{wp body} \; (t \; P)) \; \text{by}(\text{auto})
\text{hence} \; \text{wlp body} \; (t \; P) \; s \leq 1 \; \text{by}(\text{auto})
\text{with} \; \text{uP have} \; ('G' \; s \; \ast \; \text{wp body} \; (t \; P) \; s + \; (1 - \; 'G' \; s) \; \ast \; P \; s \leq ('G' \; s \; \ast \; 1 + (1 - ('G' \; s) \; \ast \; 1
\; \text{by}(\text{blast intro!add-mono \; \text{mult-left-mono})
\text{also have } \ldots = 1 \; \text{by}(\text{simpl})
\text{finally show} \; ('G' \; s \; \ast \; \text{wp body} \; (t \; P) \; s + \; (1 - ('G' \; s) \; \ast \; P \; s \leq 1 \; .
\textbf{qed}

\textbf{lemma wp-loop-step-sound:}\text{fixes} \; t #:}'s \; \text{trans}
\textbf{assumes} hb: \text{healthy} \; (\text{wp body})
\; and \; \text{ht: } \; \exists P. \; \text{sound} \; P \implies \text{sound} \; (t \; P)
\; and \; sP: \text{sound} \; P
\textbf{shows} \text{sound} \; (\text{wp body} \; (\text{body} \; t \; _\ast \; G \; \oplus \; \text{Skip} \; P) \)
\textbf{proof}(\text{intro soundI2 \; \text{nexpI} \; \text{bounded-byI}, \; \text{simp-all add:wp-eval})
\text{fix} \; s::'s
\text{from} \; \text{ht \; sP have} \; stP: \text{sound} \; (t \; P) \; \text{by}(\text{auto})
\text{with} \; \text{hb have} \; 0 \leq \; \text{wp body} \; (t \; P) \; s \; \text{by}(\text{auto})
\text{with} \; \text{sP show} \; 0 \leq \; ('G' \; s \; \ast \; \text{wp body} \; (t \; P) \; s + \; (1 - ('G' \; s) \; \ast \; P \; s
by(auto intro: add-nonneg-nonneg mult-nonneg-nonneg)

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

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

**lemma wlp-Loop1**: fixes body :: 's prog
assumes unitary: unitary P
and healthy: nearly-healthy (wlp body)
shows wlp (do G -> body od) P =
gfp-exp (λQ s. «G» s * wlp body Q s + «N G» s * P s)
is ?X = gfp-exp (?Y P)
proof —
let ?Z u = (body ;; Embed u ` G ⊕ Skip)
show ?thesis
proof(unfold wp-eval, intro gfp-pulldown assms le-funI)
  fix u P
  show wlp (?Z u) P = ?Y P (u P) by(simp add: wp-eval negate-embed)
next
  fix t::'s trans and P::'s expect
  assume at: ∀Q. unitary Q ==> unitary (t Q) and uP: unitary P
  thus unitary (wlp (?Z t) P)
    by(rule wlp-loop-step-unitary[OF healthy])
next
  fix P Q::'s expect
  assume uP: unitary P and uQ: unitary Q
  show unitary (λa. « G » a * wlp body Q a + « N G » a * P a)
proof(intro unitaryI2 nnegI bounded-byI)
  fix s::'
  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 + (1 − «G» s) * P s ≤ 1
by(auto intro:mult-left-mono)
qed

next
fix P Q R::′ s expect and s::′ s
assume uP: unitary P and uQ: unitary Q and uR: unitary R
and le: Q ⊢ R
hence wlp body Q s ≤ wlp body R s
by(blast intro:le-funD[OF nearly-healthy-monoD, OF healthy])
thus «G» s * wlp body Q s + «N G» s * P s ≤
«G» s * wlp body R s + «N G» s * P s
by(auto intro:mult-left-mono)
next
fix t u::′ s trans
assume le-utrans t u
∀ P. unitary P ⇒ unitary (t P)
∀ P. unitary P ⇒ unitary (u P)
thus le-utrans (wlp (λZ t)) (wlp (λZ u))
by(blast intro: wp-loop-step-mono[OF healthy])
qed

lemma wp-loop-sound:
assumes sP: sound P
and hb: healthy (wp body)
shows sound (wp do G −→ body od P)
unfolding wp-eval
proof(intro lfp-trans-sound sP)
let ?v = λP s. bound-of P
show le-trans (wp (body ;; Embed ?v ∘ G ;; Skip)) ?v
by(intro le-transI, simp add: wp-eval lfp-loop-fp unfolded negate-embed hb)
show ∀ P. sound P ⇒ sound (? v P) by(auto)
qed

Likewise, we can rewrite strict loops.

lemma wp-Loop1:
fixes body :: ′ s prog
assumes sP: sound P
and healthy: healthy (wp body)
shows wp (do G −→ body od) P =
lfp-exp (λQ s. «G» s * wp body Q s + «N G» s * P s)
4.2. HEALTHINESS

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

from sP have 0 ≤ P s by(auto)
moreover from sQ healthy have 0 ≤ wp body Q s by(auto)
ultimately show 0 ≤ «G» s * wp body Q s + «N G» s * P s
    by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)
qed
next
fix P Q R::'s expect and s::'s
assume sQ: sound Q and sR: sound R
and le: Q ⊇ R
hence wp body Q s ≤ wp body R s
    by(blast intro:le-funD[OF mono-transD, OF healthy-monoD, OF healthy])
thus «G» s * wp body Q s + «N G» s * P s ≤ «G» s * wp body R s + «N G» s * P s
    by(auto intro:mult-left-mono)
next
fix t u:’s trans
assume le: le-trans t u
and st: \( \forall P, \text{sound} P \implies \text{sound} (t P) \)
and su: \( \forall P, \text{sound} P \implies \text{sound} (u P) \)
with healthy show le-trans (wp (\(?Z t\)) (wp (\(?Z u\)))
by(rule wp-loop-step-mono)
next
from healthy show le-trans (wp (\(?Z (\lambda P.s. \text{bound-of} P)\)) (\(\lambda P. \text{bound-of} P\))
by(intro le-transI, simp add:wp-eval lfp-loop-fp[unfolded negate-embed])
next
fix P::’s expect and s::’s
assume sound P
thus sound (\(\lambda s. \text{bound-of} P\)) by(auto)
qed
qed

lemma nearly-healthy-wlp-loop:
fixes body::’s prog
assumes hb: nearly-healthy (wlp body)
shows nearly-healthy (wlp (do G \rightarrow 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 \(?X R = \lambda Q s. \langle G \rangle s \ast \text{wlp body} Q s + \langle N G \rangle s \ast R s\)
show \(\lambda s. 0 \vdash \text{gfp-exp} (?X P)\)
proof(rule gfp-exp-upperbound)
  show unitary (\(\lambda s. 0::\text{real}\)) by(auto)
  with hb have unitary (wlp body (\(\lambda s. 0\))) by(auto)
  with uP show \(\lambda s. 0 \vdash (?X P (\lambda s. 0))\)
  by(blast intro!:le-funI add-nonneg-nonneg mult-nonneg-nonneg)
qed

show \text{gfp-exp} (?X P) \vdash \lambda s. 1
proof(rule gfp-exp-least)
  show unitary (\(\lambda s. 1::\text{real}\)) by(auto)
  fix Q::’s expect
  assume uQ: 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 \text{gfp-exp} (?X P) \vdash \text{gfp-exp} (?X Q)
proof(rule gfp-exp-least)
  fix R::’s expect assume uR: unitary R
  assume fp: R \vdash (?X P R)
  also from le have ... \vdash (?X Q R)
4.2. HEALTHINESS

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

**Lemma healthy-wp-loop:**

- **Fixes** body: ’s prog
- **Assumes** hb: healthy (wp body)
- **Shows** healthy (wp (do G → body od))

**Proof**

- **Let** ?X P = (λQ s. "G" s * wp body Q s + «N G» s * P s)
- **Show** ?thesis
  - **Proof**(intro healthy-parts bounded-byI2 nnegI2, simp-all add:wp-Loop1 hb soundI2 sound-intros)
    - **Fix** P: ’s expect and c::real and s: ’s
    - **Assume** sP: sound P and nnc: 0 ≤ c
    - **Show** c * (lfp-exp (?X P)) s = lfp-exp (?X (λs. c * P s)) s
      - **Proof**(cases)
        - **Assume** c = 0 thus ?thesis
          - **Proof**(simp, intro antisym)
            - **From** hb have fp: λs. "G" s * wp body (λ-. 0) s ⊨ λs. 0 by(simp)
              - **Hence** lfp-exp (λP s. "G" s * wp body P s) ⊨ λs. 0
                - by(auto intro:lfp-exp-lowerbound)
            - **Thus** lfp-exp (λP s. "G" s * wp body P s) s ≤ 0 by(auto)
              - **Have** λs. 0 ⊨ lfp-exp (λP s. "G" s * wp body P s)
                - by(auto intro:lfp-exp-greatest fp)
            - **Thus** 0 ≤ lfp-exp (λP s. "G" s * wp body P s) s by(auto)
                - **Qed**
              - **Next**
                - **Have** onesided: \( P. c, c \neq 0 \implies 0 \leq c \implies sound P \implies \lambda a. c * lfp-exp (\lambda a. "G" b * wp body a b + «N G» b * P b) a \vdash \)

CHAPTER 4. THE PGCL LANGUAGE

\[ \text{lfp-exp} (\lambda a \ b. \ «G» \ b \ * \ \text{wp body} \ a \ b + \ «N» \ G \ b \ * \ (c \ * \ P \ b)) \]

\text{proof} –

\text{fix } P \text{:: expect and } c:: \text{real}

\text{assume } cnz: \ c \neq 0 \ \text{and } nnc: \ 0 \leq c \ \text{and } sP; \ \text{sound } P

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

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

\text{show } \lambda a. \ c \ * \ \text{lfp-exp} (\lambda a \ b. \ «G» \ b \ * \ \text{wp body} \ a \ b + \ «N» \ G \ b \ * \ P \ b) \ a \vdash

\text{lfp-exp} (\lambda a \ b. \ «G» \ b \ * \ \text{wp body} \ a \ b + \ «N» \ G \ b \ * \ (c \ * \ P \ b))

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

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

\text{assume } Q:: \ \text{sound } Q

\quad \text{and } \text{fp: } \lambda b. \ «G» \ b \ * \ \text{wp body} \ Q \ b + \ «N» \ G \ b \ * \ (c \ * \ P \ b) \vdash Q

\text{hence } s \text{wp body } Q \ s + \ «N» \ G \ s \ * \ (c \ * \ P \ s) \leq Q \ s \ \text{by(auto)}

\text{with } nnic

\text{have } s \text{wp body } Q \ s + \ «N» \ G \ s \ * \ (c \ * \ P \ s) \leq

\quad \inverse c \ * \ Q \ s

\quad \text{by(auto intro:mult-left-mono)}

\text{hence } s \text{wp body } Q \ s + \ (\inverse c \ * \ c) \ * \ «N» \ G

\quad s \ * \ P \ s \leq

\quad \inverse c \ * \ Q \ s

\quad \text{by(simp add: algebra-simps)}

\text{hence } s \text{wp body } (\lambda s. \inverse c \ * \ Q \ s) \ s + \ «N» \ G \ s \ * \ P \ s \leq

\quad \inverse c \ * \ Q \ s

\quad \text{by(simp add:cnz scalingD[OF healthy-scalingD, OF hh sQ nnic])}

\text{hence } \lambda s. \ «G» \ s \ * \ \text{wp body} \ (\lambda s. \inverse c \ * \ Q \ s) \ s + \ «N» \ G \ s \ * \ P \ s \vdash

\lambda s. \inverse c \ * \ Q \ s \ \text{by(rule le-funI)}

\text{moreover from } nnic \ sQ \ \text{have sound } (\lambda s. \inverse c \ * \ Q \ s)

\quad \text{by(intro:sound-intros)}

\text{ultimately have } \text{lfp-exp} (\lambda a \ b. \ «G» \ b \ * \ \text{wp body} \ a \ b + \ «N» \ G \ b \ * \ P \ b) \vdash

\lambda s. \inverse c \ * \ Q \ s

\quad \text{by(rule lfp-exp-lowerbound)}

\text{hence } s \text{wp body } (\lambda a \ b. \ «G» \ b \ * \ \text{wp body} \ a \ b + \ «N» \ G \ b \ * \ P \ b) \ s \leq

\quad \inverse c \ * \ Q \ s

\quad \text{by(rule le-funD)}

\text{with } nnc

\text{have } s \text{wp body } (\lambda a \ b. \ «G» \ b \ * \ \text{wp body} \ a \ b + \ «N» \ G \ b \ * \ P \ b) \ s \leq

\quad c \ * (\inverse c \ * \ Q \ s)

\quad \text{by(auto intro:mult-left-mono)}

\text{also from } cnz \ \text{have } s = Q \ s \ \text{by(simp)}

\text{finally show } \lambda a. \ c \ * \ \text{lfp-exp} (\lambda a \ b. \ «G» \ b \ * \ \text{wp body} \ a \ b + \ «N» \ G \ b \ * \ P \ b) \ a \vdash Q

\quad \text{by(rule le-funI)}

\text{next}

\text{from } sP \ \text{have sound } (\lambda s. \text{bound-of } P) \ \text{by(auto)}

\text{with } \text{hb } sP \ \text{have sound } (\text{lfp-exp } (?X \ P))

\quad \text{by(blast intro:lfp-exp-sound lfp-loop-fp)}

\text{with } nnc \ \text{show sound } (\lambda s. \ c \ * \ \text{lfp-exp } (?X \ P) \ s)

\quad \text{by(auto intro:sound-intros)}
4.2. HEALTHINESS

from bb sP nnc
show λs. «G» s * wp body (λs. bound-of (λs. c * P s)) s +
  «N G» s * (c * P s) ⊨ λs. bound-of (λs. c * P s)
by(iprover intro:lfp-loop-fp sound-intros)

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

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

from nnc have nnc: inverse c ≠ 0 by(auto)
moreover with nnc have nnc: 0 ≤ inverse c by(auto)
  moreover from nnc sP have scP: sound (λs. c * P s) by(auto intro!:sound-intros)
ultimately have ?X (λs. c * P s) (inverse c) ⊨ ?Y (λs. c * P s) (inverse c)
by(rule onesided)
with nnc have λs. c * ?X (λs. c * P s) (inverse c) s ⊨
  λs. c * ?Y (λs. c * P s) (inverse c) s
by(blast intro:mult-left-mono)
with nnc show ?Y P c ⊨ ?X P c by(simp add:mult.assoc[symmetric])
qed

next
fix P::'s expect and b::real
assume bP: bounded-by b P and nP: nneg P
show (lfp-exp (λQ s. «G» s * wp body Q s + «N G» s * P s) ⊨ λs. b)
proof(intro lfp-exp-lowerbound le-funI)
fix s::'s
from bP nP hb have bounded-by b (wp body (λs. b)) by(auto)
hence wp body (λs. b) s ≤ b by(auto)
moreover from bP nP have P s ≤ b by(auto)
ultimately have «G» s * wp body (λs. b) s + «N G» s * P s ≤ «G» s * b
+ «N G» s * b
by(auto intro!:add-mono mult-left-mono)
also have ... = b by(simp add:negate-embed field-simps)
finally show «G» s * wp body (λs. b) s + «N G» s * P s ≤ b .
from bP nP have 0 ≤ b by(auto)
thus sound (λs. b) by(auto)
qed

from hb bP nP show λs. 0 ⊨ lfp-exp (λQ s. «G» s * wp body Q s + «N G»)
  s * P s)
by(auto dest!:sound-nneg intro!:lfp-loop-greatest)

next
CHAPTER 4. THE PGCL LANGUAGE

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: \ls. "G" s * wp body R s + "N" G s * Q s ⊢ R
  from le have \ls. "G" s * wp body R s + "N" G s * P s ⊢
    \ls. "G" s * wp body R s + "N" G s * Q s
    by (auto intro:le-funI add-left-mono mult-left-mono)
  also note fp
finally show lfp-exp (\ls. "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 (\ls. "G" s * wp body R s + "N" G s * P s))
    by (rule lfp-loop-sound)
  from hb sQ
  show sound (\ls. "G" s * wp body (\ls. bound-of Q) s + "N" G s * Q
    s ⊢ \ls. bound-of Q
    by (rule lfp-loop-fp)
  from sQ
  show sound (\ls. bound-of Q) by (auto)
qed
qed

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

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

end

4.3 Continuity

theory Continuity imports Healthiness begin

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

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

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

lemma bd-ctsD:
[ 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 (Suc 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)))
⇒ bd-cts-tr T
by(simp add:bd-cts-tr-def)

4.3.1 Continuity of Primitives

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

lemma cts-wp-Skip:
bd-cts (wp Skip)
by(rule bd-ctsI, simp add:wp-def Skip-def o-def)

lemma cts-wp-Apply:
bd-cts (wp (Apply f))
proof —
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 1**: \texttt{cts-wp-Bind}:

- **Fixes**: \( a : \langle \cdot \rightarrow \text{nat} \rangle \) and \( \cdot : \text{real} \)
- **Assumes**: \( ca : \bigwedge s . \text{bd-cts} ( wp (a (f s))) \)
- **Shows**: \( \text{bd-cts} ( wp (\text{Bind} f a)) \)

**Proof** (rule \texttt{bd-ctsI})

- **Fix** \( M : \text{nat} \Rightarrow \langle \cdot \rightarrow \text{nat} \rangle \)
- **Assume** \( \text{chain} : \bigwedge i . M i \vdash M (\text{Suc} i) \) and \( sM : \bigwedge i . \text{sound} (M i) \) and \( bM : \bigwedge i . \text{bounded-by} c (M i) \)
- **With** \( \text{bd-ctsD}(OF ca) \)
- **Have** \( \bigwedge s . wp (a (f s)) (\text{Sup-exp} (\text{range} M)) = \text{Sup-exp} (\text{range} (wp (a (f s)) \circ M)) \)
  - **By** (\texttt{auto})
- **Moreover have** \( \bigwedge s . \{ fa s | fa . fa \in \text{range} (\lambda x . \text{wp} (a (f s)) (M x)) \} = \{ fa s | fa . fa \in \text{range} (\lambda x s . \text{wp} (a (f s)) (M x) s) \} \)
  - **By** (\texttt{auto})
- **Ultimately show** \( \text{wp} (\text{Bind} f a) (\text{Sup-exp} (\text{range} M)) = \text{Sup-exp} (\text{range} (wp (\text{Bind} f a) \circ M)) \)
  - **By** (\texttt{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 2**: \texttt{cts-wp-DC}:

- **Fixes**: \( a b : \langle \cdot \rangle \)
- **Assumes**: \( ca : \text{bd-cts} (wp a) \)
- **And**: \( cb : \text{bd-cts} (wp b) \)
- **And**: \( ha : \text{healthy} (wp a) \)
- **And**: \( hb : \text{healthy} (wp b) \)
- **Shows**: \( \text{bd-cts} (wp (a \sqcap b)) \)

**Proof** (rule \texttt{bd-ctsI, rule antisym})

- **Fix** \( M : \text{nat} \Rightarrow \langle \cdot \rangle \)
- **Assume** \( \text{chain} : \bigwedge i . M i \vdash M (\text{Suc} i) \) and \( sM : \bigwedge i . \text{sound} (M i) \) and \( bM : \bigwedge i . \text{bounded-by} c (M i) \)
- **From** \( ha hb \) have \( \text{hab} : \text{healthy} (wp (a \sqcap b)) \) by (rule \texttt{healthy-intros})
- **From** \( bM \) have \( \text{leSup} : \bigwedge i . M i \vdash \text{Sup-exp} (\text{range} M) \) by (auto intro:Sup-exp-upper)
- **From** \( sM bM \) have \( \text{sSup} : \text{sound} (\text{Sup-exp} (\text{range} M)) \) by (auto intro:Sup-exp-sound)
- **Show** \( \text{Sup-exp} (\text{range} (wp (a \sqcap b) \circ M)) \vdash wp (a \sqcap b) (\text{Sup-exp} (\text{range} M)) \)
- **Proof** (rule \texttt{Sup-exp-least, clarsimp, rule le-funI})
- **Fix** \( i s \)
- **From** \( \text{mono-transD}(OF \text{healthy-monoD}(OF \text{hab})] \) \( \text{leSup} sM sSup \)
- **Have** \( wp (a \sqcap b) (M i) \vdash wp (a \sqcap b) (\text{Sup-exp} (\text{range} M)) \) by (auto)
4.3. CONTINUITY

thus \( \text{wp} (a \sqcap b) (M i) s \leq \text{wp} (a \sqcap b) (\text{Sup-exp} (\text{range } M)) s \) by(auto)

from \( \text{pub} \text{sM} \text{bM} \text{ha} \) have sound \((a \sqcap b) (\text{Sup-exp} (\text{range } M))) \) by(auto)
thus \( \text{neg} (a \sqcap b) (\text{Sup-exp} (\text{range } M)) \) by(auto)
qed

from \( \text{sm bM ha} \) have \( \land i. \text{bounded-by } c \ (\text{wp } a (M i)) \) by(auto)
hence \( \text{baM} \land i. \ (\text{wp } a (M i)) s \leq c \) by(auto)
from \( \text{sm bM hb} \) have \( \land i. \text{bounded-by } c \ (\text{wp } b (M i)) \) by(auto)
hence \( \text{bbM} \land i. \ (\text{wp } b (M i)) s \leq c \) by(auto)

show \( \text{wp} (a \sqcap b) (\text{Sup-exp} (\text{range } M)) \vdash \text{Sup-exp} (\text{range } (\text{wp} (a \sqcap b) \circ M)) \)
proof(simp add:wp-eval o-def, rule le-funI)
fix \( s \)\:
from bd-ctsD[OF ca, of M, OF chain \( \text{sm bM} \)] bd-ctsD[OF cb, of M, OF chain \( \text{sm bM} \)]
have \( \text{min} \ (\text{wp } a (\text{Sup-exp} (\text{range } M))) s \ (\text{wp } b (\text{Sup-exp} (\text{range } M))) s = \text{min} \ (\text{Sup-exp} (\text{range } (\text{wp } a \circ M))) s \ (\text{Sup-exp} (\text{range } (\text{wp } b \circ M))) s \)
by(simp)
also \{
have \{f s \mid f \in \text{range } (\lambda x. \text{wp } a (M x))\} = \text{range } (\lambda i. \text{wp } a (M i)) s
\{f s \mid f \in \text{range } (\lambda x. \text{wp } b (M x))\} = \text{range } (\lambda i. \text{wp } b (M i)) s
by(auto)
hence \( \text{min} \ (\text{Sup-exp} (\text{range } (\text{wp } a \circ M))) s \ (\text{Sup-exp} (\text{range } (\text{wp } b \circ M))) s \)
= \( \text{min} \ (\text{Sup } (\text{range } (\lambda i. \text{wp } a (M i))) s) \ (\text{Sup } (\text{range } (\lambda i. \text{wp } b (M i))) s)) \)
by(simp add:Sup-exp-def o-def)
\}
also \{
have (\lambda i. \text{wp } a (M i) s) \dashrightarrow \text{Sup } (\text{range } (\lambda i. \text{wp } a (M i) s))
proof(rule increasing-LIMSEQ)
fix \( n \)
from mono-transD[OF healthy-monoD, OF ha] sm chain
show \( \text{wp } a (M n) s \leq \text{wp } a (M (\text{Suc } n)) s \) by(auto intro:le-funI)
from baM show \( \text{wp } a (M n) s \leq \text{Sup } (\text{range } (\lambda i. \text{wp } a (M i))) s) \)
by(intro cSup-upper bdd-aboveI, auto)
fix \( c\)\:
real assume \( pe: 0 < c \)
from baM have \( \text{cSup}: \text{Sup } (\text{range } (\lambda i. \text{wp } a (M i) s)) \in \text{closure } (\text{range } (\lambda i. \text{wp } a (M i) s)) \)
by(blast intro; closure-contains-Sup)
with \( pe \) obtain \( y \) where \( \text{yin}: y \in (\text{range } (\lambda i. \text{wp } a (M i) s)) \)
and \( dy: \text{dist } y (\text{Sup } (\text{range } (\lambda i. \text{wp } a (M i) s))) < e \)
by(blast dest:iffD1[OF closure-approachable])
from \( \text{yin} \) obtain \( i \) where \( y = \text{wp } a (M i) s \) by(auto)
with \( dy \) have \( \text{dist } (\text{wp } a (M i) s) (\text{Sup } (\text{range } (\lambda i. \text{wp } a (M i) s))) < e \)
by(simp)
moreover from baM have \( \text{wp } a (M i) s \leq \text{Sup } (\text{range } (\lambda i. \text{wp } a (M i) s)) \)
by(intro cSup-upper bdd-aboveI, auto)
ultimately have \( \text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s)) \leq \text{wp} \ a \ (M \ i) \ s + e \)
by(simp add:dist-real-def)
thus \( \exists i. \ \text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s)) \leq \text{wp} \ a \ (M \ i) \ s + e \) by(auto)
qed

moreover
have \( (\lambda i. \ \text{wp} \ b \ (M \ i) \ s) \rightarrow\rightarrow \text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s)) \)
proof(rule increasing-LIMSEQ)
fix \( n \)
from mono-transD[OF healthy-monoD, OF \( bbM \)] \( sm \) chain
show \( \text{wp} \ b \ (M \ n) \ s \leq \text{wp} \ b \ (M \ (\text{Suc} \ n)) \ s \) by(auto intro:le-funD)
from \( bbM \) show \( \text{wp} \ b \ (M \ n) \ s \leq \text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s)) \)
by(intro \( c\text{Sup-upper bdd-aboveI}, \ \text{auto} \)

fix \( c :: \text{real} \) assume \( pc: \ 0 < e \)
from \( bbM \) have \( \text{cSup}: \ \text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s)) \in \text{closure} \ (\text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s)) \)
by(blast intro:closure-contains-Sup)
with \( pc \) obtain \( y \) where \( yin: \ y \in (\text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s)) \)
and \( dy: \ \text{dist} \ y \ (\text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s))) < e \)
by(blast dest:iffD1[OF closure-approachable])
from \( yin \) obtain \( i \) where \( y = \text{wp} \ b \ (M \ i) \ s \) by(auto)
with \( dy \) have \( \text{dist} \ (\text{wp} \ b \ (M \ i) \ s) \ (\text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s))) < e \)
by(simp)
moreover from \( bbM \) have \( \text{wp} \ b \ (M \ i) \ s \leq \text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s)) \)
by(intro \( c\text{Sup-upper bdd-aboveI}, \ \text{auto} \)
ultimately have \( \text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s)) \leq \text{wp} \ b \ (M \ i) \ s + e \)
by(simp add:dist-real-def)
thus \( \exists i. \ \text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s)) \leq \text{wp} \ b \ (M \ i) \ s + e \) by(auto)
qed
ultimately have \( (\lambda i. \ \text{min} \ (\text{wp} \ a \ (M \ i) \ s) \ (\text{wp} \ b \ (M \ i) \ s)) \rightarrow\rightarrow \text{min} \ (\text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s))) \ (\text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s))) \)
by(rule tendsto-min)
moreover have \( \text{bdd-above} \ (\text{range} \ (\lambda i. \ \text{min} \ (\text{wp} \ a \ (M \ i) \ s) \ (\text{wp} \ b \ (M \ i) \ s))) \)
proof(intro \( \text{bdd-aboveI}, \ \text{clar simp} \)
fix \( i \)
have \( \text{min} \ (\text{wp} \ a \ (M \ i) \ s) \ (\text{wp} \ b \ (M \ i) \ s) \leq \text{wp} \ a \ (M \ i) \ s \) by(auto)
also { \( \text{from} \ ha \ \text{sm} \ \text{bbM} \ \text{have} \ \text{bounded-by} \ c \ (\text{wp} \ a \ (M \ i) \ s) \) by(auto) 
  hence \( \text{wp} \ a \ (M \ i) \ s \leq c \) by(auto) 
}
finally show \( \text{min} \ (\text{wp} \ a \ (M \ i) \ s) \ (\text{wp} \ b \ (M \ i) \ s) \leq c . \)
qed
ultimately have \( \text{min} \ (\text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s))) \ (\text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s))) \)
\leq \( \text{Sup} \ (\text{range} \ (\lambda i. \ \text{min} \ (\text{wp} \ a \ (M \ i) \ s) \ (\text{wp} \ b \ (M \ i) \ s))) \)
by(blast intro:LIMSEQ-le-const2 \( c\text{Sup-upper min.mono}[OF \baM \ bbM]\) }
4.3. CONTINUITY

also \{ 
  \begin{align*}
  &\text{have range } (\lambda i. \min (\wp a (M i) s) (\wp b (M i) s)) = \\
  &\{ f | f \in \text{range } (\lambda i s. \min (\wp a (M i) s) (\wp b (M i) s)) \} \\
  &\text{by (auto)} \\
  \end{align*}
\}

hence \begin{align*}
\sup (\text{range } (\lambda i. \min (\wp a (M i) s) (\wp b (M i) s))) = \\
\text{Sup-exp } (\text{range } (\lambda i s. \min (\wp a (M i) s) (\wp b (M i) s))) s \\
\text{by (simp add: Sup-exp-def)} \\
\}
\]

finally show \begin{align*}
\min (\wp a (\text{Sup-exp } (\text{range } M)) s) (\wp b (\text{Sup-exp } (\text{range } M)) s) \\
\leq \\
\text{Sup-exp } (\text{range } (\lambda i s. \min (\wp a (M i) s) (\wp b (M i) s))) s .
\end{align*}

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: \( \forall i. M i \vdash M (Suc i) \) and sM: \( \forall i. \text{sound } (M i) \)
and bM: \( \forall i. \text{bounded-by } c (M i) \)

hence wp a (wp b (\text{Sup-exp } (\text{range } M))) = wp a (\text{Sup-exp } (\text{range } (wp b o M)))
  by (subst bd-ctsD[OF cb], auto)
also \{ 
  from sM hb have \( \forall i. \text{sound } ((wp b o M) i) \) by (auto)
  moreover from chain sM
  have \( \forall i. (wp b o M) i \vdash (wp b o M) (Suc i) \)
  by (auto intro: mono-transD[OF healthy-monoD, OF hb])
  moreover from sM bM hb have \( \forall i. \text{bounded-by } c ((wp b o M) i) \) by (auto)
  ultimately have wp a (\text{Sup-exp } (\text{range } (wp b o M))) = \\
  \text{Sup-exp } (\text{range } (wp a o (wp b o M)))
  by (subst bd-ctsD[OF ca], auto)
\}
also have \( \text{Sup-exp } (\text{range } (wp a o (wp b o M))) = \\
\text{Sup-exp } (\text{range } (\lambda i. wp a (wp b (M i)))) \)
  by (simp add: o-def)

finally show \( wp a (wp b (\text{Sup-exp } (\text{range } M))) = \\
\text{Sup-exp } (\text{range } (\lambda i. wp a (wp b (M i)))) .
\)

qed 

lemma cts-wp-PC:
fixes a b :: 's prog
assumes ca: bd-cts (wp a)
  and cb: bd-cts (wp b)
  and ha: healthy (wp a)
  and hb: healthy (wp b)
and \( wp \) unitary \( p \)
shows bd-cts (\( wp (PC a p b) \))
proof (rule bd-ctsI, rule ext, simp add:o-def wp-eval)
fix \( M::nat \Rightarrow 's \) expect and c::real and \( s::'s \)
assume chain: \( \forall i. M i \vdash M (Suc i) \) and \( sM: \forall i. sound (M i) \)
and \( bM: \forall i. bounded-by c (M i) \)
from \( sM \) have \( \forall i. nneg (M i) \) by(auto)
with \( bM \) have \( nc: 0 \leq c \) by(auto)
from chain \( sM \) \( bM \) have \( wp \ a \ (Sup-exp (range M)) = Sup-exp (range (wp a o M)) \)
  by(rule bd-ctsD[OF ca])
hence \( wp \ a \ (Sup-exp (range M)) \ s = Sup-exp (range (wp a o M)) \ s \)
  by(simp)
also {  
  have \( \{ f s \mid f. f \in range (\lambda x. wp a (M x)) \} = range (\lambda i. wp a (M i) s) \)  
    by(auto)  
  hence \( Sup-exp (range (wp a o M)) \ s = Sup (range (\lambda i. wp a (M i) s)) \)
    by(simp add:Sup-exp-def o-def)
}
finally have \( p s \star wp \ a \ (Sup-exp (range M)) \ s = \)
  \( p s \star Sup (range (\lambda i. wp a (M i) s)) \) by(simp)
also have \( \ldots = Sup \{ p s \star x \mid x. x \in range (\lambda i. wp a (M i) s) \} \)
proof (rule cSup-mult, blast, clarsimp)
  from \( wp \) show \( \theta \leq p s \) by(auto)
fix \( i \)
from \( sM \) \( bM \) ha have bounded-by c (\( wp a (M i) \)) by(auto)
thus \( wp a (M i) s \leq c \) by(auto)
qed
also {  
  have \( \{ p s \star x \mid x. x \in range (\lambda i. wp a (M i) s) \} = range (\lambda i. p s \star wp a (M i) s) \)  
    by(auto)  
  hence \( Sup \{ p s \star x \mid x. x \in range (\lambda i. wp a (M i) s) \} = \)
    \( Sup (range (\lambda i. p s \star wp a (M i) s)) \) by(simp)
}
finally have \( p s \star wp \ a \ (Sup-exp (range M)) \ s = Sup (range (\lambda i. p s \star wp a (M i) s)) \)
moreover {  
  from chain \( sM \) \( bM \) have \( wp \ b \ (Sup-exp (range M)) = Sup-exp (range (wp b o M)) \)
    by(rule bd-ctsD[OF cb])  
  hence \( wp \ b \ (Sup-exp (range M)) \ s = Sup-exp (range (wp b o M)) \ s \)
    by(simp)
  also {  
    have \( \{ f s \mid f. f \in range (\lambda x. wp b (M x)) \} = range (\lambda i. wp b (M i) s) \)
      by(auto)  
    hence \( Sup-exp (range (wp b o M)) \ s = Sup (range (\lambda i. wp b (M i) s)) \)
  }
4.3. CONTINUITY

by(simp add: Sup-exp-def o-def)

} finally have \((1 - p \cdot s) * wp b (Sup-exp (range M)) s =
(1 - p \cdot s) * Sup (range (\lambda i. wp b (M i) s)) \) by(simp)
also have ... = Sup \{ (1 - p \cdot s) * x | x \in range (\lambda i. wp b (M i) s) \}

proof (rule cSup-mult, blast, clarsimp)
  from up show \(0 \leq 1 - p \cdot s \) by(auto simp: sign-simps)
  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 \(\{(1 - p \cdot s) * x | x \in range (\lambda i. wp b (M i) s)\} =
range (\lambda i. (1 - p \cdot s) * wp b (M i) s)
\) by(auto)
  hence Sup \{ (1 - p \cdot s) * x | x \in range (\lambda i. wp b (M i) s) \} =
Sup (range (\lambda i. (1 - p \cdot s) * wp b (M i) s)) \) by(simp)
}

finally have \((1 - p \cdot s) * wp b (Sup-exp (range M)) s =
Sup (range (\lambda i. (1 - p \cdot s) * wp b (M i) s)) \).
}

ultimately
have \(p \cdot s \cdot wp a (Sup-exp (range M)) s + (1 - p \cdot s) * wp b (Sup-exp (range M))\)
\(s =
Sup (range (\lambda i. p \cdot s \cdot wp a (M i) s)) + Sup (range (\lambda i. (1 - p \cdot s) * wp b (M i) s))\)
by(simp)
also { from bM sM ha have \(\land i. \text{bounded-by c } (wp a (M i))\) by(auto)
  hence \(\land i. wp a (M i) s \leq c\) by(auto)
  moreover from up have \(0 \leq p \cdot s\) by(auto)
  ultimately have \(\land i. p \cdot s \cdot wp a (M i) s \leq p \cdot s \cdot c\) by(auto simp: mult-left-mono)
  also from wp nc have \(p \cdot s \cdot c \leq 1 * c\) by(blast simp: mult-right-mono)
  also have ... = \(c\) by(simp)
  finally have baM: \(\land i. p \cdot s \cdot wp a (M i) s \leq c\).

have lima: \((\lambda i. p \cdot s \cdot wp a (M i) s) \longrightarrow Sup (range (\lambda i. p \cdot s \cdot wp a (M i) s))\)
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 \(p \cdot s \cdot wp a (M n) s \leq p \cdot s \cdot wp a (M (Suc n)) s\)
by(blast intro: mult-left-mono)
  from baM show \(p \cdot s \cdot wp a (M n) s \leq Sup (range (\lambda i. p \cdot s \cdot wp a (M i) s))\)
by(intro cSup-upper bdd-aboveI, auto)
next
  fix \(e::real\)
  assume pe: \(\theta < e\)
from baM have $\sup (\text{range} (\lambda i. p \cdot s \cdot \text{wp} a (M \cdot i) \cdot s)) \in$
closure (\text{range} (\lambda i. p \cdot s \cdot \text{wp} a (M \cdot i) \cdot s))
by(blast intro\text{-}closure\text{-}contains\text{-}Sup)

\text{thm} \text{ closure}-approachable

with pe obtain $y$ where yin: $y \in \text{range} (\lambda i. p \cdot s \cdot \text{wp} a (M \cdot i) \cdot s)$
and dy: $\text{dist} y (\sup (\text{range} (\lambda i. p \cdot s \cdot \text{wp} a (M \cdot i) \cdot s))) < e$
by(blast dest::iffD1[of closure\text{-}approachable])

from yin obtain $i$ where $y = p \cdot s \cdot \text{wp} a (M \cdot i) \cdot s$ by(auto)
with dy have $\text{dist} (p \cdot s \cdot \text{wp} a (M \cdot i) \cdot s) (\sup (\text{range} (\lambda i. p \cdot s \cdot \text{wp} a (M \cdot i) \cdot s))) < e$
by(simp)
moreover from baM have $p \cdot s \cdot \text{wp} a (M \cdot i) \cdot s \leq \sup (\text{range} (\lambda i. p \cdot s \cdot \text{wp} a (M \cdot i) \cdot s))$
by(intro cSup-upper bdd\text{-}aboveI, auto)

ultimately have $\sup (\text{range} (\lambda i. p \cdot s \cdot \text{wp} a (M \cdot i) \cdot s)) \leq p \cdot s \cdot \text{wp} a (M \cdot i) \cdot s + e$
by(simp add:dist\text{-}real\text{-}def)
thus $\exists i. \sup (\text{range} (\lambda i. p \cdot s \cdot \text{wp} a (M \cdot i) \cdot s)) \leq p \cdot s \cdot \text{wp} a (M \cdot i) \cdot s + e$
by(auto)

qed

from bM sM bbM have $\bigwedge i. \text{ bounded\text{-}by} c (\text{wp} b (M \cdot i))$ by(auto)

hence $\bigwedge i. \text{ wp} b (M \cdot i) \cdot s \leq c$ by(auto)

moreover from wp have $0 \leq (1 - p \cdot s)$ by(auto simp\text{-}sign\text{-}simps)

ultimately have $\bigwedge i. (1 - p \cdot s) \cdot \text{wp} b (M \cdot i) \cdot s \leq (1 - p \cdot s) \cdot c$ by(auto intro\text{-}mult\text{-}left\text{-}mono)

also {
from wp have $1 - p \cdot s \leq 1$ by(auto)
with nc have $(1 - p \cdot s) \cdot c \leq 1 \cdot c$ by(blast intro\text{-}mult\text{-}right\text{-}mono)
}

also have $1 \cdot c = c$ by(simp)

finally have bbM: $\bigwedge i. (1 - p \cdot s) \cdot \text{wp} b (M \cdot i) \cdot s \leq c$.

have limb: $(\lambda i. (1 - p \cdot s) \cdot \text{wp} b (M \cdot i) \cdot s) \dashrightarrow \sup (\text{range} (\lambda i. (1 - p \cdot s) \cdot \text{wp} b (M \cdot i) \cdot s))$

by(rule\text{-}increasing\text{-}LIMSEQ)

fix $n$

from sM chain healthy\text{-}monoD[of OF bb] have $\text{wp} b (M \cdot n) \vdash \text{wp} b (M \cdot (\text{Suc} \cdot n))$
by(auto)

moreover from wp have $0 \leq 1 - p \cdot s$ by(auto simp\text{-}sign\text{-}simps)

ultimately show $(1 - p \cdot s) \cdot \text{wp} b (M \cdot n) \cdot s \leq (1 - p \cdot s) \cdot \text{wp} b (M \cdot (\text{Suc} \cdot n))$

by(blast intro\text{-}mult\text{-}left\text{-}mono)

from bbM show $(1 - p \cdot s) \cdot \text{wp} b (M \cdot n) \cdot s \leq \sup (\text{range} (\lambda i. (1 - p \cdot s) \cdot \text{wp} b (M \cdot i) \cdot s))$
by(intro cSup-upper bdd\text{-}aboveI, auto)

next
fix $e::\text{real}$
assume pe: $0 < e$
from \( bbM \) have \( \text{Sup} (\text{range} (\lambda i. (1 - p) s) \ast \text{wp} b (M i) s)) \in \\
\text{closure} (\text{range} (\lambda i. (1 - p) s) \ast \text{wp} b (M i) s)) \\
\text{by}(\text{blast intro:closure-contains-Sup}) \\
\text{with pe obtain} \ y \ \text{where} \ y = (1 - p) s \ast \text{wp} b (M i) s \ \text{by(auto)} \\
\text{with dy have} \ \text{dist} ((1 - p) s) \ast \text{wp} b (M i) s \\
\quad (\text{Sup} (\text{range} (\lambda i. (1 - p) s) \ast \text{wp} b (M i) s))) < e \\
\quad \text{by}(\text{simp}) \\
\text{moreover from} \ bbM \\
\text{have} \ (1 - p) s \ast \text{wp} b (M i) s \leq \text{Sup} (\text{range} (\lambda i. (1 - p) s) \ast \text{wp} b (M i) s)) \\
\text{by}(\text{intro cSup-upper bdd-aboveI, auto}) \\
\text{ultimately have} \ \text{Sup} (\text{range} (\lambda i. (1 - p) s) \ast \text{wp} b (M i) s)) \leq (1 - p) s \ast \text{wp} b (M i) s + e \\
\text{by}(\text{simp add:dist-real-def}) \\
\text{thus} \ \exists i. \ \text{Sup} (\text{range} (\lambda i. (1 - p) s) \ast \text{wp} b (M i) s)) \leq (1 - p) s \ast \text{wp} b (M i) s + e \ \text{by(auto)} \\
\text{qed} \\
\text{from lima limb have} \ (\lambda i. p s \ast \text{wp} a (M i) s) + (1 - p) s \ast \text{wp} b (M i) s) \\
\text{----->} \\
\text{Sup} (\text{range} (\lambda i. p s \ast \text{wp} a (M i) s) + \text{Sup} (\text{range} (\lambda i. (1 - p) s) \ast \text{wp} b (M i) s)) \\
\text{by}(\text{rule tendsto-add}) \\
\text{moreover from} \ \text{add-mono}[OF bM bbM] \\
\text{have} \ \big\{ \lambda i. p s \ast \text{wp} a (M i) s + (1 - p) s \ast \text{wp} b (M i) s \leq \text{Sup} (\text{range} (\lambda i. p s \ast \text{wp} a (M i) s + (1 - p) s) \ast \text{wp} b (M i) s)) \big\} \\
\text{by(auto)} \\
\text{hence} \text{Sup} (\text{range} (\lambda i. p s \ast \text{wp} a (M i) s + (1 - p) s) \ast \text{wp} b (M i) s)) = \\
\text{Sup-exp} (\text{range} (\lambda x s. p s \ast \text{wp} a (M x) s + (1 - p) s) \ast \text{wp} b (M x) s)) s \\
\text{by}(\text{simp add:Sup-exp-def}) \\
\} \\
\text{also} \ \{ \\
\text{have range} (\lambda i. p s \ast \text{wp} a (M i) s + (1 - p) s) \ast \text{wp} b (M i) s) = \\
\{ f s \mid f \in \text{range} (\lambda x s. p s \ast \text{wp} a (M x) s + (1 - p) s) \ast \text{wp} b (M x) s) \} \\
\text{by(auto)} \\
\} \\
\text{finally} \\
\text{have} \ p s \ast \text{wp} a (\text{Sup-exp} (\text{range} M)) s + (1 - p) s \ast \text{wp} b (\text{Sup-exp} (\text{range} M)) s \leq \\
\text{Sup-exp} (\text{range} (\lambda i s. p s \ast \text{wp} a (M i) s + (1 - p) s) \ast \text{wp} b (M i) s)) s \ast \\
\text{moreover} \\
\text{have} \ \text{Sup-exp} (\text{range} (\lambda i s. p s \ast \text{wp} a (M i) s + (1 - p) s) \ast \text{wp} b (M i) s)) s \leq \\
p s \ast \text{wp} a (\text{Sup-exp} (\text{range} M)) s + (1 - p) s) \ast \text{wp} b (\text{Sup-exp} (\text{range} M)) s
proof (rule le-funD [OF Sup-exp-least], clarsimp, rule le-funI)
fix i :: nat and s :: 's
from bM have leSup: M i ⊢ Sup-exp (range M)
  by (blast intro: Sup-exp-upper)
moreover from sM bM have sSup: sound (Sup-exp (range M))
  by (auto intro: Sup-exp-sound)
moreover note healthy-monoD [OF ha]
ultimately have wp a (M i) s ≤ wp a (Sup-exp (range M)) s
  by (auto)
moreover from leSup sSup healthy-monoD [OF hb]
ultimately have wp b (M i) s ≤ wp b (Sup-exp (range M)) s
  by (auto)
moreover from up have 0 ≤ p s ≤ 1 − p s
  by (auto simp: sign-simps)
ultimately show p s * wp a (M i) s + (1 − p s) * wp b (M i) s ≤
  wp a (Sup-exp (range M)) s + (1 − p s) * wp b (Sup-exp (range M)) s
  by (blast intro: add-mono mult-left-mono)
moreover from sSup ha hb have sound (wp a (Sup-exp (range M)))
  sound (wp b (Sup-exp (range M)))
  by (auto)
ultimately show \( \forall s. \ 0 \leq wp a (\text{Sup-exp} (\text{range } M)) s \land \forall s. \ 0 \leq wp b (\text{Sup-exp} (\text{range } M)) s \)
  by (auto)
moreover from up have \( \forall s. \ 0 \leq p s \land \forall s. \ 0 \leq 1 - p s \)
  by (auto simp: sign-simps)
ultimately show \( \forall \lambda s. \ p s * wp a (\text{Sup-exp} (\text{range } M)) s + (1 - p s) * wp b (\text{Sup-exp} (\text{range } M)) s = \text{Sup-exp} (\lambda x s. \ p s * wp a (M x) s + (1 - p s) * wp b (M x) s)) s \)
  by (auto)
qed

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

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

lemma SetPC-remove:
assumes nz: p x ≠ 0 and n1: p x ≠ 1
and \( \text{fsupp: finite (supp p)} \)
shows \( \text{SetPC} \ a \ (\lambda\ -. \ p) = \text{PC} \ (a \ x) \) (\( \lambda\ -. \ p \ x \) (SetPC \ a \ (\lambda\ -. \ \text{dist-remove} \ p \ x))

\textbf{proof}(\text{intro ext, simp add:SetPC-def PC-def})

\( \text{fix} \ ab \ P \ s \)
\( \text{from} \ nz \ \text{have} \ x \in \ \text{supp} \ p \ \text{by}(\text{simp add:supp-def}) \)
\( \text{hence} \ \text{supp} \ p = \text{insert} \ x \ (\text{supp} \ p - \{x\}) \ \text{by(auto)} \)
\( \text{hence} \ (\sum x \in \text{supp} \ p. \ p \ x \ a \ x \ ab \ P \ s) = \)
\( \ (\sum x \in \text{insert} \ x \ (\text{supp} \ p - \{x\}). \ p \ x \ a \ x \ ab \ P \ s) \)
\( \ \text{by}(\text{simp}) \)
\( \text{also from} \ \text{fsupp} \)
\( \text{have} \ \ldots = \ p \ x \ a \ x \ ab \ P \ s + (\sum x \in \text{supp} \ p - \{x\}. \ p \ x \ a \ x \ ab \ P \ s) \)
\( \ \text{by}(\text{blast intro:setsum.insert}) \)
\( \text{also from} \ \text{n1} \)
\( \text{have} \ \ldots = \ p \ x \ a \ x \ ab \ P \ s + (1 - p \ x) * ((\sum x \in \text{supp} \ p - \{x\}. \ p \ x \ a \ x \ ab \ P \ s) / (1 - p \ x)) \)
\( \ \text{by}(\text{simp add:field-simps}) \)
\( \text{also have} \ \ldots = \ p \ x \ a \ x \ ab \ P \ s + (1 - p \ x) * ((\sum y \in \text{supp} \ p - \{x\}. \ dist-remove \ p \ x \ y \ a \ y \ ab \ P \ s)) \)
\( \ \text{by}(\text{simp add:dist-remove-def}) \)
\( \text{also from} \ \text{n1} \)
\( \text{have} \ \ldots = \ p \ x \ a \ x \ ab \ P \ s + (1 - p \ x) * ((\sum y \in \text{supp} \ (\text{dist-remove} \ p \ x). \ dist-remove \ p \ x \ y \ a \ y \ ab \ P \ s)) \)
\( \ \text{by}(\text{simp add:supp-dist-remove}) \)
\( \text{finally show} \ (\sum x \in \text{supp} \ p. \ p \ x \ a \ x \ ab \ P \ s) = \)
\( \ p \ x \ a \ x \ ab \ P \ s + (1 - p \ x) * (\sum y \in \text{supp} \ (\text{dist-remove} \ p \ x). \ dist-remove \ p \ x \ y \ a \ y \ ab \ P \ s) \).
\( \text{qed} \)

\textbf{lemma} \ cts-bot:
\( \text{bd-cts} \ (\lambda(P::'s expect) \ (s::'s). \ 0::real) \)
\( \text{proof} - \)
\( \text{have} \ X: \ \bigwedge \ s::'s. \ {\{(P::'s expect) \ s \mid \ P \ P \in \text{range} \ (\lambda P s. \ 0)\}} = \{0\} \ \text{by(auto)} \)
\( \text{show} \ \?thesis \ \text{by}(\text{intro bd-ctsI, simp add:Sup-exp-def o-def X}) \)
\( \text{qed} \)

\textbf{lemma} \ wp-SetPC-nil:
\( \text{wp} \ (\text{SetPC} \ a \ (\lambda s \ a. \ 0)) = (\lambda P s. \ 0) \)
\( \text{by}(\text{intro ext, simp add:wp-eval}) \)

\textbf{lemma} \ SetPC-sgl:
\( \text{supp} \ p = \{x\} \Rightarrow \text{SetPC} \ a \ (\lambda-. \ p) = (\lambda ab P s. \ p \ x \ a \ x \ ab \ P \ s) \)
\( \text{by}(\text{simp add:SetPC-def}) \)

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

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

from sM bM have sSup: sound (Sup-exp (range M)) by(auto intro:Sup-exp-sound)
with healthy-scalingD[OF ha] nnc
have c * a (Sup-exp (range M)) s = a (λs. c * Sup-exp (range M) s) s
  by(auto intro:scalingD)
also {  
  have \s. {f s | f ∈ range M} = range (\i. M i s) by(auto)
  hence a (λs. c * Sup-exp (range M) s) s =
    a (λs. c * Sup (range (\i. M i s))) s
    by(simp add:Sup-exp-def)
}  
also {
  from bM have \x s. x ∈ range (\i. M i s) \implies x ≤ d by(auto)
  with nnc have a (λs. c * Sup (range (\i. M i s))) s =
    a (λs. Sup {c*x | x ∈ range (\i. M i s)}) s
    by(subst cSup-mult, blast+)
}  
also {  
  have X: \s. {c * x | x ∈ range (\i. M i s)} = range (\i. c * M i s) by(auto)
  have a (λs. Sup {c * x | x ∈ range (\i. M i s)}) s =
    a (λs. Sup (range (\i. c * M i s))) s by(simp add:X)
}  
also {
  have \s. range (\i. c * M i s) = {f s | f ∈ range (\i s. c * M i s)}
    by(auto)
  hence (λs. Sup (range (\i. c * M i s))) = Sup-exp (range (\i s. c * M i s))
    by(simp add:Sup-exp-def)
  hence a (λs. Sup (range (\i. c * M i s))) s =
    a (Sup-exp (range (\i s. c * M i s))) s by(simp)
}  
also {
  from le-funD[OF chain] nnc
  have \i. (λs. c * M i s) ⊢ (λs. c * M (Suc i) s)
    by(auto intro:le-funD[OF mult-left-mono])
  moreover from sM nnc
  have \i. sound (\i. c * M i s)
    by(auto intro:sound-intros)
4.3. CONTINUITY

\[\text{moreover from } bM \text{ nnc }\]
\[\text{have } \bigwedge_i \text{ bounded-by } (\lambda s. c \ast M i s)\]
\[\text{by} (\text{auto intro: mult-left-mono})\]

\[\text{ultimately}\]
\[\text{have } a \ (\text{Sup-exp } (\text{range } (\lambda i s. c \ast M i s))) = \]
\[\text{Sup-exp } (\text{range } (a \ o (\lambda i s. c \ast M i s)))\]
\[\text{by} (\text{rule bd-ctsD[OF ca]})\]

\[\text{hence } a \ (\text{Sup-exp } (\text{range } (\lambda i s. c \ast M i s))) s = \]
\[\text{Sup-exp } (\text{range } (a \ o (\lambda i s. c \ast M i s))) s\]
\[\text{by(\text{auto})}\]

\[\} \]
\[\text{also have } \text{Sup-exp } (\text{range } (a \ o (\lambda i s. c \ast M i s))) s = \]
\[\text{Sup-exp } (\text{range } (\lambda x. a (\lambda s. c \ast M x s))) s\]
\[\text{by(\text{simp add:a-o-def})}\]

\[\} \]
\[\text{also } \{\]
\[\text{from } nnc \ sM\]
\[\text{have } \bigwedge x. a (\lambda s. c \ast M x s) = (\lambda s. c \ast a (M x) s)\]
\[\text{by} (\text{auto intro:scalingD[OF healthy-scalingD, OF ha, symmetric]})\]

\[\text{hence } \text{Sup-exp } (\text{range } (\lambda x. a (\lambda s. c \ast M x s))) s = \]
\[\text{Sup-exp } (\text{range } (\lambda x. c \ast a (M x) s)) s\]
\[\text{by(\text{simp})}\]

\[\}\]
\[\text{finally show } c \ast a \ (\text{Sup-exp } (\text{range } M)) s = \text{Sup-exp } (\text{range } (\lambda x. c \ast a (M x) s)) s\]
\[\text{qed}\]

\textbf{lemma cts-wp-SetPC-const:}
\begin{itemize}
    \item \textbf{fixes } a::'a \Rightarrow 's prog
    \item \textbf{assumes } ca: \bigwedge x. x \in (\text{supp } p) \implies \text{bd-cts } (wp (a x))
    \item \text{and } ha: \bigwedge x. x \in (\text{supp } p) \implies \text{healthy } (wp (a x))
    \item \text{and } up: \text{unitary } p
    \item \text{and } sump: \text{setsum } p \ (\text{supp } p) \leq 1
    \item \text{and } fsupp: \text{finite } (\text{supp } p)
\end{itemize}
\[\text{shows } \text{bd-cts } (wp (\text{SetPC } a (\lambda\cdot . p)))\]
\[\text{proof}(\text{cases } \text{supp } p = \{\}, \text{ simp add:supp-empty SetPC-def wp-def cts-bot})\]
\[\text{assume } \text{nesupp: } \text{supp } p \neq \{\}\]
\[\text{from } \text{fsupp} \text{ have unitary } p \longrightarrow \text{setsum } p \ (\text{supp } p) \leq 1 \longrightarrow \]
\[\ (\forall x \in \text{supp } p. \text{ bd-cts } (wp (a x))) \longrightarrow \]
\[\ (\forall x \in \text{supp } p. \text{ healthy } (wp (a x))) \longrightarrow \]
\[\text{bd-cts } (wp (\text{SetPC } a (\lambda\cdot . p)))\]
\[\text{proof}(\text{induct } \text{supp } p \text{ arbitrary;} p, \text{ simp add:supp-empty wp-SetPC-nil cts-bot, clarify})\]
\[\text{fix } x::'a \text{ and } F::'a set \text{ and } p::'a \Rightarrow \text{real}\]
\[\text{assume } fF: \text{finite } F\]
\[\text{assume } \text{insert } x F = \text{supp } p\]
\[\text{hence } \text{pstep: } \text{supp } p = \text{insert } x F \text{ by(simp)}\]
\[\text{hence } \text{xin: } x \in \text{supp } p \text{ by(auto)}\]
\[\text{assume } wp: \text{unitary } p \text{ and } ca: \forall x \in \text{supp } p. \text{ bd-cts } (wp (a x))\]
\[\text{and } ha: \forall x \in \text{supp } p. \text{ healthy } (wp (a x))\]
and \(\text{supp}: \text{setsum} \ p \ (\text{supp} \ p) \leq 1\)
and \(\text{xri}: x \notin F\)

**assume IH:** \(\bigwedge \ p. \ F = \text{supp} \ p \implies\)
\((\forall x \in \text{supp} \ p. \ \text{bd-cts} \ (wp \ (a \ x))) \implies\)
\((\forall x \in \text{supp} \ p. \ \text{healthy} \ (wp \ (a \ x))) \implies\)
\(\text{bd-cts} \ (wp \ (\text{SetPC} \ a \ (\lambda_. \ p)))\)

from \(fF\) \(\text{pstep} \) have \(\text{fsupp}: \text{finite} \ (\text{supp} \ p)\) by (auto)

from \(\text{xin}\) have \(\text{nzp}: \ p \ x \neq 0\) by (simp add: \text{supp-def})

**have xy-le-sum:**
\(\bigwedge \ y. \ y \in \text{supp} \ p \implies y \neq x \implies p \ x + p \ y \leq \text{setsum} \ p \ (\text{supp} \ p)\)

**proof** -
fix \(y\) assume \(\text{yin}: y \in \text{supp} \ p\) and \(\text{yne}: y \neq x\)
from \(\text{wp}\) have \(0 \leq \text{setsum} \ p \ (\text{supp} \ p - \{x,y\})\)
by (auto intro: \text{setsum-nonneg})
hence \(p \ x + p \ y \leq p \ x + p \ y + \text{setsum} \ p \ (\text{supp} \ p - \{x,y\})\)
by (auto)
also {
from \(\text{yin} \ \text{yne} \ \text{fsupp}\)
have \(p \ y + \text{setsum} \ p \ (\text{supp} \ p - \{x,y\}) = \text{setsum} \ p \ (\text{supp} \ p - \{x\})\)
by (subst \text{setsum.insert[symmetric]}, (blast intro!: \text{setsum.cong})+)
moreover
from \(\text{xin} \ \text{fsupp}\)
have \(p \ x + \text{setsum} \ p \ (\text{supp} \ p - \{x\}) = \text{setsum} \ p \ (\text{supp} \ p)\)
by (subst \text{setsum.insert[symmetric]}, (blast intro!: \text{setsum.cong})+)
ultimately
have \(p \ x + p \ y + \text{setsum} \ p \ (\text{supp} \ p - \{x,y\}) = \text{setsum} \ p \ (\text{supp} \ p)\) by (simp)
}
finally show \(p \ x + p \ y \leq \text{setsum} \ p \ (\text{supp} \ p)\) .
qed

**have n1p:** \(\bigwedge \ y. \ y \in \text{supp} \ p \implies y \neq x \implies p \ x \neq 1\)

**proof** (rule ccontr, simp)
assume \(\text{px1}: p \ x = 1\)
fix \(y\) assume \(\text{yin}: y \in \text{supp} \ p\) and \(\text{yne}: y \neq x\)
from \(\text{wp}\) have \(0 \leq p \ y\) by (auto)
with \(\text{yin}\) have \(0 < p \ y\) by (auto simp: \text{supp-def})
hence \(0 + p \ x < p \ y + p \ x\) by (rule add-strict-right-mono)
with \(\text{px1}\) have \(1 < p \ x + p \ y\) by (simp)
also from \(\text{yin} \ \text{yne}\) have \(p \ x + p \ y \leq \text{setsum} \ p \ (\text{supp} \ p)\)
by (rule xy-le-sum)
finally show \(\text{False}\) using \(\text{supp}\) by (simp)
qed

**show** \(\text{bd-cts} \ (wp \ (\text{SetPC} \ a \ (\lambda_. \ p)))\)

**proof** (cases \(F = \{\}\))
4.3. CONTINUITY

\[ \text{case True with } p \text{ step have } \text{supp } p = \{ x \} \text{ by(simp)} \]

\[ \text{hence } wp (\text{SetPC } a (\lambda\cdot p)) = (\lambda P \ s \ \cdot \ p \ x * wp (a \ x) \ P \ s) \]

\[ \text{by(simp add:SetPC-sgl wp-def)} \]

\[ \text{moreover} \]

\[ \text{from } up \ ca \ ha \ xin \ \text{have bd-cts (wp (a \ x)) } \text{healthy (wp (a \ x)) } 0 \leq p \ x \]

\[ \text{by(auto)} \]

\[ \text{hence bd-cts (}\lambda P \ s \ \cdot \ p \ x * wp (a \ x) \ P \ s) \]

\[ \text{by(rule bd-cts-scale)} \]

\[ \}

\[ \text{ultimately show } \text{thesis by(simp)} \]

\[ \text{next} \]

\[ \text{assume neF: } F \neq \{\} \]

\[ \text{then obtain } y \ \text{where yinF: } y \in F \text{ by(auto)} \]

\[ \text{with } xni \text{ have yne: } y \neq x \text{ by(auto)} \]

\[ \text{from yinF pstep have yin: } y \in \text{supp } p \text{ by(auto)} \]

\[ \text{from } \text{supp-dist-remove[of p x, OF nzp n1p, OF yin yne]} \]

\[ \text{have supp-sub: supp (dist-remove p x) } \subseteq \text{supp } p \text{ by(auto)} \]

\[ \text{from } xin \ ca \ \text{have cax: bd-cts (wp (a \ x)) by(auto)} \]

\[ \text{from } xin \ ha \ \text{have hax: healthy (wp (a \ x)) by(auto)} \]

\[ \text{from supp-sub ha have hra: } \forall x \in \text{supp (dist-remove p x). healthy (wp (a \ x)) by(auto)} \]

\[ \text{from supp-dist-remove[of p x, OF nzp n1p, OF yin yne] pstep xni} \]

\[ \text{have Fsupp: } F = \text{supp (dist-remove p x) by(simp)} \]

\[ \text{have udp: unitary (dist-remove p x) by(simp)} \]

\[ \text{proof(intro unitaryI2 nnegI bounded-byI)} \]

\[ \text{fix } y \]

\[ \text{show } 0 \leq \text{dist-remove p x y by(auto)} \]

\[ \text{proof(cases } y=x, \text{ simp-all add:dist-remove-def)} \]

\[ \text{from } wp \text{ have } 0 \leq p \ y \ 0 \leq 1 - p \ x \text{ by(auto simp:sign-simps)} \]

\[ \text{thus } 0 \leq p \ y / (1 - p \ x) \]

\[ \text{by(rule divide-nonneg-nonneg)} \]

\[ \text{qed} \]

\[ \text{show dist-remove p x y } \leq 1 \]

\[ \text{proof(cases } y=x, \text{ simp-all add:dist-remove-def, cases } y\in \text{supp } p, \text{ simp-all add:nsupp-zero)} \]

\[ \text{assume yne: } y \neq x \text{ and yin: } y \in \text{supp } p \]

\[ \text{hence } p \ x + p \ y \leq \text{setsum } p \ (\text{supp } p) \]

\[ \text{by(auto intro:xy-le-sum)} \]

\[ \text{also note sump} \]

\[ \text{finally have } p \ y \leq 1 - p \ x \text{ by(auto)} \]

\[ \text{moreover from } wp \text{ have } p \ x \leq 1 \text{ by(auto)} \]
moreover from \( \text{gin yne} \) have \( p \neq 1 \) by (rule \( nIp \))
ultimately show \( p y / (1 - p x) \leq 1 \) by (auto)
qed
qed

from \( \text{xin} \) have \( pxn0: p x \neq 0 \) by (auto simp:supp-def)
from \( \text{gin yne} \) have \( pxn1: p x \neq 1 \) by (rule \( nIp \))

from \( pxn0 pxn1 \) have setsum (dist-remove p x) (supp (dist-remove p x)) =
setsum (dist-remove p x) (supp p - \{x\})
by (simp add:supp-dist-remove)
also have ... = \( \sum y \in \text{supp p} - \{x\}. p y / (1 - p x) \)
by (simp add:dist-remove-def)
also have ... = \( \sum y \in \text{supp p} - \{x\}. p y / (1 - p x) \)
by (simp add:setsum-divide-distrib)
also {
from \( \text{xin} \) have insert x \( (\text{supp p}) = \text{supp p} \) by (auto)
with \( \text{fsupp} \) have \( p x + \sum y \in \text{supp p} - \{x\}. p y / (1 - p x) = \) setsum p \( (\text{supp p}) \)
by (simp add:setsum.insert[symmetric])
also note \( \text{sump} \)
finally have setsum p \( (\text{supp p} - \{x\}) \leq 1 - p x \) by (auto)
moreover {
from \( \text{up} \) have \( p x \leq 1 \) by (auto)
with \( \text{pxn1} \) have \( p x < 1 \) by (auto)
hence \( 0 < 1 - p x \) by (auto)
}
ultimately have setsum p \( (\text{supp p} - \{x\}) / (1 - p x) \leq 1 \)
by (auto)
}
finally have \( \text{sdp: setsum (dist-remove p x) (supp (dist-remove p x))} \leq 1 \).

from \( \text{Fsupp udp sdp hra cra IH} \)
have \( \text{cts-dr: bd-cts (wp (SetPC a (\lambda-. dist-remove p x)))} \)
by (auto)

from \( \text{up} \) have \( \text{upx: unitary (\lambda-. p x)} \) by (auto)

from \( pxn0 pxn1 \) \( \text{fsupp hra show \ ?thesis} \)
by (simp add:SetPC-remove,
blast intro:cts-wp-PC caz cts-dr hax healthy-intros
unitary-sound[OF udp] sdp upx)
qed
qed
with \( \text{assms} \) show \( \text{?thesis} \) by (auto)
qed

lemma \( \text{cts-wp-SetPC:} \)
fixes \( a::a \Rightarrow 's \text{ prog} \)
assumes \( \text{ca: } \forall x. s. x \in (\text{supp (p s)}) \Rightarrow \text{bd-cts (wp (a x))} \)
4.3. CONTINUITY

\[ x : x \in (\text{supp} (p s)) \implies \text{healthy (wp (a x))} \]
\[ \text{wp: } a : a \text{ prog} \]
\[ \text{and ha: } \forall x. x \in S \implies \text{bd-cts (wp (a x))} \]
\[ \text{and ha: } \forall x. x \in S \implies \text{bd-cts (wp (a x))} \]
\[ \text{and fS: finite S} \]
\[ \text{and neS: S \neq \{\}} \]
\[ \text{shows bd-cts (wp (SetDC a \{\lambda \}. p)))} \]

\text{lemma wp-SetDC-Bind:}
\[ \text{SetDC a S = Bind S (\lambda S. SetDC a (\lambda -. S))} \]
\[ \text{by (intro ext, simp add: SetDC-def Bind-def)} \]

\text{lemma SetDC-finite-insert:}
\[ \text{assumes fS: finite S} \]
\[ \text{and neS: S \neq \{\}} \]
\[ \text{shows SetDC a (\lambda -. insert x S) = a x \bigcap SetDC a (\lambda -. S)} \]
\[ \text{by (intro ext, simp add: SetDC-def DC-def del: Inf-image-eq)} \]

\text{proof}
\[ \text{fix ab P s} \]
\[ \text{from fS have A: finite (insert (a x ab P s) ((\lambda x. a x ab P s) \ ' S))} \]
\[ \text{and B: finite (\{\lambda x. a x ab P s\} \ ' S)) by (auto)} \]
\[ \text{from neS have C: insert (a x ab P s) ((\lambda x. a x ab P s) \ ' S) \neq \{\}} \]
\[ \text{and D: (\lambda x. a x ab P s) \ ' S \neq \{\} by (auto)} \]
\[ \text{from A C have Inf (insert (a x ab P s) ((\lambda x. a x ab P s) \ ' S)) =} \]
\[ \text{Min (insert (a x ab P s) ((\lambda x. a x ab P s) \ ' S))} \]
\[ \text{by (auto intro: Inf-eq-Min)} \]
\[ \text{also from B D have ... = min (a x ab P s) (Min (\{\lambda x. a x ab P s\} \ ' S))} \]
\[ \text{by (auto intro: Min-insert)} \]
\[ \text{also from B D have ... = min (a x ab P s) (Inf (\{\lambda x. a x ab P s\} \ ' S))} \]
\[ \text{by (simp add: Inf-eq-Min del: Inf-image-eq)} \]
\[ \text{finally show Inf (insert (a x ab P s) ((\lambda x. a x ab P s) \ ' S)) =} \]
\[ \text{min (a x ab P s) (Inf (\{\lambda x. a x ab P s\} \ ' S))} \]
\[ \text{by (simp add: Inf-eq-Min del: Inf-image-eq)} \]

\text{qed}

\text{lemma SetDC-singleton:}
\[ \text{SetDC a (\lambda -. \{x\}) = a x} \]
\[ \text{by (simp add: SetDC-def)} \]

\text{lemma cts-wp-SetDC-const:}
\[ \text{fixes s: \{a \to \} s prog} \]
\[ \text{assumes ca: } \forall x. x \in S \implies \text{bd-cts (wp (a x))} \]
\[ \text{and ha: } \forall x. x \in S \implies \text{bd-cts (wp (a x))} \]
\[ \text{and fS: finite S} \]
\[ \text{and neS: S \neq \{\}} \]
\[ \text{shows bd-cts (wp (SetDC a (\lambda -. S)))} \]
\[ \text{proof} \]
\[ \text{have finite S \implies S \neq \{\} \implies} \]
(\forall x \in S. \text{bd-cts}(wp\,(a\,x))) \rightarrow
(\forall x \in S. \text{healthy}(wp\,(a\,x))) \rightarrow
\text{bd-cts}(wp\,(\text{SetDC}\,a\,(\lambda-.\,S)))

\text{proof} (\text{induct } S \text{ rule:finite-induct, simp, clarsimp})
\begin{align*}
\text{fix } & x::'a \text{ and } F::'a \text{ set } \\
\text{assume } & F: \text{ finite } F \\
\text{and } & IH: F \neq \{\} \implies \text{bd-cts}(wp\,(\text{SetDC}\,a\,(\lambda-.\,F))) \\
\text{and } & cax: \text{bd-cts}(wp\,(a\,x)) \\
\text{and } & hax: \text{healthy}(wp\,(a\,x)) \\
\text{and } & haF: \forall x \in F. \text{healthy}(wp\,(a\,x))
\end{align*}
\text{show } \text{bd-cts}(wp\,(\text{SetDC}\,a\,(\lambda-.\,\text{insert}\,x\,F)))
\text{proof} (\text{cases } F = \{\}, \text{ simp add: SetDC-singleton cax})
\text{assume } F \neq \{\} \\
\text{with } fF \text{ cax} \text{ hax} \text{ haF} \text{ IH} \text{ show } \text{bd-cts}(wp\,(\text{SetDC}\,a\,(\lambda-.\,\text{insert}\,x\,F)))
\text{by} (\text{auto intro!: cts-wp-DC healthy-intros simp: SetDC-finite-insert})
\text{qed}
\text{qed with assms show } \text {?thesis by (auto)}
\text{qed}

\text{lemma} \ cts-wp-SetDC:
\text{fixes } a::'a \Rightarrow 's \text{ prog }
\text{assumes ca: } \text{\lambda x s. } x \in S \implies \text{bd-cts}(wp\,(a\,x)) \\
\text{and } ha: \text{\lambda x s. } x \in S \implies \text{healthy}(wp\,(a\,x)) \\
\text{and } fS: \text{\lambda s. finite}(S\,s) \\
\text{and } neS: \text{\lambda s. } S\,s \neq \{\}
\text{shows } \text{bd-cts}(wp\,(\text{SetDC}\,a\,S))
\text{proof} -
\text{from assms have } \text{bd-cts}(wp\,(\text{Bind}\,S\,(\lambda S.\,\text{SetDC}\,a\,(\lambda-.\,S))))
\text{by (iprover intro!: cts-wp-Bind cts-wp-SetDC-const)}
\text{thus } \text {?thesis by (simp add: wp-SetDC-Bind[symmetric])}
\text{qed}

\text{lemma} \ cts-wp-repeat:
\text{bd-cts}(wp\,a) \implies \text{healthy}(wp\,a) \implies \text{bd-cts}(wp\,(\text{repeat}\,n\,a))
\text{by (induct } n, \text{ auto intro!: cts-wp-Skip cts-wp-Seq healthy-intros)}

\text{lemma} \ cts-wp-Embed:
\text{bd-cts } t \implies \text{bd-cts}(wp\,(\text{Embed}\,t))
\text{by (simp add: wp-eval)}

4.3.2 Continuity of a Single Loop Step

A single loop iteration is continuous, in the more general sense defined above
for transformer transformers.

\text{lemma} \ cts-wp-loopstep:
\text{fixes } body::'s \text{ prog }
\text{assumes hl: healthy}(wp\,body) \\
\text{and } cb: \text{bd-cts}(wp\,body)
4.3. CONTINUITY

shows \( bd-cts-tr \ (Ax. \ wp \ (body \ ∘ Embed \ x \ ∈ \ G \oplus \ Skip)) \) (is \( bd-cts-tr \ ?F\))

proof\( (\text{rule } bd-cts-tr! \), \text{rule } le-trans-antisym)\)

fix \( M::nat \Rightarrow 's \text{ trans and } b::\text{real} \)

assume \( \text{chain}: \bigwedge i. \text{le-trans} (M \ i) (M \ (Suc \ i)) \)

and \( \text{fM}: \bigwedge i. \text{feasible} (M \ i) \)

show \( f:\text{le-trans} (\text{Sup-trans} \ \text{range} \ (?F \ o \ M)) \) (?F \ (\text{Sup-trans} \ \text{range} \ M))

proof\( (\text{rule } le-transI[\text{OF } \text{Sup-trans-least2}], \text{clarsimp})\)

fix \( P Q::'s \text{ expect and } t \)

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

assume \( nQ: \text{nneg } Q \text{ and } bP: \text{bounded-by (bound-of } P) \) \( Q \)

hence \( sQ: \text{sound } Q \) by(auto)

from \( fM \) have \( f\text{Sup}: \text{feasible} (\text{Sup-trans} \ \text{range} \ M) \)

(by(auto intro:feasible-Sup-trans)

from \( sQ fM \) have \( M \ t Q \vdash \text{Sup-trans} \ \text{range} \ M) \ Q \)

by(auto intro:Sup-trans-upper2)

moreover from \( sQ fM \text{fSup} \)

have \( sMtP: \text{sound } (M \ t) \text{ sound (Sup-trans} \ \text{range} \ M) \ Q \) by(auto)

ultimately have \( wp \) body \((M \ t) \vdash wp \) body \((\text{Sup-trans} \ \text{range} \ M) \ Q) \)

using \( \text{healthy-monoD}[\text{OF } \text{hs}] \) by(auto)

hence \( \bigwedge s. \text{wp body} (M \ t) \ s \leq \text{wp body} (\text{Sup-trans} \ \text{range} \ M) \ Q \) \( s \)

by(\text{rule } \text{le-funD})

thus \( ?F \ (M \ t) \ Q \vdash ?F \ (\text{Sup-trans} \ \text{range} \ M)) \ Q \)

by(\text{intro le-fun1}; simp add:wp-eval mult-left-mono)

show \( \text{nneg} (wp \ (body \ ∘ Embed \ (\text{Sup-trans} \ \text{range} \ M)) \circ G \oplus \ Skip)) \ Q) \)

proof\( (\text{rule } \text{nnegI}; \text{simp add:wp-eval})\)

fix \( s::'s \)

from \( f\text{Sup} \) have \( \text{sound} (\text{Sup-trans} \ \text{range} \ M) \ Q \) by(auto)

with \( \text{hs} \) have \( \text{sound} (wp \) body \((\text{Sup-trans} \ \text{range} \ M) \ Q)) \) by(auto)

hence \( \theta \leq wp \) body \((\text{Sup-trans} \ \text{range} \ M) \ Q) \( s \) by(auto)

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

ultimately show \( \theta \leq (G\ s\star \text{wp body} (\text{Sup-trans} \ \text{range} \ M) \ Q) \ s + (1 - (G\ s)) \ Q \ s \)

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

qed

next

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

thus \( \text{nneg } P \text{ bounded-by (bound-of } P) \) \( P \) by(auto)

show \( \forall u\in\text{range} \ ((Ax. \ wp \ (body \ ∘ Embed \ x \ ∈ \ G \oplus \ Skip)) \circ M)). \)

\( \forall R. \text{nneg } R \land \text{bounded-by (bound-of } P) \) \( R \longrightarrow \text{nneg (u } R) \land \text{bounded-by (bound-of } P) \) \( (u \ R) \)

proof\( (\text{clarsimp}; \text{intro conj1} \ \text{nnegI} \text{ bounded-byI}; \text{clarsimp}; \text{simp add:wp-eval})\)

fix \( u::\text{nat and } R::'s \text{ expect and } s::'s \)

assume \( nR: \text{nneg } R \) and \( bR: \text{bounded-by (bound-of } P) \) \( R \)

hence \( sR: \text{sound } R \) by(auto)

with \( fM \) have \( sMfR: \text{sound } (M \ u \ R) \) by(auto)

with \( \text{hs} \) have \( \text{sound} (wp \) body \((M \ u \ R)) \) by(auto)
hence \(0 \leq \text{wp body } (M \sqcup R)\) \(s\) \text{ by(auto)}

moreover from \(\nuR\) have \(0 \leq R\) \(s\) \text{ by(auto)}

ultimately show \(0 \leq \langle G\rangle s \ast \text{wp body } (M \sqcup R)\) \(s + (1 - \langle G\rangle s) \ast R\) \(s\)

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

from \(\nuR bR fM\) have \(\text{bounded-by } (\text{bound-of } P) (M \sqcup R)\) \text{ by(auto)}

with \(\text{sMuR } h \text{b}\) have \(\text{bounded-by } (\text{bound-of } P) (\text{wp body } (M \sqcup R))\) \text{ by(auto)}

hence \(\text{wp body } (M \sqcup R)\) \(s \leq \text{bound-of } P\) \text{ by(auto)}

moreover from \(bR\) have \(R \leq \text{bound-of } P\) \text{ by(auto)}

ultimately have \(\langle G\rangle s \ast \text{wp body } (M \sqcup R)\) \(s + (1 - \langle G\rangle s) \ast R\) \(s \leq \langle G\rangle s \ast \text{bound-of } P + (1 - \langle G\rangle s) \ast \text{bound-of } P\)

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

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

finally show \(\langle G\rangle s \ast \text{wp body } (M \sqcup R)\) \(s + (1 - \langle G\rangle s) \ast R\) \(s \leq \text{bound-of } P\)

\text{ qed}

\text{ qed}

\text{ show } \text{le-trans } (?F (\text{Sup-trans } (\text{range } M))) \ (\text{Sup-trans } (\text{range } (?F \circ M)))

\text{ proof } \text{(rule le-transI, rule le-funI, simp add:wp-eval)}

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

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

have \(\{ t P \mid t. \ t \in \text{range } M \} = \text{range } (\lambda i. M \ i P)\)

\text{ by(blast)}

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(simp add:Sup-trans-def)}

also \{

from \(sP fM\) have \(\forall i. \text{sound } (M \ i P)\) \text{ by(auto)}

moreover from \(sP\) chain have \(\forall i. M \ i P \vdash M \ Suc \ i \ P\) \text{ by(auto)}

moreover \{

from \(sP\) have \(\text{bounded-by } (\text{bound-of } P)\) \(P\) \text{ by(auto)}

with \(sP fM\) have \(\forall i. \text{bounded-by } (\text{bound-of } P) \ (M \ i P)\) \text{ by(auto)}

\}

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(subst bd-ctsD[OF cb], auto simp:o-def)}

\}

also have \(\text{Sup-exp } (\text{range } (\lambda i. \text{wp body } (M \ i P)))\) \(s = \text{Sup } \{ f s \mid f. \ f \in \text{range } (\lambda i. \text{wp body } (M \ i P))\}\)

\(s\)

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

finally have \(\langle G\rangle s \ast \text{wp body } (\text{Sup-trans } (\text{range } M) \ P)\) \(s + (1 - \langle G\rangle s) \ast P\)

\(s\)

\text{ by(simp)}

also \{

from \(sP fM\) have \(\forall i. \text{sound } (M \ i P)\) \text{ by(auto)}

moreover from \(sP fM\) have \(\forall i. \text{bounded-by } (\text{bound-of } P) \ (M \ i P)\) \text{ by(auto)}

\}

also have \(\langle G\rangle s \ast \text{wp body } (\text{Sup-trans } (\text{range } M) \ P)\) \(s + (1 - \langle G\rangle s) \ast P\)

\(s\)
ultimately have $\forall i. \text{bounded-by} (\text{bound-of} P) (\text{wp body} (M \ i \ P))$ using \textit{hb} by(auto)

hence $\text{bound}: \forall i. \text{wp body} (M \ i \ P) \ s \leq \text{bound-of} P$ by(auto)

moreover

have $\{\{G\} \ s \ast x \ x. x \in \{f \ s \ | f. f \ \in \ \text{range} (\lambda i. \text{wp body} (M \ i \ P))\}\} =
\{\{G\} \ s \ast f \ s \ | f. f \ \in \ \text{range} (\lambda i. \text{wp body} (M \ i \ P))\}$
by(blast)

ultimately

have $\{G\} \ s \ast \text{Sup} \ \{f \ s \ | f. f \ \in \ \text{range} (\lambda i. \text{wp body} (M \ i \ P))\} =
\text{Sup} \ \{\{G\} \ s \ast f \ s \ | f. f \ \in \ \text{range} (\lambda i. \text{wp body} (M \ i \ P))\}$
by(subst cSup-mult, auto)

moreover

have $\{x + (1\ast G) \ s \ast P \ s \ x. x \in \{\{G\} \ s \ast f \ s \ | f. f \ \in \ \text{range} (\lambda i. \text{wp body} (M \ i \ P))\}\} =
\{\{G\} \ s \ast f \ s \ + (1\ast \{G\}) \ s \ast P \ s \ | f. f \ \in \ \text{range} (\lambda i. \text{wp body} (M \ i \ P))\}$
by(blast)

moreover from \textit{bound sP} have $\forall i. \{G\} \ s \ast \text{wp body} (M \ i \ P) \ s \leq \text{bound-of} P$

by(cases $G$ s, auto)

ultimately

have $\text{Sup} \ \{\{G\} \ s \ast f \ s \ | f. f \ \in \ \text{range} (\lambda i. \text{wp body} (M \ i \ P))\} + (1\ast \{G\}) \ s \ast P \ s =
\text{Sup} \ \{\{G\} \ s \ast f \ s \ + (1\ast \{G\}) \ s \ast P \ s \ | f. f \ \in \ \text{range} (\lambda i. \text{wp body} (M \ i \ P))\}$
by(subst cSup-add, auto)

} ultimately

have $\{G\} \ s \ast \text{Sup} \ \{f \ s \ | f. f \ \in \ \text{range} (\lambda i. \text{wp body} (M \ i \ P))\} + (1\ast \{G\}) \ s \ast P \ s =
\text{Sup} \ \{\{G\} \ s \ast f \ s \ + (1\ast \{G\}) \ s \ast P \ s \ | f. f \ \in \ \text{range} (\lambda i. \text{wp body} (M \ i \ P))\}$
by(simp)

} also

have $\forall i. \{G\} \ s \ast \text{wp body} (M \ i \ P) \ s + (1\ast \{G\}) \ s \ast P \ s =
(\lambda x. \text{wp} (\text{body} :: \text{Embed} \ x \ \{G\} \oplus \text{Skip})) \circ M \ i \ P$
by(simp add:wp-eval)

also have $\forall i. \text{bound-of} P \ s \leq \text{bound-of} P \ s \leq \text{bound-of} P$ 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) + (1\ast \{G\}) \ s \ast (\text{bound-of} P)
by(auto intro: add: mono mult-left-mono)
also have \( \ldots = \text{bound-of} \ P \) by (simp add: algebra-simps)

finally show \( \langle G \rangle s * \ wp \ \text{body} \ (M \ i \ P) \ s + (1 - \langle G \rangle s) * P \ s \leq \text{bound-of} \ P \).

qed

finally have \( \text{Sup} \ \{ s \langle G \rangle f s + (1 - \langle G \rangle s) * P s \ | f \in \text{range} \ (\lambda i. \ wp \ \text{body} (M i P)) \} \leq \text{Sup-trans} \ (\text{range} \ ((\lambda x. \ wp \ ) \ (\text{body} ;; \ \text{Embed} \ x ;; \langle G \rangle \oplus \text{Skip}) \circ M)) \ P s \)

by (blast intro: cSup-least)

also have \( \text{Sup} \ \{ s \langle G \rangle f s + (1 - \langle G \rangle s) * P s \ | f \in \text{range} \ (\lambda x. \ wp \ ) \ (\text{body} ;; \ \text{Embed} \ x ;; \langle G \rangle \oplus \text{Skip}) \circ M) \} = \text{Sup-trans} \ (\text{range} \ ((\lambda x. \ wp \ ) \ (\text{body} ;; \ \text{Embed} \ x ;; \langle G \rangle \oplus \text{Skip}) \circ M)) \ P s \)

by (simp add: Sup-trans-def Sup-exp-def)

finally show \( \langle G \rangle s * \ wp \ \text{body} \ (\text{Sup-trans} \ (\text{range} \ M) \ P) s + (1 - \langle G \rangle s) * P s \leq \text{Sup-trans} \ (\text{range} \ ((\lambda x. \ wp \ ) \ (\text{body} ;; \ \text{Embed} \ x ;; \langle G \rangle \oplus \text{Skip}) \circ M)) \ P s \).

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 (\lambda Q s. \langle G \rangle s * wp body Q s + \langle N \rangle s * P s)

proof (intro mono-transI le-funI, simp)

fix Q R::'s expect and s::'s

assume sQ: sound Q and sR: sound R and le: Q \leq R

hence wp body Q \vdash wp body R

by (rule mono-transD[OF healthy-monoD, OF hb])

thus \langle G \rangle s * wp body Q s \leq \langle G \rangle s * wp body R s

by (auto dest:le-funD intro:mult-left-mono)

qed
4.4. CONTINUITY AND INDUCTION FOR LOOPS

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:**

from assms show lfp-exp-unfold
by (blast intro: wp-loop-step-mono-trans)

**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:**

from uQ hb have uwQ: unitary (wp body Q) by (auto)
with uP have 0 ≤ wp body Q s 0 ≤ P s by (auto)
thus 0 ≤ «G» s * wp body Q s + «N G» s * P s by (auto intro: add-nonneg-nonneg multi-nonneg-nonneg)

from uP uwQ have wp body Q s ≤ 1 P s ≤ 1 by (auto)

hence «G» s * wp body Q s + «N G» s * P s ≤ «G» s * 1 + «N G» s * 1
by (blast intro: add-mono multi-left-mono)
also have ... = 1 by (simp add: negate-embed)
finally show «G» s * wp body Q s + «N G» s * P s ≤ 1.

**Qed**

**Lemma lfp-loop-unitary:**

fixes body::'s prog
assumes hb: healthy (wp body)
and uP: unitary P
shows unitary (lfp-exp (λQ s. «G» s * wp body Q s + «N G» s * P s))

using assms by (blast intro: lfp-exp-unitary wp-loop-step-unitary)
CHAPTER 4. THE PGCL LANGUAGE

From the lattice structure on transformers, we establish a transfinite induction principle for loops. We use this to show a number of properties, particularly subdistributivity, for loops. This proof follows the pattern of lemma lfp_ordinal_induct in HOL/Inductive.

**Lemma loop-induct:**

```plaintext
lemma loop-induct:
  fixes body :: 's prog
  assumes hwp: healthy (wp body)
  and hwlp: nearly-healthy (wlp body)
— The body must be healthy, both in strict and liberal semantics.
  and Limit: \forall S. \[ \forall x \in S. P (fst x) (snd x); \forall x \in S. feasible (fst x); \forall x \in S. \forall Q. unitary Q \rightarrow unitary (snd x Q) \] \implies
— The property holds at limit points.
  and IH: \forall t u. \[ \forall Q. unitary Q \implies unitary (u Q) \] \implies
— The inductive step. The property is preserved by a single loop iteration.
  and P-equiv: \forall t t' u u'. \[ \forall Q. unitary Q \implies unitary (u Q) \] \implies
— The property must be preserved by equivalence
  shows P (wp (do G \rightarrow body od)) (wlp (do G \rightarrow body od))
— The property can refer to both interpretations simultaneously. The unifier will happily apply the rule to just one or the other, however.
```

**Proof (simp add: wp-eval)**

```plaintext
let ?X t = wp (body ;; Embed t « G » ⊕ Skip)
let ?Y t = wlp (body ;; Embed t « G » ⊕ Skip)

let ?M = \{ x. P (fst x) (snd x) \land
  feasible (fst x) \land
  (\forall Q. unitary Q \implies unitary (snd x Q)) \land
  le-trans (fst x) (lfp-trans ?X) \land
  le-utrans (gfp-trans ?Y) (snd x) \}

have fSup: feasible (Sup-trans (fst ' ?M))
proof (intro feasibleI bounded-byI2 nnegI2)
  fix Q::'s expect and b::real
  assume nQ: nneg Q and bQ: bounded-by b Q
  show Sup-trans (fst ' ?M) Q ⊢ \lambda s. b
    unfolding Sup-trans-def
    using nQ bQ by (auto intro!: Sup-exp-least)
  show \lambda s. 0 ⊢ Sup-trans (fst ' ?M) Q
    proof (cases)
      assume empty: ?M = {}
      show ?thesis by (simp add: Sup-trans-def Sup-def empty)
    next
      assume ne: ?M ≠ {}
      hence \exists x. x ∈ ?M by (rule nonempty-witness)
      then obtain x where xin: x ∈ ?M by (rule exE)
      hence ffix: feasible (fst x) by (simp)
```
with \( nQ \ bQ \) have \( \lambda s. 0 \vdash \text{fst } x \ Q \) by(auto)
also from \( \text{xin} \) have \( \text{fst } x \ Q \vdash \text{Sup-trans} \ (\text{fst } \ ?M) \ Q \)
   apply(intro \text{Sup-trans-upper2}\{\text{OF imageI} - nQ \ bQ\}, \text{assumption})
   apply(clarsimp, blast intro: \text{sound-nneg}\{\text{OF feasible-sound}\} \text{feasible-boundedD})
done
finally show \( \lambda s. 0 \vdash \text{Sup-trans} \ (\text{fst } \ ?M) \ Q \).

qed

have \( u\text{Inf}: \forall P. \text{unitary } P \implies \text{unitary} \ (\text{Inf-utrans} \ (\text{snd } \ ?M) \ P) \)
proof(cases \( ?M = \{\} \))
  fix \( P \)
  assume empty: \( ?M = \{\} \)
  show \( \exists \ \text{thesis } P \) by(simp only:empty, simp add: \text{Inf-utrans-def})
next
  fix \( P::\'s \text{ expect} \)
  assume uP: \( \text{unitary } P \)
  and ne: \( ?M \neq \{\} \)
  show \( \exists \ \text{thesis } P \)
proof(intro unitaryI2 nneqI2 bounded-byI2)
  from \text{nonempty-witness}\{\text{OF ne}\} obtain \( x \) where \( \text{xin: } x \in \ ?M \) by(iprover)
  hence \( \text{xin: } \text{snd } x \in \text{snd } \ ?M \) by(simp)
  hence \( \text{le-utrans} \ (\text{Inf-utrans} \ (\text{snd } \ ?M)) \ (\text{snd } x) \)
    by(intro \text{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 \text{Inf-exp-greatest})
  from \( \text{sxin} \) show \( \{t \ P \mid t. \ t \in \text{snd } \ ?M\} \neq \{\} \) by(auto)
  show \( \forall P\in\{t \ P \mid t. \ t \in \text{snd } \ ?M\}. \lambda s. 0 \vdash P \)
proof(clarsimp)
    fix \( t::\'s \text{ trans} \)
    assume \( \forall Q. \text{unitary } Q 
    \implies \text{unitary} \ (t \ Q) \)
    with uP have \( \text{unitary} \ (t \ P) \) by(auto)
    thus \( \lambda s. 0 \vdash t \ P \)
  qed
  qed
  also {
    from \( \text{ne} \) 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 \( \forall t, \emptyset \vdash \text{Inf-utrans (snd ' ?M)} \ P \).

\( \text{qed} \)

\( \text{qed} \)

have wp-loop-mono: \( \forall t, u. [\text{le-trans } t u; \forall P. \text{sound } P \Rightarrow \text{sound } (t P); \]
\( \forall P. \text{sound } P \Rightarrow \text{sound } (u P)] \Rightarrow \text{le-trans } (?X t) (?X u) \)

proof(intro le-transI le-funI, simp add:wp-eval)

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

assume le: \( \text{le-trans } t u \)

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

and su: \( \forall P. \text{sound } P \Rightarrow \text{sound } (u P) \)

and sp: \( \text{sound } P \)

have wp body \( (t P) \) \( \vdash \vdash \text{wp body } (u P) \)

proof(auto)

hence \( \text{wp body } (t P) \) \( s \leq \text{wp body } (u P) \) by(auto)

thus \( \text{G'} s * \text{wp body } (t P) \) \( s \leq \text{G'} s * \text{wp body } (u P) \) by(auto intro:mult-left-mono)

\( \text{qed} \)

have wlp-loop-mono: \( \forall t, u. [\text{le-utrans } t u; \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P); \]
\( \forall P. \text{unitary } P \Rightarrow \text{unitary } (u P)] \Rightarrow \text{le-utrans } (?Y t) (?Y u) \)

proof(intro le-utransI le-funI, simp add:wp-eval)

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

assume le: \( \text{le-utrans } t u \)

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

and su: \( \forall P. \text{unitary } P \Rightarrow \text{unitary } (u P) \)

and sp: \( \text{unitary } P \)

hence \( \text{unitary } (t P) \) \( \text{unitary } (u P) \) by(auto)

with healthy-monoD[OF hwlp] le \( sP \) have wp body \( (t P) \) \( \vdash \vdash \text{wp body } (u P) \)

by(auto)

hence \( \text{wp body } (t P) \) \( s \leq \text{wp body } (u P) \) by(auto)

thus \( \text{G'} s * \text{wp body } (t P) \) \( s \leq \text{G'} s * \text{wp body } (u P) \) by(auto intro:mult-left-mono)

\( \text{qed} \)

from hwlp have \( hX: \forall t. \text{healthy } t \Rightarrow \text{healthy } (?X t) \)

by(auto intro:healthy-intros)

from hwlp have \( hY: \forall t. \text{nearly-healthy } t \Rightarrow \text{nearly-healthy } (?Y t) \)

by(auto intro:healthy-intros)

have PLimit: \( P (\text{Sup-trans } (\text{fst } ?M)) (\text{Inf-utrans (snd } ?M)) \)

by(auto intro:Limit)

have feasible-lfp-loop:
feasible \( (\text{lfp-trans } ?X) \)

proof(intro feasibleI bounded-byI2 nnegI2,
4.4. CONTINUITY AND INDUCTION FOR LOOPS

simp-all add:wp-Loop1[unfolded wp-eval] soundI2 hwlp

fix P::'s expect and b::real
assume bP: bounded-by b P and nP: nneg P
hence sP: sound P by(auto)
show lfp-exp (λQ s. « G » s * wp body Q s + « N G » s * P s) ⊢ λs. b
proof(intro lfp-exp-upperbound le-funI)
  fix s::s
  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:
  λP. unitary P ⇒ unitary (gfp-trans ?Y P)
proof(intro unitaryI2 nnegI2 bounded-byI2,
      simp-all add:wp-Loop1[unfolded wp-eval] hwlp)
  fix P::'s 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::s
      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-left-mono)
      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
    by(auto intro:gfp-exp-least)
  qed

have fX:
  λt. feasible t ⇒ feasible (?X t)
proof(intro feasibleI nnegI bounded-byI, simp-all add:wp-eval)
  fix t::'s trans and Q::'s expect and b::real and s::s
  assume ft: feasible t and bQ: bounded-by b Q and nQ: nneg Q
  hence nneg (t Q) bounded-by b (t Q) by(auto)
  moreover hence stQ: sound (t Q) by(auto)
  ultimately have wp body (t Q) s ≤ b using hwlp by(auto)
moreover from $bQ$ have $Q s \leq b$ by(auto)
ultimately have $\langle G s \star \text{wp body} (t Q) s + (1 - \langle G s \rangle) \star Q s \leq$
    $\langle G s \rangle \diamond b + (1 - \langle G s \rangle) \star b$
    by(auto intro:add-mono mult-left-mono)
thus $\langle G s \rangle \star \text{wp body} (t Q) s + (1 - \langle G s \rangle) \star Q s \leq b$
    by(simp add:algebra-simps)
from $nQ stQ \text{hwp have } 0 \leq \text{wp body} (t Q) s \ 0 \leq Q s$ by(auto)
thus $0 \leq \langle G s \rangle \star \text{wp body} (t Q) s + (1 - \langle G s \rangle) \star Q s$
    by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)
qed

have $uY$:
    $\forall t P. (\forall P. \text{unitary } P \Longrightarrow \text{unitary} (t P)) \Longrightarrow \text{unitary } P \Longrightarrow \text{unitary} (\forall Y t P)$
proof(intro unitaryI2 nnegI bounded-byI, simp-all add:wp-eval)
  fix $t$::'s trans and $P$::'s expect and $s$::'s
  assume $ut$: $\forall P. \text{unitary } P \Longrightarrow \text{unitary} (t P)$
  and $up$: $\text{unitary } P$
  hence $utP$: $\text{unitary} (t P)$ by(auto)
  with $hwlp$ have $ubtP$: $\text{unitary} (\text{wp body} (t P))$ by(auto)
  with $up$ have $0 \leq P s \ 0 \leq \text{wp body} (t P) s$ by(auto)
  thus $0 \leq \langle G s \rangle \star \text{wp body} (t P) s + (1 - \langle G s \rangle) \star P s$
    by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)
  from $up$ $ubtP$ have $P s \leq 1 \ \text{wp body} (t P) s \leq 1$ by(auto)
  hence $\langle G s \rangle \star \text{wp body} (t P) s + (1 - \langle G s \rangle) \star P s \leq \langle G s \rangle \star 1 + (1 - \langle G s \rangle) s \star 1$
    by(blast intro:add-mono mult-left-mono)
  also have $\ldots = 1$ by(simp add:algebra-simps)
  finally show $\langle G s \rangle \star \text{wp body} (t P) s + (1 - \langle G s \rangle) \star P s \leq 1$.
  qed

have $fu$-lfp: $\text{le-trans} (\text{Sup-trans} (\text{fst} \ ?M)) (\text{lfp-trans} \ ?X)$
  using feasible-nnegD[OF feasible-lfp-loop]
  by(intro le-transI[OF Sup-trans-least2], blast+)
  hence $\text{le-trans} (\forall X (\text{Sup-trans} (\text{fst} \ ?M))) (\forall X (\text{lfp-trans} \ ?X))$
    by(auto intro:wp-loop-mono feasible-sound[OF fSup]
        feasible-sound[OF feasible-lfp-loop])
  also have $\text{equiv-trans} \ldots (\text{lfp-trans} \ ?X)$
proof(rule iffD1[OF equiv-trans-comm, OF lfp-trans-unfold], iprover intro:wp-loop-mono)
  fix $t$::'s trans and $P$::'s expect
  assume $st$: $\forall Q. \text{sound } Q \Longrightarrow \text{sound} (t Q)$
  and $sp$: $\text{sound } P$
  show $\text{sound} (\forall X t P)$
proof(intro soundI2 bounded-byI nnegI, simp-all add:wp-eval)
  fix $s$::'s
  from $sp$ $st$ $hwlp$ have $0 \leq P s \ 0 \leq \text{wp body} (t P) s$ by(auto)
  thus $0 \leq \langle G s \rangle \star \text{wp body} (t P) s + (1 - \langle G s \rangle) \star P s$
    by(blast intro:add-nonneg-nonneg mult-nonneg-nonneg)
from $sP \land \text{st}$ have bounded-by (bound-of $(t \mathcal{P})$) $(t \mathcal{P})$ by(auto)
with $sP \land \text{hwp}$ have bounded-by (bound-of $(t \mathcal{P})$) $(\mathcal{W} \text{body} (t \mathcal{P}))$ by(auto)
hence $\mathcal{W} \text{body} (t \mathcal{P}) s \leq \text{bound-of} (t \mathcal{P})$ by(auto)
moreover from $sP \land \text{hwp}$ have $P s \leq \text{bound-of} P$ by(auto)
moreover have $\succeq G s \leq 1 - \succeq G s \leq 1$ by(auto)
moreover from $sP \land \text{hwp}$ have $0 \leq \mathcal{W} \text{body} (t \mathcal{P}) s 0 \leq P s$ by(auto)
moreover have $(0 :: \text{real}) \leq 1$ by(simp)
ultimately show $\succeq G s * \mathcal{W} \text{body} (t \mathcal{P}) s + (1 - \succeq G s) * P s \leq 1 * \text{bound-of} (t \mathcal{P}) + 1 * \text{bound-of} P$
by(blast intro:add-mono mult-mono)
qed

next
let $?fp = \lambda R s. \text{bound-of} R$
show $\text{le-trans} (?X ?fp) ?fp$ by(auto intro:healthy-intros hwp)
fix $P :: s$ expect assume sound $P$
thus sound (?fp $P$) by(auto)

finally have $\text{le-lfp}: \text{le-trans} (?X (\text{Sup-trans (fst ?M)})) (\text{lfp-trans ?X})$.

have $\text{fu-gfp}: \text{le-trans} (\text{gfp-trans ?Y}) (\text{Inf-trans (snd ?M)})$
by(auto intro:Inf-trans-greatest unitary-gfp)

have $\text{equi-utrans} (\text{gfp-trans ?Y}) (?Y (\text{gfp-trans ?Y}))$
by(auto intro!:gfp-trans-unfold wlp-loop-mono uY)
also from $\text{fu-gfp}$ have $\text{le-utrans} (?Y (\text{gfp-trans ?Y})) (?Y (\text{Inf-trans (snd ?M)}))$
by(auto intro:wlp-loop-mono uInf unitary-gfp)
finally have $\text{ge-gfp}: \text{le-utrans} (\text{gfp-trans ?Y}) (?Y (\text{Inf-trans (snd ?M)}))$.
from $\text{PLimit fX uY fSup uInf}$ have $P (?X (\text{Sup-trans (fst ?M)})) (?Y (\text{Inf-trans (snd ?M)}))$
by(auto intro:uY uInf)
moreover note le-lfp ge-gfp
ultimately have $\text{pair-in}: (?X (\text{Sup-trans (fst ?M)}), ?Y (\text{Inf-trans (snd ?M)})) \in ?M$
by(simp)

have $?X (\text{Sup-trans (fst ?M)}) \in \text{fst ?M}$
by(rule image[OF pair-in, of fst, simplified])
hence $\text{le-trans} (?X (\text{Sup-trans (fst ?M)})) (\text{Sup-trans (fst ?M)})$
proof(rule le-transI[OF Sup-trans-upper2|where $t=?X (\text{Sup-trans (fst ?M)})$
and $S=$fst $?M])
fix $P :: s$ expect
assume $sP: \text{sound} P$
thus $\text{nneg} P$ by(auto)
from $sP$ show bounded-by (bound-of $P$) $P$ by(auto)
from $sP$ show $\forall u \in \text{fst ?M}. \forall Q. \text{nneg} Q \land \text{bounded-by} (\text{bound-of} P) Q \rightarrow$
\[ \text{nneg} \ (u \ Q) \land \text{bounded-by} \ (\text{bound-of} \ P) \ (u \ Q) \]

by(auto)

qed

**hence** \( \text{le-trans} \ (\text{lfp-trans} \ ?X) \ (\text{Sup-trans} \ (\text{fst} \ ?M)) \)

by(auto intro: lfp-trans-lowerbound feasible-sound[OF fSup])

**with** fu-lfp have eqt: equiv-trans (Sup-trans (fst \ ?M)) (lfp-trans \ ?X)

by(rule le-trans-antisym)

have \(?Y \ (\text{Inf-utrans} \ (\text{snd} \ ?M)) \in \text{snd} \ ?M\)

by(rule imageI[OF pair-in, of snd, simplified])

**hence** \( \text{le-utrans} \ (\text{Inf-utrans} \ (\text{snd} \ ?M)) \ (\text{inf-trans} \ ?Y) \)

by(blast intro: le-utrans-lower

**with** fu-gfp have equ: equiv-utrans (Inf-trans (snd \ ?M)) (gfp-trans \ ?Y)

by(auto intro: le-utrans-antisym)

from PLimit eqt equ show \( P \ (\text{lfp-trans} \ ?X) \ (\text{gfp-trans} \ ?Y) \) by(rule P-equiv)

qed

## 4.4.1 The Limit of Iterates

The iterates of a loop are its sequence of finite unrollings. We show shortly that this converges on the least fixed point. This is enormously useful, as we can appeal to various properties of the finite iterates (which will follow by finite induction), which we can then transfer to the limit.

**definition** iterates :: `'s prog ⇒ ('s ⇒ bool) ⇒ nat ⇒ 's trans

**where** iterates body G i = ((λx. wp (body ;; Embed x « G « Skip)) \( i \) ) (λP s. 0)

**lemma** iterates-0[simp]:

iterates body G 0 = (λP s. 0)

by(simp add: iterates-def)

**lemma** iterates-Suc[simp]:

iterates body G (Suc i) = wp (body ;; Embed (iterates body G i) « G«Skip)

by(simp add: iterates-def)

All iterates are healthy.

**lemma** iterates-healthy:

healthy (wp body) \(⇒\) healthy (iterates body G i)

by(induct i, auto intro: healthy-intros)

The iterates are an ascending chain.

**lemma** iterates-increasing:

fixes body: `'s prog

**assumes** hb: healthy (wp body)

**shows** le-trans (iterates body G i) (iterates body G (Suc i))

**proof** (induct i)
4.4. CONTINUITY AND INDUCTION FOR LOOPS

show le-trans (iterates body G 0) (iterates body G (Suc 0))
proof(simp add:iterates-def, rule le-trans)
fix P::'s expect
assume sound P
with hb have sound (wp (body ;; Embed (λP s. 0) « G s ⊕ Skip) P)
by(auto intro: wp-loop-step-sound)
thus λs. 0 ⊢ wp (body ;; Embed (λP s. 0) « G s ⊕ Skip) P
by(auto)
qed

fix i
assume IH: le-trans (iterates body G i) (iterates body G (Suc i))
have equiv-trans (iterates body G (Suc i))
(wp (body ;; Embed (iterates body G i) « G s ⊕ Skip))
by(simp)
also from iterates-healthy[OF hb]
have le-trans ...
(wp (body ;; Embed (iterates body G (Suc i)) « G s ⊕ Skip))
by(black intro: wp-loop-step-mono[OF hb IH])
also have equiv-trans ...
(iterates body G (Suc (Suc i)))
by(simp)
finally show le-trans (iterates body G (Suc i)) (iterates body G (Suc (Suc i)))
qed

lemma wp-loop-step-bounded:
fixes t::'s trans and Q::'s expect
assumes nQ: nneg Q
and bQ: bounded-by b Q
and ht: healthy t
and hb: healthy (wp body)
shows bounded-by b (wp (body ;; Embed t « G s ⊕ Skip) Q)
proof(rule bounded-byI, simp add:wp-eval)
fix s::'
from nQ bQ have sQ: sound Q by(auto)
with bQ ht have sound (t Q) bounded-by b (t Q) by(auto)
with hb have bounded-by b (wp body (t Q)) by(auto)
with bQ have wp body (t Q) s ≤ b Q s ≤ b by(auto)
hence «G» s * wp body (t Q) s + (1−«G» s) s ≤ «G» s * b + (1−«G» s) s * b
by(auto intro:add-mono mult-left-mono)
also have ... = b by(simp add:algebra-simps)
finally show «G» s * wp body (t Q) s + (1−«G» s) s * Q s ≤ b
qed

This is the key result: The loop is equivalent to the supremum of its iterates.
This proof follows the pattern of lemma continuous_lfp in HOL/Library/Continuity.

lemma lfp-iterates:
fixes body::'s prog
assumes hb: healthy (wp body)
and cb: bd-cts (wp body)
shows equiv-trans \((\text{wp (do } G \to body \odot)) (\text{Sup-trans (range (iterates body } G)))\) (is equiv-trans ?X ?Y)

**proof** (rule le-trans-antisym)

let \(\varphi = \lambda x. \text{wp (body ;; Embed } x \prec G \triangleright \text{Skip)}\)

let \(\varphi_0 = \lambda (P::s \Rightarrow \text{real}) s::s. 0::\text{real}\)

have \(\varphi : \\forall i. \text{healthy } ((\varphi \triangleleft i) \varphi_0)\)
**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 \(\varphi (\text{Sup-trans (range (iterates body } G)))\)

by (auto intro: feasible-Sup-trans)

{ fix \(i\)

have le-trans \((\varphi \triangleleft i) \varphi_0\) \(\varphi\)

**proof** (induct \(i\))

show le-trans \((\varphi \triangleleft 0) \varphi_0\) \(\varphi\)

**proof** (simp, intro le-transI)

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

with hb healthy-wp-loop

have sound \((\text{wp (mu } x. \text{body ;; Embed } x \prec G \triangleright \text{Skip}) \, P)\)

by (auto)

thus \(\lambda s. 0 \vdash \text{wp (mu } x. \text{body ;; x } \prec G \triangleright \text{Skip}) \, P\)

by (auto)

qed

fix \(i\)

assume \(\varphi ; s\Rightarrow \text{sound } P\)

with hb healthy-wp-loop

show sound \((\text{wp (mu } x. \text{body ;; x } \prec G \triangleright \text{Skip}) \, P)\)

by (auto)

from \(\varphi \Rightarrow \text{sound } ((\varphi \triangleleft i) \varphi_0)\)

by (rule healthy-sound [OF hf])

qed

also { from hb have X: le-trans \((\text{wp (body ;; Embed } (\lambda P \, s. \text{bound-of } P) \prec G \triangleright \text{Skip}))\)

(\lambda P \, s. \text{bound-of } P) \)

by (intro le-transI, simp add: wp-eval, auto intro: lfp-loop-fp unfolded
4.4. CONTINUITY AND INDUCTION FOR LOOPS

negate-embed]
  have equiv-trans (?F ?X) ?X
  unfolding wp-eval
  by (intro ifD1 [OF equiv-trans-comm, OF lfp-trans-unfold]
      wp-loop-step-mono [OF hb] wp-loop-step-sound [OF hb], (blast | rule X)+)
}

} finally show le-trans ((?F ^^ (Suc i)) ?bot) ?X .
qed

hence \( \forall i. \) le-trans (iterates body G i) (wp do G \( \rightarrow \) body od)
thus le-trans ?Y ?X
by (auto simp add: iterates-def)
thus show le-trans ?X ?Y
unfolding wp-eval
proof (rule lfp-trans-lowerbound)
from hb cb have bd-cts-tr ?F
by (rule cts-wp-loopstep)
with iterates-increasing [OF hb] iterates-healthy [OF hb]
have equiv-trans (?F ?Y) (Sup-trans (range (?F o (iterates body G))))
by (auto intro!: healthy-feasibleD bd-cts-trD)
also have le-trans (Sup-trans (range (?F o (iterates body G)))) ?Y
proof (rule le-transI)
fix P :: "\( \vdash \) expect\)
assume sP: sound P
show (Sup-trans (range (?F o (iterates body G)))) P \( \vdash \) ?Y P
proof (rule Sup-trans-least2, clarsimp)
show \( \forall u \in \text{range} \ (\lambda x. \ wp (\text{body @} \text{Embed} \ x @} G \oplus \text{Skip})) \circ \text{iterates body} \ G) .
  \( \forall R. \) nneg R \( \land \) bounded-by (bound-of P) R \( \rightarrow \)
nneg (u R) \( \land \) bounded-by (bound-of P) (u R)
proof (clarsimp, intro conjI)
fix Q:: expect and i
assume nQ: nneg Q and bQ: bounded-by (bound-of P) Q
hence sound Q by (auto)
moreover from iterates-healthy [OF hb]
have \( \forall P. \) sound P \( \rightarrow \) sound (iterates body G i P) by (auto)
moreover note hb
ultimately have sound (wp (body @} Embed (iterates body G i) \( \oplus \) Skip) Q)
  by (iprover intro: wp-loop-step-sound)
thus nneg (wp (body @} Embed (iterates body G i) \( \oplus \) 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) \( \oplus \) Skip) Q)
  by (auto)
\begin{quote}
\texttt{« \textit{G} » ⊕ \texttt{Skip}) \text{ Q}}
\text{by (rule \textit{wp-loop-step-bounded})}
\text{qed}
\text{from \texttt{sP show \textit{nneg P bounded-by (bound-of \textit{P}) \texttt{P by(auto)}}}}
\text{next}
\text{fix Q:\hspace{1em}‘s expect}
\text{assume nQ: \textit{nneg Q and bQ: bounded-by (bound-of \textit{P}) \textit{Q}}}
\text{hence sound \textit{Q by(auto)}}
\text{with \texttt{fSup have sound (\textit{Sup-trans (range (iterates body \textit{G}) \textit{Q)} by(auto)}}
\text{thus \textit{nneg (Sup-trans (range (iterates body \textit{G}) \textit{Q)} by(auto)}}
\text{fix i show \texttt{wp (body ;; Embed (iterates body \textit{G i}) « \textit{G} » ⊕ \texttt{Skip}) \text{ Q ⊢ \text{Sup-trans (range (iterates body \textit{G}) \textit{Q}}\text{)}}
\text{proof (rule \textit{Sup-trans-upper2 [OF - - nQ bQ]}})
\text{from \texttt{iterates-healthy[OF hh]}}
\text{show \texttt{∀ u∈range (iterates body \textit{G}).}}
\texttt{∀ R. \textit{nneg R ∧ bounded-by (bound-of \textit{P}) \textit{R → nneg (u R) ∧ bounded-by (bound-of \textit{P}) (u R)}}}
\text{by (auto)}
\text{have \texttt{wp (body ;; Embed (iterates body \textit{G i}) « \textit{G} » ⊕ \texttt{Skip}) = iterates body \textit{G (Suc i)}}
\text{by (simp)}
\text{also have ... \texttt{∈ range (iterates body \textit{G})}}
\text{by (blast)}
\text{finally show \texttt{wp (body ;; Embed (iterates body \textit{G i}) « \textit{G} » ⊕ \texttt{Skip}) ∈ \text{range (iterates body \textit{G})}}
\text{qed}
\text{qed}
\text{finally show \texttt{le-trans (?F \textit{?Y} \textit{?Y}} \text{)}
\text{fix \texttt{P:\hspace{1em}‘s expect}}
\text{assume sound \textit{P}}
\text{with \texttt{fSup show sound (?Y \textit{P)) by(auto)}}
\text{qed}
\text{qed}
\end{quote}

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{quote}
\texttt{fixes body:\hspace{1em}‘s prog}
\texttt{assumes hb: healthy (wp body)}
\texttt{and cb: bd-cts (wp body)}
\texttt{and sP: sound \textit{P}}
\texttt{shows (λi. iterates body \textit{G i P s) →→\textit{ wp (do G → body od) P s}})
\text{proof –}
\texttt{let \textit{?X = {f s \mid f, f \in \{t P \mid t ∈ range (iterates body \textit{G})\}}}}
\texttt{have closure-Sup: Sup \textit{?X ∈ closure ?X}
\end{quote}
proof (rule closure-contains-Sup, simp, clarsimp)
  fix i
  from sP have bounded-by (bound-of P) P by (auto)
  with iterates-healthy[of hb] sP have \( \forall j. \) bounded-by (bound-of P) (iterates body G i P)
    by (auto)
  thus iterates body G i P s \( \leq \) bound-of P by (auto)
  qed

have (\( \lambda i. \) iterates body G i P s) \( \longrightarrow \) Sup \( \{ f s \mid f. f \in \{ t P \mid t. t \in \text{range (iterates body G)} \} \}

proof (rule LIMSEQ-I)
  fix \( r :: \text{real} \) assume posr: \( 0 < r \)
  with closure-Sup obtain y where yin: \( y \in ?X \) and ey: dist y (Sup ?X) < r
    by (simp only: closure-approachable, blast)
  from yin obtain i where yit: \( y = \) iterates body G i P s by (auto)
  { fix j
    have i \( \leq \) j \( \longrightarrow \) le-trans (iterates body G i) (iterates body G j)
      proof (induct j, simp, clarify)
        fix k
        assume IH: \( i \leq k \longrightarrow \) le-trans (iterates body G i) (iterates body G k)
        and le: \( i \leq \) Suc k
        show le-trans (iterates body G i) (iterates body G (Suc k))
          proof (cases i = Suc k, simp)
            assume i \( \neq \) Suc k
            with le have i \( \leq \) k by (auto)
            with IH have le-trans (iterates body G i) (iterates body G k) by (auto)
            also note iterates-increasing[of hb]
            finally show le-trans (iterates body G i) (iterates body G (Suc k))
          qed
        qed
      qed
    with sP have \( \forall j \geq i. \) iterates body G i P s \( \leq \) iterates body G j P s
      by (auto)
    moreover {
      from sP have bounded-by (bound-of P) P by (auto)
      with iterates-healthy[of hb] sP have \( \forall j. \) bounded-by (bound-of P) (iterates body G j P)
        by (auto)
      hence \( \forall j. \) iterates body G j P s \( \leq \) bound-of P by (auto)
      hence \( \forall j. \) iterates body G j P s \( \leq \) Sup ?X
        by (intro cSup-upper bdd-aboveI, auto)
    }
    ultimately have \( \forall j. i \leq j \Longrightarrow \)
      \( \| \) (iterates body G j P s - Sup ?X) \( \leq \)
      \( \| \) (iterates body G i P s - Sup ?X)
      by (auto)
    also from ey yit have \( \| \) (iterates body G i P s - Sup ?X) < r
by(simp add:dist-real-def)

finally show \( \exists n. \forall n \geq n_0. \) norm (iterates body \( G \) \( n \) \( P \) \( s \) –
\( \sup \{ f s \mid f \in \{ t P \mid t \in \mathrm{range} \ (\text{iterates body } G) \} \} \) < \( r \)
by(auto)

qed

moreover have wp do \( G \) \( \rightarrow \) body od \( P \) \( s \) = \( \sup \{ f s \mid f \in \{ t P \mid t \in \mathrm{range} \ (\text{iterates body } G) \} \} \)
by(simp add:equiv-transD[OF lfp-iterates])

ultimately show \( \) thesis by(simp)

qed

The iterates themselves are all continuous.

lemma cts-iterates:
 fixes body::'s prog
 assumes hb: healthy (wp body)
 and cb: bd-cts (wp body)
 shows bd-cts (iterates body \( G \) \( i \))

proof(induct \( i \), simp-all)
 have range \( \lambda n::nat \) (\( s::'s \)) \( . \) \( 0::real \) = \( \{ \lambda s. 0::real \} \)
 by(auto)
 thus bd-cts \( \lambda P (s::'s). 0) \)
 by(intro bd-ctsI, simp add:o-def Sup-exp-def)

next
 fix \( i \)
 assume IH: bd-cts (iterates body \( G \) \( i \))
 thus bd-cts (wp (body ;; Embed (iterates body \( G \) \( i \)) \( ⊕ \) Skip))
 healthy-intros iterates-healthy cb hb)

qed

Therefore so is the loop itself.

lemma cts-wp-loop:
 fixes body::'s prog
 assumes hb: healthy (wp body)
 and cb: bd-cts (wp body)
 shows bd-cts (wp do \( G \) \( \rightarrow \) body od)

proof(rule bd-ctsI)
 fix \( M::nat \Rightarrow 's \) expect and \( b::real \)
 assume chain: \( \bigwedge i. M i \vdash M (Suc i) \)
 and sM: \( \bigwedge i. \) sound \( (M i) \)
 and bM: \( \bigwedge i. \) bounded-by \( b \) \( (M i) \)

from sM bM iterates-healthy[OF hb]
 have \( \bigwedge j i. \) bounded-by \( b \) \( (\text{iterates body } G \ i \ (M j)) \)
 by(blast)
 hence \( iB: \bigwedge j i s. \text{iterates body } G \ i \ (M j) \ s \leq b \)
 by(auto)
from \( sM \) \( bM \) have \( sSup: \text{sound} (\text{Sup-exp} (\text{range} M)) \)
by(auto intro;Sup-exp-sound)
with \( \text{lfp-iterates} (OF \) \( hb \) \( cb \))
have \( \text{wp do} G \rightarrow \text{body od} (\text{Sup-exp} (\text{range} M)) = \text{Sup-trans} (\text{range} (\text{iterates body} G)) (\text{Sup-exp} (\text{range} M)) \)
by(simp add:equte-transD)
also {
from chain \( sM \) \( bM \)
have \( \forall i. \text{iterates body} G \) \( i \) (\( \text{Sup-exp} (\text{range} M) \)) = \( \text{Sup-exp} (\text{range} (\text{iterates body} G \) \( i \) \( \circ \) \( M \))) \)
by(blast intro:bd-ctsD cts-iterates[OF hb cb])
hence \( \{ t \ (\text{Sup-exp} (\text{range} M)) \ | t \in \text{range} (\text{iterates body} G) \} = \{ \text{Sup-exp} (\text{range} (t \circ M)) \ | t \in \text{range} (\text{iterates body} G) \} \)
by(auto intro;sym)
hence \( \text{Sup-trans} (\text{range} (\text{iterates body} G)) \) (\( \text{Sup-exp} (\text{range} M) \)) = \( \text{Sup-exp} (\text{Sup-exp} (\text{range} (t \circ M)) \ | t \in \text{range} (\text{iterates body} G) \} \)
by(simp add:Sup-trans-def)
}
also {
have \( \forall s. \{ f s \ | \exists t. f = (\lambda s. \text{Sup} \{ f s \ | f \in \text{range} (t \circ M) \}) \} \wedge \exists t \in \text{range} (\text{iterates body} G) \) = \( \text{range} (\lambda i. \text{Sup} (\text{range} (\lambda j. \text{iterates body} G \) i \( \circ \) \( M \) \) j \) s)) \)
(is \( \forall s. \exists t s = \exists t s \) )
proof(intro antisym subsetI)
fix \( s x \)
assume \( x \in \exists t s \)
then obtain \( t \) where \( \text{rux: } x = \text{Sup} \{ f s \ | f \in \text{range} (t \circ M) \} \)
and \( t \in \text{range} (\text{iterates body} G) \) by(auto)
then obtain \( i \) where \( t = \text{iterates body} G \) \( i \) by(auto)
with \( \text{rux} \) have \( x = \text{Sup} \{ f s \ | f \in \text{range} (\lambda j. \text{iterates body} G \) i \( \circ \) \( M \) \) j \) s) \)
by(simp add:o-def)
moreover have \( \{ f s \ | f \in \text{range} (\lambda j. \text{iterates body} G \) i \( \circ \) \( M \) \) j \) s) = \( \text{range} (\lambda j. \text{iterates body} G \) i \( \circ \) \( M \) \) j \) s) by(auto)
ultimately have \( x = \text{Sup} (\text{range} (\lambda j. \text{iterates body} G \) i \( \circ \) \( M \) \) j \) s) \)
by(simp)
thus \( x \in \text{range} (\lambda i. \text{Sup} (\text{range} (\lambda j. \text{iterates body} G \) i \( \circ \) \( M \) \) j \) s)) \)
by(auto)
next
fix \( s x \)
assume \( x \in \exists t s \)
then obtain \( i \) where \( A: x = \text{Sup} (\text{range} (\lambda j. \text{iterates body} G \) i \( \circ \) \( M \) \) j \) s) \)
by(auto)

have \( \forall s. \{ f s \ | f \in \text{range} (\lambda j. \text{iterates body} G \) i \( \circ \) \( M \) \) j \) s) = \( \text{range} (\lambda j. \text{iterates body} G \) i \( \circ \) \( M \) \) j \) s) by(auto)
hence \( B: (\lambda s. \text{Sup} (\text{range} (\lambda j. \text{iterates body} G \) i \( \circ \) \( M \) \) j \) s)) = (\( \lambda s. \text{Sup} \{ f s \ | f \in \text{range} (\text{iterates body} G \) i \( \circ \) \( M \) \) s) \)
by(simp add:o-def)
have $C$: iterates body $G$ $i \in \text{range (iterates body } G)$ by(auto)

have $\exists f, x = f s \land$
\hspace{1em} $(\exists t. f = (\lambda s. \text{Sup } \{f s \mid f \in \text{range (} t \circ M)\}) \land t \in \text{range (iterates body } G))$
	by(iprover intro:A B C)
thus $x \in ?X s$ by(simp)
qed

hence $\text{Sup-exp } \{\text{Sup-exp } \{\text{range (} t \circ M)\} \mid t \in \text{range (iterates body } G)\} =$
\hspace{1em} $(\lambda s. \text{Sup } \{\text{range (} \lambda i. \text{Sup } \{\text{range (} \lambda j. \text{iterates body } G i \ (M j) s)\})\})\}$
	by(simp add: Sup-exp-def)

also have $(\lambda s. \text{Sup } \{\text{range (} \lambda i. \text{Sup } \{\text{range (} \lambda j. \text{iterates body } G i \ (M j) s)\})\}) =$
\hspace{1em} $(\lambda s. \text{Sup } \{\text{range (} \lambda (i,j). \text{iterates body } G i \ (M j) s)\})$
(is $?X = ?Y$)

proof(rule ext, rule antisym)
fix s::s
show $?Y s \leq ?X s$

proof(rule cSup-least, blast, clarify)
fix i j::nat
from iB have iterates body $G$ $i \ (M j) s \leq \text{Sup } \{\text{range (} \lambda j. \text{iterates body } G i \ (M j) s)\}$
	by(intro cSup-upper bdd-aboveI, auto)
also from iB have ... $\leq \text{Sup } \{\text{range (} \lambda i. \text{Sup } \{\text{range (} \lambda j. \text{iterates body } G i \ (M j) s)\})\}$
	by(intro cSup-upper cSup-least bdd-aboveI, (blast intro:cSup-least)+)

finally show iterates body $G$ $i \ (M j) s \leq$
\hspace{1em} $\text{Sup } \{\text{range (} \lambda i. \text{Sup } \{\text{range (} \lambda j. \text{iterates body } G i \ (M j) s)\})\}$
.qed

have $\forall i j. \text{iterates body } G i \ (M j) s \leq$
\hspace{1em} $\text{Sup } \{\text{range (} \lambda (i,j). \text{iterates body } G i \ (M j) s)\}$
	by(rule cSup-upper, auto intro:iB)
thus $?X s \leq ?Y s$
	by(intro cSup-least, blast, clarify, simp del: Sup-image-eq, blast intro:cSup-least)

qed

also have ... $= (\lambda s. \text{Sup } \{\text{range (} \lambda j. \text{Sup } \{\text{range (} \lambda i. \text{iterates body } G i \ (M j) s)\})\})$
(is $?X = ?Y$)

proof(rule ext, rule antisym)
fix s::s
have $\forall i j. \text{iterates body } G i \ (M j) s \leq$
\hspace{1em} $\text{Sup } \{\text{range (} \lambda (i,j). \text{iterates body } G i \ (M j) s)\}$
	by(rule cSup-upper, auto intro:iB)
thus $?Y s \leq ?X s$
	by(intro cSup-least, blast, clarify, simp del: Sup-image-eq, blast intro:cSup-least)

show $?X s \leq ?Y s$
proof(rule cSup-least, blast, clarify)
fix i j::nat
from iB have iterates body $G$ $i \ (M j) s \leq \text{Sup } \{\text{range (} \lambda i. \text{iterates body } G i \ (M j) s)\}$
4.4. CONTINUITY AND INDUCTION FOR LOOPS

\[(M \, j \, s)\]

by \(\text{intro cSup-upper bdd-aboveI, auto}\)
also from \(\text{iB have ...} \leq \text{Sup (range (}\lambda j. \text{Sup (range (}\lambda i. \text{iterates body G i (M j) s))}}\)

by \(\text{intro cSup-upper cSup-least bdd-aboveI, blast, blast intro:cSup-least}\)
finally show \(\text{iterates body G i (M j) s} \leq \text{Sup (range (}\lambda j. \text{Sup (range (}\lambda i. \text{iterates body G i (M j) s))}}\)

qed

qed
also { have \(\bigwedge s. \text{range (}\lambda j. \text{Sup (range (}\lambda i. \text{iterates body G i (M j) s))}} = \{f s | f \in \text{range ((}\lambda P s. \text{Sup } \{f s | \exists t. f = t P \wedge t \in \text{range (iterates body G)}\}) \circ M)\} / \{s s = ?Y s\) proof \(\text{intro antisym subsetI}\)
fix s x
assume \(x \in ?X s\)
then obtain \(j\) where \(\text{rwx: } x = \text{Sup (range (}\lambda i. \text{iterates body G i (M j) s))}\)
by \(\text{(auto)}\)
moreover { have \(\bigwedge s. \text{range (}\lambda i. \text{iterates body G i (M j) s}) = \{f s | f \in \text{range ((}\lambda P s. \text{Sup } \{f s | \exists t. f = t P \wedge t \in \text{range (iterates body G)}\}) \circ M)\} / \{s s = ?Y s\) proof \(\text{intro o-def}\)
fix s x
assume \(x \in ?Y s\)
then obtain \(P\) where \(\text{rwx: } x = P s\)
and \(\text{Pin: } P \in \text{range ((}\lambda P s. \text{Sup } \{f s | \exists t. f = t P \wedge t \in \text{range (iterates body G)}\}) \circ M)\)
by \(\text{(auto)}\)
then obtain \(j\) where \(P = (\lambda s. \text{Sup } \{f s | \exists t. f = t (M j) \wedge t \in \text{range (iterates body G)}\})\)
by \(\text{(auto)}\)
also { have \(\bigwedge s. \{f s | \exists t. f = t (M j) \wedge t \in \text{range (iterates body G)}\} = \text{range (}\lambda i. \text{iterates body G i (M j) s})\)
by \(\text{(auto)}\)
hehence \(\bigwedge s. \text{Sup } \{f s | \exists t. f = t (M j) \wedge t \in \text{range (iterates body G)}\} = (\lambda s. \text{Sup (range (}\lambda i. \text{iterates body G i (M j) s)}))\)
by \(\text{(simp)}\)
}
finally have \(x = \text{Sup (range (}\lambda i. \text{iterates body G i (M j) s))}\)
by \(\text{(simp add:rwx)}\)
thus \(x \in ?X s\) by \(\text{(simp)}\)
qed
hence \((\lambda s. \text{Sup} (\text{range} (\lambda j. \text{Sup} (\text{range} (\lambda i. \text{iterates body } G \ i \ (M \ j \ s))))) = \text{Sup-exp} (\text{range} (\text{Sup-trans} (\text{range} (\text{iterates body } G)) \ o \ M))\)
by (simp add: Sup-exp-def Sup-trans-def)
}
also have \(\text{Sup-exp} (\text{range} (\text{Sup-trans} (\text{range} (\text{iterates body } G)) \ o \ M)) = \text{Sup-exp} (\text{range} (\text{Sup-trans} (\text{iterates body } G)) \ o \ M)\)
by (simp add:o-def equie-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 =
c ts-wp-Abort ts-wp-Skip
c ts-wp-Seq cts-wp-PC
c ts-wp-DC cts-wp-Embed
c ts-wp-Apply cts-wp-SetDC
c ts-wp-SetPC cts-wp-Bind
c ts-wp-repeat

4.5 Sublinearity

theory Sublinearity imports Embedding Healthiness LoopInduction begin

4.5.1 Nonrecursive Primitives

Sublinearity of non-recursive programs is generally straightforward, and follows from the algebraic properties of the underlying operations, together with healthiness.

lemma sublinear-wp-Skip:
sublinear (wp Skip)
by (auto simp: wp-eval)

lemma sublinear-wp-Abort:
sublinear (wp Abort)
by (auto simp: wp-eval)

lemma sublinear-wp-Apply:
sublinear (wp (Apply f))
by (auto simp: wp-eval)

lemma sublinear-wp-Seq:
fixes x::'s prog
assumes slx: sublinear (wp x) and sly: sublinear (wp y)
and hx: healthy (wp x) and hy: healthy (wp y)
shows sublinear (wp (x :: y))
proof (rule sublinear1, simp add: wp-eval)
4.5. SUBLINEARITY

fix \( P : 's \Rightarrow \text{real} \) \( Q : 's \Rightarrow \text{real} \) \( s : 's \)
and \( a :: \text{real} \) \( b :: \text{real} \) \( c :: \text{real} \)
assume \( sP : \text{sound} \) \( P \) \( sQ : \text{sound} \) \( Q \)
and \( \text{nna: } 0 \leq a \) \( \text{nnb: } 0 \leq b \) \( \text{nnc: } 0 \leq c \)

with \( \text{slx hy} \) have \( a * \text{wp } x (\text{wp } y P) s + b * \text{wp } x (\text{wp } y Q) s \circ c \leq \)
\( \text{wp } x (\lambda s. a * \text{wp } y P s + b * \text{wp } y Q s \circ c) s \)
by(\text{blast intro:sublinearD})
also \{ 
from \( sP sQ \text{nna nnb nnc slx} \)
have \( \text{auto intro:sublinearD} \)
moreover from \( sP sQ \) by
have \( \text{auto intro:sublinearD} \)
moreover with \( \text{nna nnb nnc} \)
have \( \text{auto intro:sublinearD} \)
moreover with \( \text{auto intro:sublinearD} \)
moreover with \( \text{auto intro:sublinearD} \)
moreover with \( \text{auto intro:sublinearD} \)
ultimately
have \( \text{auto intro:sublinearD} \)
finally show \( a * \text{wp } x (\text{wp } y P) s + b * \text{wp } x (\text{wp } y Q) s \circ c \leq \)
\( \text{wp } x (\lambda s. a * P s + b * Q s \circ c) s \).
qed

lemma \text{sublinear-wp-PC}:
fixes \( x : 's \) prog
assumes \( \text{slx: sublinear (wp } x \text{) and slx: sublinear (wp } y \text{)} \)
and \( \text{uP: unitary } P \)
shows \( \text{sublinear (wp } x P \oplus y \text{)} \)
proof(include \text{sublinearI}, simp add:wp-eval)
fix \( R: 's \Rightarrow \text{real} \) \( Q: 's \Rightarrow \text{real} \) \( s: 's \)
and \( a :: \text{real} \) \( b :: \text{real} \) \( c :: \text{real} \)
assume \( sR : \text{sound } R \) \( sQ : \text{sound} \) \( Q \)
and \( \text{nna: } 0 \leq a \) \( \text{nnb: } 0 \leq b \) \( \text{nnc: } 0 \leq c \)

have \( a * (P s * \text{wp } x Q s + (1 - P s) * \text{wp } y Q s) + \)
\( b * (P s * \text{wp } x R s + (1 - P s) * \text{wp } y R s) \circ c = \)
\( (P s * a * \text{wp } x Q s + (1 - P s) * a * \text{wp } y Q s) + \)
\( (P s * b * \text{wp } x R s + (1 - P s) * b * \text{wp } y R s) \circ c \)
by(simp add:field-simps)
also
have ... = \( (P \ s \ast a \ast wp\ x\ Q \ s + P \ s\ast b \ast wp\ x\ R\ s) + (1 - P\ s) * a * wp\ y\ Q\ s + (1 - P\ s) * b * wp\ y\ R\ s) \odot c \)
by(simp add:ac-simps)
also 

have ... = \( P \ s * (a * wp\ x\ Q\ s + b * wp\ x\ R\ s) + (1 - P\ s) * (a * wp\ y\ Q\ s + b * wp\ y\ R\ s) \odot (P \ s * c + (1 - P\ s) * c) \)
by(simp add:field-simps)
also 

have ... \leq (P \ s * (a * wp\ x\ Q\ s + b * wp\ x\ R\ s) \odot P \ s * c) + (1 - P\ s) * (a * wp\ y\ Q\ s + b * wp\ y\ R\ s) \odot (1 - P\ s) * c)
by(rule tminus-add-mono)
also 

from \( uP \) have 0 \leq P\ s\ and\ 0 \leq 1 - P\ s 
by(auto simp:sign-simps)
hence \( (P \ s * (a * wp\ x\ Q\ s + b * wp\ x\ R\ s) \odot P \ s * c) + ((1 - P\ s) * (a * wp\ y\ Q\ s + b * wp\ y\ R\ s) \odot (1 - P\ s) * c) = (P \ s * (a * wp\ x\ Q\ s + b * wp\ x\ R\ s \odot c) + (1 - P\ s) * (a * wp\ y\ Q\ s + b * wp\ y\ R\ s \odot c) \)
by(simp add:tminus-left-distrib)
also 

from \( sQ\ sR\ nna\ nnb\ nnc\ slx\) 

have \( a * wp\ x\ Q\ s + b * wp\ x\ R\ s \odot c \leq wp\ x\ (\lambda s. a * Q\ s + b * R\ s \odot c) \ s \)
by(blast)
moreover 

from \( sQ\ sR\ nna\ nnb\ nnc\ sly\) 

have \( a * wp\ y\ Q\ s + b * wp\ y\ R\ s \odot c \leq wp\ y\ (\lambda s. a * Q\ s + b * R\ s \odot c) \ s \)
by(blast)
moreover 

from \( uP \) have 0 \leq P\ s\ and\ 0 \leq 1 - P\ s 
by(auto simp:sign-simps)
ultimately 

have \( P \ s * (a * wp\ x\ Q\ s + b * wp\ x\ R\ s \odot c) + (1 - P\ s) * (a * wp\ y\ Q\ s + b * wp\ y\ R\ s \odot c) \leq P \ s \ast wp\ x\ (\lambda s. a \ast Q\ s + b \ast R\ s \odot c) \ s + (1 - P\ s) * wp\ y\ (\lambda s. a \ast Q\ s + b \ast R\ s \odot c) \ s \)
by(blast intro:add-mono mult-left-mono)
}

finally 

show \( a * (P \ s \ast wp\ x\ Q\ s + (1 - P\ s) * wp\ y\ Q\ s) + b * (P \ s \ast wp\ x\ R\ s + (1 - P\ s) * wp\ y\ R\ s) \odot c \leq P \ s \ast wp\ x\ (\lambda s. a \ast Q\ s + b \ast R\ s \odot c) \ s + (1 - P\ s) * wp\ y\ (\lambda s. a \ast Q\ s + b \ast R\ s \odot c) \ s \).
4.5. **SUBLINEARITY**

```plaintext
fixes x :: 's prog
assumes slx: sublinear (wp x) and sly: sublinear (wp y)
shows sublinear (wp (x ∪ y))
proof (rule sublinearI, simp only: wp-eval)
  fix R :: 's real and Q :: 's real and s :: 's
  assume sound R: 0 ≤ a and sound Q: 0 ≤ b and nnc: 0 ≤ c
  from nna nnb
  have a * min (wp x Q s) (wp y Q s) +
    b * min (wp x R s) (wp y R s) ⊓ c =
    min (a * wp x Q s) (a * wp y Q s) +
    min (b * wp x R s) (b * wp y R s) ⊓ c
    by (simp add: min-distrib)
  also
  have ... ≤ min (a * wp x Q s + b * wp x R s)
    (a * wp y Q s + b * wp y R s) ⊓ c
    by (auto intro: tminus-left-mono)
  also
  have ... = min (a * wp x Q s + b * wp x R s ⊓ c)
    (a * wp y Q s + b * wp y R s ⊓ c)
    by (rule min-tminus-distrib)
  also { from slx sQ sR nna nnb nnc
    have wp x (λs. a * Q s + b * R s ⊓ c) s ≤
      wp x (λs. a * Q s + b * R s ⊓ c) s
      by (blast)
    moreover
    from sly sQ sR nna nnb nnc
    have wp y (λs. a * Q s + b * R s ⊓ c) s ≤
      wp y (λs. a * Q s + b * R s ⊓ c) s
      by (blast)
    ultimately
    have min (wp x (λs. a * Q s + b * R s ⊓ c) s)
      (wp y (λs. a * Q s + b * R s ⊓ c) s) ≤
      min (a * wp x Q s + b * wp x R s ⊓ c)
        (a * wp y Q s + b * wp y R s ⊓ c)
      by (auto)
  }
  finally show a * min (wp x Q s) (wp y Q s) +
    b * min (wp x R s) (wp y R s) ⊓ c ≤
    min (wp x (λs. a * Q s + b * R s ⊓ c) s)
      (wp y (λs. a * Q s + b * R s ⊓ c) s)
    .
qed

As for continuity, we insist on a finite support.

lemma sublinear-wp-SetPC:
fixes p :: 'a ⇒ 's prog
```
assumes slp: $\forall s. a, a \in \text{supp} (P s) \Rightarrow \text{sublinear} (\text{wp} (p a))$
and sum: $\forall s. (\sum a \in \text{supp} (P s). P s a) \leq 1$
and nnP: $\forall s. 0 \leq P s a$
and fin: $\forall s. \text{finite} (\text{supp} (P s))$
shows sublinear (wp (SetPC p P))

proof (rule sublinearI, simp add: wp-eval)

fix R::'s \Rightarrow real and Q::'s \Rightarrow real and s::'

and :real and b::real and c::real

assume slR: sound R and slQ: sound Q

and nna: $0 \leq a$ and nmb: $0 \leq b$ and nnc: $0 \leq c$

have $a * (\sum a' \in \text{supp} (P s). P s a' * \text{wp} (p a')) Q s + b * \text{wp} (p a') R s \leq c$

also have $\leq (\sum a' \in \text{supp} (P s). P s a' * (a * \text{wp} (p a') Q s + b * \text{wp} (p a') R s) \oplus c$

proof (rule tminus-right-antimono)

have $(\sum a' \in \text{supp} (P s). P s a' * c) \leq (\sum a' \in \text{supp} (P s). P s a') * c$

by (simp add: setsum-right-distrib setsum_left_distrib)

also from sum and nnP have $\leq 1 * c$

by (rule mult_right_mono)

finally show $(\sum a' \in \text{supp} (P s). P s a' * c) \leq c$ by (simp)

qed

also from fin

have $\leq (\sum a' \in \text{supp} (P s). P s a' * (a * \text{wp} (p a') Q s + b * \text{wp} (p a') R s) \oplus c$

by (blast intro: tminus-setsum_mono)

also have $= (\sum a' \in \text{supp} (P s). P s a' * (a * \text{wp} (p a') Q s + b * \text{wp} (p a') R s \oplus c))$

by (simp add: nnP tminus_left_distrib)

also { from slp sQ sR nna nmb nnc

have $\forall a', a' \in \text{supp} (P s) \Rightarrow a * \text{wp} (p a') Q s + b * \text{wp} (p a') R s \oplus c \leq wp (p a') (\lambda s. a * Q s + b * R s \oplus c) s$

by (blast)

with nnP

have $(\sum a' \in \text{supp} (P s). P s a' * (a * \text{wp} (p a') Q s + b * \text{wp} (p a') R s \oplus c))$

$\leq (\sum a' \in \text{supp} (P s). P s a' * wp (p a') (\lambda s. a * Q s + b * R s \oplus c) s)$

by (blast intro: setsum_mono mult_left_mono)

}

finally

show $a * (\sum a' \in \text{supp} (P s). P s a' * wp (p a') Q s) +$

$b * (\sum a' \in \text{supp} (P s). P s a' * wp (p a') R s) \oplus c \leq$

$(\sum a' \in \text{supp} (P s). P s a' * wp (p a') (\lambda s. a * Q s + b * R s \oplus c) s)$.

qed

lemma sublinear-wp-SetDC:
4.5. **SUBLINEARITY**

```plaintext
fixes \( p::'a \Rightarrow 's \text{ prog} \)
assumes slp: \( \bigwedge s. a. a \in S \Rightarrow \text{ sublinear } (wp (p a)) \)
    and hp: \( \bigwedge s. a. a \in S \Rightarrow \text{ healthy } (wp (p a)) \)
    and ne: \( \bigwedge s. S \neq \{\} \)
shows \( \text{ sublinear } (wp (\text{ SetDC } p S)) \)

proof (rule sublinearI, simp add: wp-eval del: Inf-image-eq, rule cInf-greatest)
fix \( P::'s \Rightarrow \text{ real and } Q::'s \Rightarrow \text{ real and } s::'s \text{ and } x y \)
    and a::real and b::real and c::real
assume \( sP:: \text{ sound } P \text{ and } sQ:: \text{ sound } Q \)
    and nna: \( 0 \leq a \) and \( \text{nmb: } 0 \leq b \) and \( \text{nnc: } 0 \leq c \)
from \( \text{ ne show } (\lambda \text{pr. } wp (p pr) (\lambda s. a * P s + b * Q s \odot c) s) \triangleq S s \neq \{\} \) by (auto)
assume \( \text{ yin: } y \in (\lambda \text{pr. } wp (p pr) (\lambda s. a * P s + b * Q s \odot c) s) \triangleq S s \)
then obtain \( x \) where \( \text{ xin: } x \in S \) and \( \text{ rwy: } y = wp (p x) (\lambda s. a * P s + b * Q s \odot c) s \)
    by (auto)
from \( \text{ xin hp sP nna } \)
have \( a * \text{ Inf } ((\lambda a. wp (p a) P s) \triangleq S s) \leq a * wp (p x) P s \)
    by (intro mult-left-mono) [OF cInf-lower] bdd-belowI [where \( m=0 \)], blast+ 
moreover from \( \text{ xin hp sQ nmb } \)
have \( b * \text{ Inf } ((\lambda a. wp (p a) Q s) \triangleq S s) \leq b * wp (p x) Q s \)
    by (intro mult-left-mono) [OF cInf-lower] bdd-belowI [where \( m=0 \)], blast+ 
ultimately
have \( a * \text{ Inf } ((\lambda a. wp (p a) P s) \triangleq S s) + b * \text{ Inf } ((\lambda a. wp (p a) Q s) \triangleq S s) \odot c \leq \)
    \( a * wp (p x) P s + b * wp (p x) Q s \odot c \)
    by (blast intro: minus-left-mono add-mono)
also from \( \text{ xin slp sP sQ nna nmb nnc } \)
have \( \ldots \leq wp (p x) (\lambda s. a * P s + b * Q s \odot c) s \)
    by (blast)
finally show \( a * \text{ Inf } ((\lambda a. wp (p a) P s) \triangleq S s) + b * \text{ Inf } ((\lambda a. wp (p a) Q s) \triangleq S s) \odot c \leq y \)
    by (simp add: rwy)
qued

**Lemma** sublinear-wp-Embed:

\( \text{ sublinear } t \Rightarrow \text{ sublinear } (wp (\text{ Embed } t)) \)
by (simp add: wp-eval)

**Lemma** sublinear-wp-repeat:

\[ \text{ sublinear } (wp p) \Rightarrow \text{ sublinear } (wp (\text{ repeat } n p)) \]
by (induct \( n \), simp-all add; sublinear-wp-Seq sublinear-wp-Skip healthy-wp-repeat)

**Lemma** sublinear-wp-Bind:

\[ \bigwedge s. \text{ sublinear } (wp (a (f s))) \Rightarrow \text{ sublinear } (wp (\text{ Bind } f a)) \]
by (rule sublinearI, simp add: wp-eval, auto)

4.5.2 Sublinearity for Loops

We break the proof of sublinearity loops into separate proofs of sub-distributivity and sub-additivity. The first follows by transfinite induction.

**Lemma** sub-distrib-wp-loop:

**Fixes** body; 's prog

**Assumes** sdb: sub-distrib (wp body)

**and** hb: healthy (wp body)

**and** nhb: nearly-healthy (wp body)

**Shows** sub-distrib (wp (do G → body od))

**Proof** –

**Have** ∀ P s. sound P → wp (do G → body od) P s ⊕ 1 ≤ wp (do G → body od) (λs. P s ⊕ 1) s

**Proof** (rule loop-induct[OF hb nhb], safe)

**Fix** S: (s trans × 's trans) set and P: 's expect and s: 's

**Assume** saS: ∀ x ∈ S. ∀ P s. sound P → fst x P s ⊕ 1 ≤ fst x (λs. P s ⊕ 1) s

**and** sP: sound P

**and** fS: ∀ x ∈ S. feasible (fst x)

**From** sP have sPm: sound (λs. P s ⊕ 1) by (auto intro: tminus-sound)

**Have** nnSup: ∀ s. 0 ≤ Sup-trans (fst ' S) (λs. P s ⊕ 1) s

**Proof** (cases S = {}, simp add: Sup-trans-def Sup-exp-def)

**Fix** s

**Assume** S ≠ {}

**Then obtain** x where xin: x ∈ S by (auto)

**With** fS sPm have 0 ≤ fst x (λs. P s ⊕ 1) s by (auto)

**Also from** xin fS sPm have ... ≤ Sup-trans (fst ' S) (λs. P s ⊕ 1) s

**by** (auto intro!: le-funD[OF Sup-trans-upper2])

**Finally show** ?thesis s.

**QED**

**Have** ∀ x s. fst x P s ≤ (fst x P s ⊕ 1) + 1 by (simp add: tminus-def)

**Also from** saS sP

**Have** ∀ x s. x ∈ S → (fst x P s ⊕ 1) + 1 ≤ fst x (λs. P s ⊕ 1) s + 1

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

**Also {**

**From** sP have sound (λs. P s ⊕ 1) by (auto intro: tminus-sound)

**With** fS have ∀ x s. x ∈ S →fst x (λs. P s ⊕ 1) s + 1 ≤ Sup-trans (fst ' S) (λs. P s ⊕ 1) s + 1

**by** (blast intro!: add-right-mono le-funD[OF Sup-trans-upper2])

**}**

**Finally have** le: ∀ s. ∀ x ∈ S. fst x P s ≤ Sup-trans (fst ' S) (λs. P s ⊕ 1) s + 1

**by** (auto)

**Moreover from** nnSup have nn: ∀ s. 0 ≤ Sup-trans (fst ' S) (λs. P s ⊕ 1) s + 1

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

have leSup: Sup-trans (fst ' S) P s ≤ Sup-trans (fst ' S) (λs. P s ⊓ 1) s + 1

unfolding Sup-trans-def
by(intro le-funD[OF Sup-exp-least], auto)

show Sup-trans (fst ' S) P s ⊓ 1 ≤ Sup-trans (fst ' S) (λs. P s ⊓ 1) s
proof(cases Sup-trans (fst ' S) P s ≤ 1, simp-all add:nnSup)
  from leSup have Sup-trans (fst ' S) P s - 1 ≤
    Sup-trans (fst ' S) (λs. P s ⊓ 1) s + 1 - 1
    by(auto)
  thus Sup-trans (fst ' S) P s - 1 ≤ Sup-trans (fst ' S) (λs. P s ⊓ 1) s
by(simp)
qed
next

fix t::'s trans and P::'s expect and s::'s
assume IH: ∀ P s. sound P → t P s ⊓ 1 ≤ t (λa. P a ⊓ 1) s
  and ft: feasible t
  and sP: sound P

from sP have sound (λs. P s ⊓ 1) by(auto intro:tminus-sound)
with ft have s2: sound (t (λs. P s ⊓ 1)) by(auto)
from sP ft have sound (t P) by(auto)

hence s3: sound (λs. t P s ⊓ 1) by(auto intro:tminus-sound)

show wp (body :: Embed t « G » Skip) P s ⊓ 1 ≤
  wp (body :: Embed t « G » Skip) (λa. P a ⊓ 1) s
proof(simp add:wp-eval)
  have «G» s * wp body (t P) s + (1 - «G» s) * P s ⊓ 1 =
    «G» s * wp body (t P) s + (1 - «G» s) * P s ⊓ («G» s + (1 - «G» s))
  by(simp)
  also have ... ≤ («G» s * wp body (t P) s ⊓ «G» s) +
    ((1 - «G» s) * P s ⊓ (1 - «G» s))
  by(rule tminus-add-mono)
  also have ... = «G» s * (wp body (t P) s ⊓ 1) + (1 - «G» s) * (P s ⊓ 1)
  by(simp add:tminus-left-distrib)
  also {from ft sP have wp body (t P) s ⊓ 1 ≤ wp body (λs. t P s ⊓ 1) s
    by(auto intro:sub-distribD[OF sdb])
  also {from IH sP have λs. t P s ⊓ 1 t (λs. P s ⊓ 1) by(auto)
    with sP ft s2 s3 have wp body (λs. t P s ⊓ 1) s ≤ wp body (t (λs. P s ⊓ 1))
    by(blast intro:le-funD[OF mono-transD, OF healthy-monoD, OF hb])
  }
finally have «G» s * (wp body (t P) s ⊓ 1) + (1 - «G» s) * (P s ⊓ 1) ≤
  «G» s * wp body (t (λs. P s ⊓ 1)) s + (1 - «G» s) * (P s ⊓ 1)
  by(auto intro:add-right-mono mult-left-mono)
}
finally show «G» s * wp body (t P) s + (1 - «G» s) * P s ⊓ 1 ≤
\texttt{«G» s \ast wp body (t (\lambda s. P s \ominus 1)) s + (1 - «G» s) \ast (P s \ominus 1)}.

\texttt{qed}

\texttt{next}

\texttt{fix t t';s trans and P::'s expect and s::'s}

\texttt{assume IH: \forall P s. sound P \rightarrow t P s \ominus 1 \leq t (\lambda a. P a \ominus 1) s}

\texttt{and eq: equiv-trans t t' and sP: sound P}

\texttt{from sP have t' P s \ominus 1 = t P s \ominus 1 \texttt{by(simp add:equiv-transD[OF eq])}}

\texttt{also from sP IH have ... \leq t (\lambda s. P s \ominus 1) s \texttt{by(auto)}}

\texttt{also \{}

\texttt{from sP have sound (\lambda s. P s \ominus 1) \texttt{by(simp add:tminus-sound)}}

\texttt{hence t (\lambda s. P s \ominus 1) s = t' (\lambda s. P s \ominus 1) s \texttt{by(simp add:equiv-transD[OF eq])}}

\texttt{\}}

\texttt{finally show t' P s \ominus 1 \leq t' (\lambda s. P s \ominus 1) s}.

\texttt{qed}

\texttt{thus \texttt{thesis by(auto intro!:sub-distribl)}}

\texttt{qed}

For sub-additivity, we again use the limit-of-iterates characterisation. Firstly, all iterates are sublinear:

\texttt{lemma sublinear-iterates:}

\texttt{assumes hb: healthy (wp body)}

\texttt{and sb: sublinear (wp body)}

\texttt{shows sublinear (iterates body G i)}

\texttt{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.

\texttt{lemma sub-add-wp-loop:}

\texttt{fixes body::'s prog}

\texttt{assumes sb: sublinear (wp body)}

\texttt{and cb: bl-cts (wp body)}

\texttt{and hwp: healthy (wp body)}

\texttt{shows sub-add (wp (do G \rightarrow body od))}

\texttt{proof}

\texttt{fix P Q::'s expect and s::'s}

\texttt{assume sP: sound P and sQ: sound Q}

\texttt{from hwp cb sP have (\lambda i. iterates body G i P s) \longrightarrow wp do G \rightarrow body od P s}

\texttt{by(rule loop-iterates)}

\texttt{moreover}

\texttt{from hwp cb sQ have (\lambda i. iterates body G i Q s) \longrightarrow wp do G \rightarrow body od Q s}

\texttt{by(rule loop-iterates)}

\texttt{ultimately}

\texttt{have (\lambda i. iterates body G i P s + iterates body G i Q s) \longrightarrow}
4.5. SUBLINEARITY

\[
wp \text{ do } G \rightarrow \text{ body od } P \ s + \ wp \text{ do } G \rightarrow \text{ body od } Q \ s
\]

by (rule tendsto-add)

moreover 
\[
\text{from sublinear-subadd}(\text{OF sublinear-iterates, OF hwp sb,}
\]
\[
\text{OF healthy-feasibleD}(\text{OF iterates-healthy, OF hwp})] \ sP \ sQ
\]

have \[
\bigwedge_i \text{ iterates body } G \ i \ P \ s + \ \text{ iterates body } G \ i \ Q \ s \leq \ \text{ iterates body } G \ i \ (\lambda s. \ P \ s + Q \ s) \]

by (rule sub-addD)

\}

moreover 
\[
\text{from } sP \ sQ \ \text{ have } \text{ sound } (\lambda s. \ P \ s + Q \ s) \ \text{ by (blast intro: sound-intros)}
\]

with hwp cb have \[
(\lambda i. \ \text{ iterates body } G \ i \ (\lambda s. \ P \ s + Q \ s)) \rightarrow
\]

wp do G \rightarrow \text{ body od } (\lambda s. \ P \ s + Q \ s) \ s

by (blast intro: loop-iterates)

}\}

ultimately

show \[
wp \text{ do } G \rightarrow \text{ body od } P \ s + \ \text{ wp do } G \rightarrow \text{ body od } Q \ s \leq \ \text{ wp do } G \rightarrow \text{ body od } (\lambda s. \ P \ s + Q \ s) \ s
\]

by (blast intro: LIMSEQ-le)

qed

lemma sublinear-wp-loop:

fixes \text{ body::'s prog}

assumes \text{ hb: healthy (wp body)}

and \text{ nhb: nearly-healthy (wp body)}

and \text{ sb: sublinear (wp body)}

and \text{ cb: bd-cts (wp body)}

shows \text{ sublinear (wp (do G \rightarrow \text{ body od}))}

using sublinear-sub-distrib[\text{OF sb}] sublinear-subadd[\text{OF sb}]

\text{ healthy-feasibleD}(\text{OF hb})

by (iprover intro: sd-sa-sublinear [OF -- healthy-wp-loop [OF hb]]

sub-distrib-wp-loop sub-add-wp-loop assms)

lemmas sublinear-intros =

sublinear-wp-Abort
sublinear-wp-Skip
sublinear-wp-Apply
sublinear-wp-Seq
sublinear-wp-PC
sublinear-wp-DC
sublinear-wp-SetPC
sublinear-wp-SetDC
sublinear-wp-Embed
sublinear-wp-repeat
sublinear-wp-Bind
sublinear-wp-loop

end
4.6 Determinism

theory Determinism imports WellDefined begin

We provide a set of lemmas for establishing that appropriately restricted
programs are fully additive, and maximal in the refinement order. This is
particularly useful with data refinement, as it implies correspondence.

4.6.1 Additivity

lemma additive-wp-Abort:
  additive (wp (Abort))
  by(auto simp:wp-eval)

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)
      by(blast dest:additiveD)
    with adda sP sQ hb
    show wp a (wp b (λs. P s + Q s)) s =
      wp a (wp b P) s + (wp a (wp b Q)) s
      by(auto intro:fun-cong[OF additiveD])
  qed

lemma additive-wp-PC:
  [ additive (wp a); additive (wp b) ] ⇒ additive (wp (a ⊕ b))
  by(rule additiveI, simp add:additiveD field-simps wp-eval)

DC is not additive.
4.6. DETERMINISM

```
lemma additive-wp-SetPC:
  [ \[ \land x s A x \in \text{supp} (p s) \implies \text{additive} (wp (a x)); \land s A \text{finite} \text{supp} (p s) \] ] \implies
  additive (wp (SetPC a p))
by (rule additiveI,
    simp add:wp-eval additiveD distrib-left setsum.distrib)

lemma additive-wp-Bind:
  [ \[ \land x A \text{additive} (wp (a (f x))) \] ] \implies
  additive (wp (Bind f a))
by (simp add:wp-eval additive-def)

lemma additive-wp-Embed:
  [ additive t ] \implies
  additive (wp (Embed t))
by (simp add:wp-eval)

lemma additive-wp-repeat:
  additive (wp a) \implies
  well-def a \implies
  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) ] \implies
  maximal (wp (a ;; b))
by (simp add:wp-eval maximal-def)

lemma max-wp-PC:
  [ maximal (wp a); maximal (wp b) ] \implies
  maximal (wp (a P 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 \ DC b))
by (rule maximalI, simp add:wp-eval maximalD)

lemma max-wp-SetPC:
  [ \land s A a \in \text{supp} (P s) \implies \text{maximal} (wp (p a)); \land s A (\sum a \in \text{supp} (P s). P s a) = ]
```
\[ \begin{array}{l}
\text{lemma max-wp-SetDC:} \\
\quad \text{fixes } p: \mathbb{P} \rightarrow \mathbb{S} \text{ prog} \\
\quad \text{assumes } mp: \forall s. a \in S \; s \Rightarrow \text{maximal } (wp (p a)) \\
\quad \quad \text{and } ne: \forall s. S \; s \neq \{\} \\
\quad \text{shows } \text{maximal } (wp (\text{SetDC } p \; S)) \\
\quad \text{proof (rule maximalI, rule ext, unfold wp-eval)} \\
\quad \quad \text{fix } c: \mathbb{R} \text{ and } s: \mathbb{S} \\
\quad \quad \text{assume } 0 \leq c \\
\quad \quad \text{hence } \text{Inf } ((\lambda a. \; wp (p a) \; (\lambda -. c) \; s) \; s) = \text{Inf } ((\lambda -. c) \; s) \\
\quad \quad \quad \text{using } mp \text{ by (simp add: maximalD cong:image-cong)} \\
\quad \quad \quad \text{also } \{ \\
\quad \quad \quad \quad \text{from } ne \text{ have } \exists a. a \in S \; s \text{ by (blast)} \\
\quad \quad \quad \quad \text{hence } \text{Inf } ((\lambda -. c) \; s) = c \\
\quad \quad \quad \quad \text{by (simp add: image-def)} \\
\quad \quad \} \\
\quad \text{finally show } \text{Inf } ((\lambda a. \; wp (p a) \; (\lambda -. c) \; s) \; s) = c. \\
\text{qed}
\end{array} \]

\[ \begin{array}{l}
\text{lemma max-wp-Embed:} \\
\quad \text{maximal } t \Rightarrow \text{maximal } (wp (\text{Embed } t)) \\
\quad \text{by (simp add: wp-eval)} \\
\text{lemma max-wp-repeat:} \\
\quad \text{maximal } (wp a) \Rightarrow \text{maximal } (wp (\text{repeat } n \; a)) \\
\quad \text{by (induct } n, \text{ simp-all add: max-wp-Skip max-wp-Seq)} \\
\text{lemma max-wp-Bind:} \\
\quad \text{assumes } ma: \forall s. \text{maximal } (wp (a (f s))) \\
\quad \text{shows } \text{maximal } (wp (\text{Bind } f \; a)) \\
\quad \text{proof (rule maximalI, rule ext, simp add: wp-eval)} \\
\quad \quad \text{fix } c: \mathbb{R} \text{ and } s \\
\quad \quad \text{assume } 0 \leq c \\
\quad \quad \text{with } ma \text{ have } wp (a (f s)) \; (\lambda -. c) = (\lambda -. c) \text{ by (blast)} \\
\quad \quad \text{thus } wp (a (f s)) \; (\lambda -. c) \; s = c \text{ by (auto)} \\
\text{qed}
\end{array} \]

\[ \begin{array}{l}
\text{lemmas max-intros =} \\
\quad \text{max-wp-Skip max-wp-Apply} \\
\quad \text{max-wp-Seq max-wp-PC} \\
\quad \text{max-wp-DC max-wp-SetPC} \\
\quad \text{max-wp-SetDC max-wp-Embed} \\
\quad \text{max-wp-Bind max-wp-repeat}
\end{array} \]

A healthy transformer that terminates is maximal.

\[ \begin{array}{l}
\text{lemma healthy-term-max:}
\end{array} \]
4.6. DETERMINISM

assumes \(ht\): healthy \(t\)
and \(\text{trm}: \lambda s. 1 \vdash t (\lambda s. 1)\)
shows maximal \(t\)

proof (intro maximalI ext)
fix \(c::\text{real} \text{ and } s\)
assume \(nnc: \theta \leq c\)

have \(t (\lambda s. c) s = t (\lambda s. 1 * c) s\) by (simp)
also from \(nnc\) healthy-scalingD [OF \(ht\)]
have \(\ldots = c * t (\lambda s. 1) s\) by (simp add: scalingD)
also \{ 
  from \(ht\) have \(t (\lambda s. 1) \vdash \lambda s. 1\) by (auto)
  with \(\text{trm}\) 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 \(a\)) \Rightarrow determ (wp \(b\)) \Rightarrow well-def \(b\) \Rightarrow determ (wp (\(a ; ; b\))
by (intro determI fa-intros max-intros, auto)

lemma det-wp-PC:
determ (wp \(a\)) \Rightarrow determ (wp \(b\)) \Rightarrow determ (wp (\(a \oplus b\))
by (intro determI fa-intros max-intros, auto)

lemma det-wp-SetPC:
(\(\forall x. s. x \in \text{supp (p s)} \Rightarrow \text{determ (wp (a x))}\)) \Rightarrow
(\(\forall s. \text{finite (supp (p s))}\) \Rightarrow
(\(\forall s. \text{setsum (p s) (supp (p s)) = 1}\) \Rightarrow
determ (wp (SetPC a p))
by (intro determI fa-intros max-intros, auto)

lemma det-wp-Bind:
(\(\forall x. \text{determ (wp (a (f x))}\)) \Rightarrow \text{determ (wp (Bind f a))}
by (intro determI fa-intros max-intros, auto)

lemma det-wp-Embed:
determ \(t\) \Rightarrow determ (wp (Embed \(t\)))
CHAPTER 4. THE PGCL LANGUAGE

by (simp add: wp-eval)

lemma det-wp-repeat:
determ (wp a) \implies\ well-def a \implies determ (wp (repeat n a))
by (intro determI fa-intros max-intros, auto)

lemmas determ-intros =
det-wp-Skip det-wp-Apply
det-wp-Seq det-wp-PC
det-wp-SetPC det-wp-Bind
det-wp-Embed det-wp-repeat

end

4.7 Well-Defined Programs.

theory WellDefined imports
  Healthiness
  Sublinearity
  LoopInduction
begin

The definition of a well-defined program collects the various notions of healthiness and well-behavedness that we have so far established: healthiness of the strict and liberal transformers, continuity and sublinearity of the strict transformers, and two new properties. These are that the strict transformer always lies below the liberal one (i.e. that it is at least as strict, recalling the standard embedding of a predicate), and that expectation conjunction is distributed between them in a particular manner, which will be crucial in establishing the loop rules.

4.7.1 Strict Implies Liberal

This establishes the first connection between the strict and liberal interpretations (wp and wlp).

definition wp-under-wlp :: 's prog \Rightarrow bool
where
wp-under-wlp prog \equiv \forall P. \ unitary P \implies wp prog P \vdash wlp prog P

lemma wp-under-wlpI[intro]:
[ \forall P. \ unitary P \implies wp prog P \vdash wlp prog P ] \implies wp-under-wlp prog
unfolding wp-under-wlp-def by (simp)

lemma wp-under-wlpD[dest]:
[ wp-under-wlp prog; \ unitary P ] \implies wp prog P \vdash wlp prog P
unfolding wp-under-wlp-def by (simp)
4.7. WELL-DEFINED PROGRAMS.

lemma wp-under-le-trans:
wp-under-wlp a \implies le-utrans (wp a) (wlp a)
by(blast)

lemma wp-under-wlp-Abort:
wp-under-wlp Abort
by(rule wp-under-wlpI, unfold wp-eval, auto)

lemma wp-under-wlp-Skip:
wp-under-wlp Skip
by(rule wp-under-wlpI, unfold wp-eval, blast)

lemma wp-under-wlp-Apply:
wp-under-wlp (Apply f)
by(auto simp: wp-eval)

lemma wp-under-wlp-Seq:
assumes h-wlp-a: nearly-healthy (wlp a)
and h-wp-b: healthy (wp b)
and h-wlp-b: nearly-healthy (wlp b)
and wp-a-a: wp-under-wlp a
and wp-a-b: wp-under-wlp b
shows wp-under-wlp (a ;; b)
proof(rule wp-under-wlpI, unfold wp-eval o-def)
fix P:: a \Rightarrow real assume uP: unitary P
with h-wp-b have anitary (wp b P) by(blast)
with wp-u-a have wp a (wp b P) \vdash wlp a (wp b P) by(auto)
also {
  from wp-u-b and uP have wp b P \vdash wlp b P by(blast)
  with h-wlp-a and h-wlp-b and h-wp-b and uP
  have wp a (wp b P) \vdash wlp a (wp b P)
  by(blast intro:nearly-healthy-monoD[OF h-wlp-a])
}
finally show wp a (wp b P) \vdash wlp a (wp b P).
qed

lemma wp-under-wlp-PC:
assumes h-wp-a: healthy (wp a)
and h-wlp-a: nearly-healthy (wlp a)
and h-wp-b: healthy (wp b)
and h-wlp-b: nearly-healthy (wlp b)
and wp-a-a: wp-under-wlp a
and wp-a-b: wp-under-wlp b
and uP: unitary P
shows wp-under-wlp (a p\oplus b)
proof(rule wp-under-wlpI, unfold wp-eval, rule le-funI)
fix Q:: a \Rightarrow real and s
assume uQ: unitary Q
from uP have $P s \leq 1$ by\(\text{blast}\)

hence $0 \leq 1 - P s$ by\(\text{simp}\)

moreover

from uQ and wp-u-b have $wp \ b \ Q s \leq wlp \ b \ Q s$ by\(\text{blast}\)

ultimately

have $(1 - P s) \ast wp \ b \ Q s \leq (1 - P s) \ast wlp \ b \ Q s$

by\(\text{blast intro: mult-left-mono}\)

moreover {

from uQ and wp-u-a have $wp \ a \ Q s \leq wlp \ a \ Q s$ by\(\text{blast}\)

with uP have $P s \ast wp \ a \ Q s \leq P s \ast wlp \ a \ Q s$

by\(\text{blast intro: mult-left-mono}\)

}

ultimately

show $P s \ast wp \ a \ Q s + (1 - P s) \ast wp \ b \ Q s \leq P s \ast wlp \ a \ Q s + (1 - P s) \ast wlp \ b \ Q s$

by\(\text{blast intro: add-mono}\)

qed

lemma wp-under-wlp-DC:

assumes wp-u-a: wp-under-wlp a

and wp-u-b: wp-under-wlp b

shows wp-under-wlp \((a \prod b)\)

proof\(\text{(rule wp-under-wlpI, unfold wp-eval, rule le-funI)}\)

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

assume uQ: unitary Q

from wp-u-a uQ have $wp \ a \ Q s \leq wlp \ a \ Q s$ by\(\text{blast}\)

moreover

from wp-u-b uQ have $wp \ b \ Q s \leq wlp \ b \ Q s$ by\(\text{blast}\)

ultimately

show $\min(wp \ a \ Q s) \ast wp \ b \ Q s \leq \min(wlp \ a \ Q s) \ast wlp \ b \ Q s$

by\(\text{auto}\)

qed

lemma wp-under-wlp-SetPC:

assumes wp-u-f: $\forall s \ a. \ a \in \supp(P s) \Rightarrow \wp-under-wlp(f a)$

and nP: $\forall s \ a. \ a \in \supp(P s) \Rightarrow 0 \leq P s a$

shows wp-under-wlp \((\SetPC f P)\)

proof\(\text{(rule wp-under-wlpI, unfold wp-eval, rule le-funI)}\)

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

assume uQ: unitary Q

from wp-u-f uQ nP

show $(\sum a \in \supp(P s). \ P s a \ast wp(f a) \ Q s) \leq (\sum a \in \supp(P s). \ P s a \ast wlp(f a) \ Q s)$

by\(\text{auto intro!: setsum-mono mult-left-mono}\)

qed
4.7. WELL-DEFINED PROGRAMS.

lemma wp-under-wlp-SetDC:
assumes wp-u-f: \( \forall a. a \in S \implies wp-under-wlp (f a) \)
and hf: \( \forall a. a \in S \implies healthy (wp (f a)) \)
and nS: \( \forall s. S s \neq \{ \} \)
shows wp-under-wlp (SetDC f S)

proof (rule wp-under-wlpI, rule le-funI, unfold wp-eval)

fix Q: a \Rightarrow real and s
assume uQ: unitary Q

show Inf ((\(\lambda a. wp (f a) Q s\) \ ' S s) \ ' S s) \leq Inf ((\(\lambda a. wlp (f a) Q s\) \ ' S s)

proof (rule cInf-mono)

from nS show (\(\lambda a. wlp (f a) Q s\) \ ' S s) \neq \{ \} by (blast)

fix x assume xin: x \in (\(\lambda a. wlp (f a) Q s\) \ ' S s)
then obtain a where ain: a \in S s and xrw: x = wlp (f a) Q s
by (blast)

with wp-u-f uQ
have wp (f a) Q s \leq wlp (f a) Q s by (blast)
moreover from ain have wp (f a) Q s  \in (\(\lambda a. wp (f a) Q s\) \ ' S s)
by (blast)
ultimately show \exists y \in (\(\lambda a. wp (f a) Q s\) \ ' S s. y \leq x
by (auto simp: xrw)

next

fix y assume yin: y \in (\(\lambda a. wp (f a) Q s\) \ ' S s)
then obtain a where ain: a \in S s and yrw: y = wp (f a) Q s
by (blast)

with hf aQ have unitary (wp (f a) Q) by (auto)
with yrw show 0 \leq y by (auto)
qed

qed

lemma wp-under-wlp-Embed:
wp-under-wlp (Embed t)
by (rule wp-under-wlpI, unfold wp-eval, blast)

lemma wp-under-wlp-loop:
fixes body::'s prog
assumes hwp: healthy (wp body)
and hwlp: nearly-healthy (wlp body)
and wp-under: wp-under-wlp body
shows wp-under-wlp (do G → body od)

proof (rule wp-under-wlpI)
fix P::'s expect
assume uP: unitary P hence sP: sound P by (auto)

let ?X Q s = «G» s * wp body Q s + «\(N\) G» s * P s
let ?Y Q s = «G» s * wlp body Q s + «\(N\) G» s * P s
show \( wp \ (do \ G \rightarrow \ \text{body} \ od) \ P \vdash \ wlp \ (do \ G \rightarrow \ \text{body} \ od) \ P \)

\textbf{proof}\(\quad\)

\textbf{thm} lfp-loop-fp

\textbf{from} hwp sP \textbf{have} lfp-exp \( ?X = ?X \ (lfp-exp \ ?X) \)

\textbf{by}(rule lfp-wp-loop-unfold)

\textbf{hence} lfp-exp \( ?X \vdash ?X \ (lfp-exp \ ?X) \textbf{ by}(\text{simp}) \)

\textbf{also} \{

\textbf{from} hwp uP \textbf{have} wp body (lfp-exp \ ?X) \vdash wlp body (lfp-exp \ ?X)

\textbf{by}(auto intro:wp-under-wlpD[OF wp-under lfp-loop-unitary])

\textbf{hence} \( ?X \ (lfp-exp \ ?X) \vdash ?Y \ (lfp-exp \ ?X) \)

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

\}

\textbf{finally show} lfp-exp \( ?X \vdash ?Y \ (lfp-exp \ ?X) \).

\textbf{from} hwp uP \textbf{show} unitary (lfp-exp \ ?X)

\textbf{by}(auto intro:lfp-loop-unitary)

\textbf{qed}

\textbf{qed}

\textbf{lemma} wp-under-wlp-repeat:

\[ \text{healthy} (wp \ a); \text{nearly-healthy} (wlp \ a); \text{wp-under-wlp} \ a \] \implies wp-under-wlp (repeat \( n \) \ a)

\textbf{by}(induct \( n \), auto intro!:wp-under-wlp-Skip wp-under-wlp-Seq healthy-intros)

\textbf{lemma} wp-under-wlp-Bind:

\[ \forall s. \text{wp-under-wlp} \ (a \ (f \ s)) \] \implies wp-under-wlp (Bind \( f \ a \))

\textbf{unfolding} wp-under-wlp-def \textbf{by}(auto simp:wp-eval)

\textbf{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

\subsection*{4.7.2 Sub-Distributivity of Conjunction}

\textbf{definition}

\( \text{sub-distrib-pconj :: 's prog \Rightarrow bool} \)

\textbf{where}

\( \text{sub-distrib-pconj prog} \equiv \)

\( \forall P \ Q. \ \text{unitary} \ P \rightarrow \text{unitary} \ Q \rightarrow \)

\( \text{wlp prog} \ P \&\& \ wp \ prog \ Q \vdash \ wp \ prog \ (P \&\& Q) \)

\textbf{lemma} sub-distrib-pconj[intro]:

\[ \forall P \ Q. \ [ \ \text{unitary} \ P; \ \text{unitary} \ Q ] \rightarrow \text{wlp prog} \ P \&\& \ wp \ prog \ Q \vdash \ wp \ prog \ (P \&\& Q) \]

\textbf{sub-distrib-pconj prog}
4.7. WELL-DEFINED PROGRAMS.

unfolding sub-distrib-pconj-def by(simp)

lemma sub-distrib-pconjD[dest]:
\[ \forall P Q. \ [\ [ \text{sub-distrib-pconj prog; unitary } P; \text{ unitary } Q ] \] \implies \]
\[ \text{wlp prog } P \&\& \text{ wp prog } Q \vdash \text{ wp prog } (P \&\& Q) \]

unfolding sub-distrib-pconj-def by(simp)

lemma sdp-Abort:
sub-distrib-pconj Abort
by(rule sub-distrib-pconjI, unfold wp-eval, auto intro:exp-conj-rzero)

lemma sdp-Skip:
sub-distrib-pconj Skip
by(rule sub-distrib-pconjI, simp add:wp-eval)

lemma sdp-Seq:
fixes a and b
assumes sdp-a: sub-distrib-pconj a
and sdp-b: sub-distrib-pconj b
and h-wp-a: healthy (wp a)
and h-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 wlp a (wp b P) \&\& wp a (wp b Q) \vdash 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 wlp b P \&\& wp b Q \vdash 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) \vdash wp a (wp b (P \&\& Q))
by(blast intro!:mono-transD[OF healthy-monoD, OF h-wp-a] unitary-sound
unitary-intros sound-intros)
}
finally show wlp a (wp b P) \&\& wp a (wp b Q) \vdash wp a (wp b (P \&\& Q)) .
qed

lemma sdp-Apply:
sub-distrib-pconj (Apply f)
by(rule sub-distrib-pconjI, simp add:wp-eval)

lemma sdp-DC:
fixes a::'s prog and b
assumes sdp-a: sub-distrib-pconj a
and sdp-b: sub-distrib-pconj b
and h-wp-a: healthy (wp a)
and h-wp-b: healthy (wp b)
and h-wlp-b: nearly-healthy (wp b)
shows \text{sub-distrib-pconj} (a \sqcup 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 \text{exp-conj-def} by (rule min-conj)
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 \ldots \vdash 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 \ldots \vdash wp b (P & & Q)
    by (blast)
    finally have wlp b P s & wp b Q s ≤ wp b (P & & Q) s
    by (rule le-funD)
  }

  ultimately
  have min (wlp a P s & wp a Q s) (wlp b P s & wp b Q s) ≤
  min (wp a (P & & Q) s) (wp b (P & & Q) s)
  by (auto)
}

finally
show ((λs. min (wlp a P s) (wlp b P s)) &&
(λs. min (wp a Q s) (wp b Q s))) s ≤
min (wp a (P & & Q) s) (wp b (P & & Q) s)

qed

lemma sdp-PC:
fixes a::'s prog and b
assumes sdp-a: \text{sub-distrib-pconj} a
  and sdp-b: \text{sub-distrib-pconj} b
  and h-wp-a: healthy (wp a)
  and h-wp-b: healthy (wp b)
  and h-wlp-b: nearly-healthy (wp b)
  and uP: unitary P
shows \text{sub-distrib-pconj} (a \oplus 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
4.7. WELL-DEFINED PROGRAMS.

have \(nnA: 0 \leq P s \text{ and } nnB: 0 \leq 1 - P s\)

using \(uP\) by(auto simp:sign-simps)

note \(nn = nnA nnB\)

have \((\lambda s. P s \ast wlp a Q s + (1 - P s) \ast wlp b Q s) \&\&
(\lambda s. P s \ast wp a R s + (1 - P s) \ast wp b R s)) s =

\((P s \ast wlp a Q s + (1 - P s) \ast wlp b Q s) +
(P s \ast wp a R s + (1 - P s) \ast wp b R s)) \odot 1\)

by(simp add:exp-conj-def pconj-def)

also have ... = \(P s \ast (wlp a Q s + wp a R s) +
(1 - P s) \ast (wlp b Q s + wp b R s) \odot 1\)

by(simp add:field-simps)

also have ... \(= (P s \ast ((wlp a Q s + wp a R s) \&\&
(1 - P s) \ast ((wlp b Q s + wp b R s) \&\&
(P s + (1 - P s))\)

by(simp)

also have ... \(\leq (P s \ast (wlp a Q s + wp a R s) \odot P s) +
((1 - P s) \ast (wlp b Q s + wp b R s) \odot (1 - P s))\)

by(rule tminus-add-mono)

also have ... \(= (P s \ast ((wlp a Q s + wp a R s) \&\&
(1 - P s) \ast ((wlp b Q s + wp b R s) \&\&
Q s) +
(1 - P s) \ast ((wlp b Q s + wp b R s) \&\&
(R s) s)\)

by(simp add:exp-conj-def pconj-def)

also \{ from \(sdp-a\ sdp-b uQ aR\)

have \(P s \ast (wlp a Q \&\& wp a R) s \leq P s \ast wp a (Q \&\& R) s\)

and \((1 - P s) \ast (wlp b Q \&\& wp b R) s \leq (1 - P s) \ast wp b (Q \&\& R) s\)

by(auto intro:le-funD sub-distrib-pconjD nn mult-left-mono)

hence \(P s \ast ((wlp a Q \&\& wp a R) s) +\)

\((1 - P s) \ast ((wlp b Q \&\& wp b R) s) \leq\)

\(P s \ast wp a (Q \&\& R) s + (1 - P s) \ast wp b (Q \&\& R) s\)

by(auto) \}

finally show \((\lambda s. P s \ast wlp a Q s + (1 - P s) \ast wlp b Q s) \&\&
(\lambda s. P s \ast wp a R s + (1 - P s) \ast wp b R s)) s \leq\)

\(P s \ast wp a (Q \&\& R) s + (1 - P s) \ast wp b (Q \&\& R) s\).

qed

lemma \(sdp-Embed:\)

\([\forall P Q. \text{ unitary } P; \text{ unitary } Q \implies t P \&\& t Q \vdash t (P \&\& Q)] \implies\)

sub-distrib-pconj (Embed t)

by(auto simp:wp-eval)

lemma \(sdp-repeat:\)

fixes \(a::'s prog\)

assumes \(sdpa: \text{ sub-distrib-pconj } a\)
and hwp: healthy (wp a) and hwlp: nearly-healthy (wlp a)
shows sub-distrib-pconj (repeat n a) (is ?X n)
proof (induct n)
show ?X 0 by (simp add: sdp-Skip)
fix n assume IH: ?X n
show ?X (Suc n)
proof (rule sub-distrib-pconjI, simp add: wp-eval)
five P::'s ⇒ real and Q::'s ⇒ real
assume uP: unitary P and uQ: unitary Q
from asms have hwlp: nearly-healthy (wlp (repeat n a))
    and hwp: healthy (wp (repeat n a))
by (auto intro: healthy-intros)
from uP and hwlp have unitary (wlp (repeat n a) P) by (blast)
moreover from uQ and hwp have unitary (wp (repeat n a) Q) by (blast)
ultimately have wp a (wp (repeat n a) P) &&
    wp a (wp (repeat n a) Q) ⊢
using sdp by (blast)
also {
from hwlp have nearly-healthy (wlp (repeat n a)) by (rule healthy-intros)
with uP have sound (wlp (repeat n a) P) by (auto)
moreover from hwlp uQ have sound (wp (repeat n a) Q)
by (auto intro: healthy-intros)
ultimately have sound (wlp (repeat n a) P && wp (repeat n a) Q)
by (rule exp-conj-sound)
moreover {
from uP uQ have sound (P && Q) by (auto intro: exp-conj-sound)
with hwp have sound (wp (repeat n a) (P && Q))
by (auto intro: healthy-intros)
}
moreover from uP uQ IH
have wp (repeat n a) P && wp (repeat n a) Q ⊢ wp (repeat n a) (P && Q)
by (blast)
ultimately have wp a (wp (repeat n a) P && wp (repeat n a) Q) ⊢
    wp a (wp (repeat n a) (P && Q))
by (rule mono-transD[OF healthy-monoD, OF hwlp])
}
finally show wp a (wp (repeat n a) P) && wp a (wp (repeat n a) Q) ⊢
    wp a (wp (repeat n a) (P && Q)).
qed
qed

lemma sdp-SetPC:
fixes p::'a ⇒ 's prog
assumes sdp: (\s. a ∈ supp (P s) → sub-distrib-pconj (p a)
and fin: (\s. finite (supp (P s)))
and nnp: (\s. 0 ≤ P s a
and sub: (\s. setsum (P s) (supp (P s)) ≤ 1
4.7. WELL-DEFINED PROGRAMS.

shows sub-distrib-pconj (SetPC p P)
proof (rule sub-distrib-pconj1, simp add: wp-eval, rule le-funI)
fix Q::'s ⇒ real and R::'s ⇒ real and s::'s
assume uQ: unitary Q and uR: unitary R
have ((λs. ∑ a∈supp (P s). P s a * wp (p a) Q s) &&
(λs. ∑ a∈supp (P s). P s a * wp (p a) R s)) s =
(∑ a∈supp (P s). P s a * wp (p a) Q s) + (∑ a∈supp (P s). P s a * wp (p a) R s) ⊔ 1
by (simp add: exp-conj-def pconj-def)
also have ... = (∑ a∈supp (P s). P s a * (wp (p a) Q s + wp (p a) R s)) ⊔ 1
by (simp add: setsum.distrib field-simps)
also from sub
have ... ≤ (∑ a∈supp (P s). P s a * (wp (p a) Q s + wp (p a) R s)) ⊔
(∑ a∈supp (P s). P s a)
by (rule tminus-right-antimono)
also from fin
have ... ≤ (∑ a∈supp (P s). P s a * (wp (p a) Q s + wp (p a) R s) ⊔ P s a)
by (rule tminus-setsum-mono)
also from nnp
have ... = (∑ a∈supp (P s). P s a * (wp (p a) Q s + wp (p a) R s ⊔ 1))
by (simp add: tminus-left-distrib)
also have ... = (∑ a∈supp (P s). P s a * (wp (p a) Q & & wp (p a) R s))
by (simp add: pconj-def exp-conj-def)
also { from sdp uQ aR
have (∀a. a ∈ supp (P s) ⇒ wp (p a) Q & & wp (p a) R ⇒ wp (p a) (Q & & R))
by (blast intro: sub-distrib-pconjD)
with nnp
have (∑ a∈supp (P s). P s a * (wp (p a) Q & & wp (p a) R s) ≤
(∑ a∈supp (P s). P s a * (wp (p a) (Q & & R)) s)
by (blast intro: setsum-mono mult-left-mono)
}
finally show ((λs. ∑ a∈supp (P s). P s a * wp (p a) Q s) & &
(λs. ∑ a∈supp (P s). P s a * wp (p a) R s)) s ≤
(∑ a∈supp (P s). P s a * wp (p a) (Q & & R) s).
qed

lemma sdp-SetDC:
fixes p::'a ⇒ 's prog
assumes sdp: ∀s a. a ∈ S s ⇒ sub-distrib-pconj (p a)
and hwv: ∀s a. a ∈ S s ⇒ healthy (wp (p a))
and hwpv: ∀s a. a ∈ S s ⇒ nearly-healthy (wp (p a))
and ne: ∀s. S s ≠ {}
shows sub-distrib-pconj (SetDC p S)
proof (rule sub-distrib-pconj1, rule le-funI)
fix P::'s ⇒ real and Q::'s ⇒ real and s::'s
assume uP: unitary P and uQ: unitary Q
from \( uP \) \text{hwlp}
have \( \exists x. \ x \in (\lambda a. \ \text{wlp} (p \ a) \ P) \cdot \ S \vdash \text{unitary} \ x \ \text{by(auto)} \)
hence \( \exists y. \ y \in (\lambda a. \ \text{wlp} (p \ a) \ P) \cdot \ S \vdash 0 \leq y \ \text{by(auto)} \)
hence \( \exists a. \ a \in S \ s \Rightarrow \text{wlp} (\text{SetDC} p S) \ P \ s \leq \text{wlp} (p \ a) \ P \ s \)
  unfolding \text{wp-eval} \ \text{by(intro cInf-lower bdd-belowI, auto)}
moreover {  
  from \( uQ \) \text{hwp} have \( \exists a. \ a \in S \ s \Rightarrow 0 \leq (p \ a) \ Q \ s \ \text{by(blast)} \)
hence \( \exists a. \ a \in S \ s \Rightarrow \text{wp} (\text{SetDC} p S) \ Q \ s \leq \text{wp} (p \ a) \ Q \ s \)
  unfolding \text{wp-eval} \ \text{by(intro cInf-lower bdd-belowI, auto)}
}
ultimately
have \( \exists a. \ a \in S \ s \Rightarrow \text{wlp} (\text{SetDC} p S) \ P \ s + \text{wp} (\text{SetDC} p S) \ Q \ s \odot I \leq \text{wp} (p \ a) \ P \ s + \text{wp} (p \ a) \ Q \ s \odot I \)
  by(auto intro:tminus-left-mono add-mono)
also have \( \exists a. \ (p \ a) \ P \ s + \text{wp} (p \ a) \ Q \ s \odot I = (\text{wp} (p \ a) \ P \ &\& \text{wp} (p \ a) \ Q) \ s \)
  by(simp add:exp-conj-def pconj-def)
also from \( \text{sdp} \ uP \ uQ \)
have \( \exists a. \ a \in S \ s \Rightarrow \ldots \ a \leq (p \ a) \ (P \ &\& \ Q) \ s \)
  by(blast)
also have \( \exists a. \ \ldots \ a = (p \ a) \ (\lambda s. \ P \ s + Q \ s \odot I) \ s \)
  by(simp add:exp-conj-def pconj-def)
finally
show \( (\text{wp} (\text{SetDC} p S) \ P \ &\& \text{wp} (\text{SetDC} p S) \ Q) \ s \leq \text{wp} (\text{SetDC} p S) \ (P \ &\& \ Q) \ s \)
  unfolding \text{exp-conj-def pconj-def wp-eval}
  using ne by(blast intro::cInf-greatest)

\text{qed}

\text{lemma} \ \text{sdp-Bind:}
\[
[ \exists s. \ \text{sub-distrib-pconj} (p \ (f \ s)) ] \Rightarrow \text{sub-distrib-pconj} (\text{Bind} f p)
\]
unfolding \text{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.

\text{lemma} \ \text{sdp-loop:}
fixes \text{body::'s prog}
assumes \( \text{sdp-body: sub-distrib-pconj body} \)
  and \( \text{hwlp: nearly-healthy (wp body)} \)
  and \( \text{hwp: healthy (wp body)} \)
shows \( \text{sub-distrib-pconj (do G \Rightarrow body od)} \)
proof(rule sub-distrib-pconjI, rule loop-induct[OF hwp hwlp])
fix \( P \ Q::'s \text{ expect and } S::'(s \text{trans } x 's \text{trans}) \set\)
assume \( uP::\text{unitary P and } uQ::\text{unitary Q} \)
  and \text{fst: } \forall x \in S. \ \text{feasible (fst x)}
  and \text{snd: } \forall x \in S. \ \text{unitary (snd x Q)} \mapsto \text{unitary (snd x Q)}
  and \text{IH: } \forall x \in S. \ \text{snd x P } &\& \text{fst x Q } \Rightarrow \text{fst x (P } &\& \text{Q)
4.7. WELL-DEFINED PROGRAMS.

show Inf-utrans (snd ' S) P && Sup-trans (fst ' S) Q ⊢
        Sup-trans (fst ' S) (P && Q)

proof (cases)
  assume S = {}
  thus ?thesis
    by (simp add: Inf-trans-def Sup-trans-def Inf-utrans-def
          Inf-exp-def Sup-exp-def exp-conj-def)

next
  assume ne: S ≠ {}

  let ?f s = 1 + Sup-trans (fst ' S) (P && Q) s - Inf-utrans (snd ' S) P s

from ne obtain t where tin: t ∈ fst ' S by (auto)
from ne obtain u where uin: u ∈ snd ' S by (auto)

from tin ffst uP uQ have utPQ: unitary (t (P && Q))
  by (auto intro: exp-conj-unitary)

  hence ∀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) \implies R ≤ ?f
  proof (rule contrapos-pp, assumption)
    fix R
    assume ¬R ≤ ?f
    then obtain s where ¬R s ≤ ?f s by (auto)
    hence gt: ?f s < R s by (simp)

    from nn-rhs have g1: 1 ≤ 1 + Sup-trans (fst ' S) (P && Q) s by (auto)
    hence Sup-trans (fst ' S) (P && Q) s = Inf-utrans (snd ' S) P s && ?f s
      by (simp add: pconj-def)
    also from g1 have ... = Inf-utrans (snd ' S) P s + ?f s - 1
      by (simp)
    also from gt have ... < Inf-utrans (snd ' S) P s + R s - 1
      by (simp)
    also {
      with g1 have 1 ≤ Inf-utrans (snd ' S) P s + R s
        by (simp)
      hence Inf-utrans (snd ' S) P s + R s - 1 = Inf-utrans (snd ' S) P s && R s
        by (simp add: pconj-def)
    }
  }
finally
have \( \neg (\text{Inf-utrans} \ (\text{snd} \ ' S) \ P \land R) \ s \leq \text{Sup-trans} \ (\text{fst} \ ' S) \ (P \land Q) \ s \)
by(simp add:exp-conj-def)
thus \( \neg \text{Inf-utrans} \ (\text{snd} \ ' S) \ P \land R \proves \text{Sup-trans} \ (\text{fst} \ ' S) \ (P \land Q) \)
by(auto)
qed

moreover have \( \forall t \in \text{fst} \ ' S. \text{Inf-utrans} \ (\text{snd} \ ' S) \ P \land t \ Q \proves \text{Sup-trans} \ (\text{fst} \ ' S) \ (P \land Q) \)
proof
fix \( t \) assume \( \text{tin: } t \in \text{fst} \ ' S \)
then obtain \( x \) where \( \text{xin: } x \in S \) and \( \text{fx: } t = \text{fst} \ x \)
by(auto)

from \( \text{xin} \) have \( \text{snd} \ x \in \text{snd} \ ' S \)
by(auto)

with \( uP \ usnd \ have \ \text{Inf-utrans} \ (\text{snd} \ ' S) \ P \proves \text{snd} \ x \ P \)
by(auto intro:utransD[OF Inf-utrans-lower])

hence \( \text{Inf-utrans} \ (\text{snd} \ ' S) \ P \land \text{fst} \ x \ Q \proves \text{snd} \ x \ P \land \text{fst} \ x \ Q \)
by(auto intro:entails-frame)

also from \( \text{xin} \) \( \text{IH} \) have \( \ldots \proves \text{fst} \ x \ (P \land Q) \)
by(auto)

also from \( \text{xin} \) \( \text{ffst} \ \text{exp-conj-unitary} \)[OF \( uP \ \text{uQ} \)]

have \( \ldots \proves \text{Sup-trans} \ (\text{fst} \ ' S) \ (P \land Q) \)
by(auto intro:utransD[OF Sup-trans-upper])

finally show \( \text{Inf-utrans} \ (\text{snd} \ ' S) \ P \land t \ Q \proves \text{Sup-trans} \ (\text{fst} \ ' S) \ (P \land Q) \)
by(simp add:fx)
qed

ultimately have \( \text{bt: } \forall t \in \text{fst} \ ' S. \ t \ Q \proves \ ?f \)
by(blast)

have \( \text{Sup-trans} \ (\text{fst} \ ' S) \ Q = \text{Sup-exp} \ \{ t \ Q \mid t. t \in \text{fst} \ ' S \} \)
by(simp add:Sup-trans-def)
also have \( \ldots \proves \ ?f \)
proof(rule Sup-exp-least)
from \( \text{bt} \) show \( \forall R \in \{ t \ Q \mid t. t \in \text{fst} \ ' S \}. \ R \proves \ ?f \)
by(blast)
from \( \text{ne} \) obtain \( t \) where \( \text{tin: } t \in \text{fst} \ ' S \)
by(auto)

with \( \text{ffst} \ \text{uQ} \) have \( \text{unitary} \ (t \ Q) \)
by(auto)

hence \( \lambda s. 0 \proves t \ Q \)
by(auto)

also from \( \text{tin} \) \( \text{bt} \) have \( \ldots \proves \ ?f \)
by(auto)

finally show \( \text{mneg} \ (\lambda s. 1 + \text{Sup-trans} \ (\text{fst} \ ' S) \ (P \land Q) \ s) = \text{Inf-utrans} \ (\text{snd} \ ' S) \ P \ s) \)
by(auto)
qed

finally have \( \text{Inf-utrans} \ (\text{snd} \ ' S) \ P \land \text{Sup-trans} \ (\text{fst} \ ' S) \ Q \proves \ ?f \)
by(auto intro:entails-frame)
also from \( \text{nn-rhs} \) have \( \ldots \proves \text{Sup-trans} \ (\text{fst} \ ' S) \ (P \land Q) \)
by(simp add:exp-conj-def pconj-def)
finally show \( \text{thesis} \).
qed

next
4.7. WELL-DEFINED PROGRAMS.

\[\text{fix } P\ Q::'s \text{ expect and } t\ w::'s \text{ trans} \]
\[\text{assume } uP::\text{ unitary } P \text{ and } uQ::\text{ unitary } Q \]
\[\text{and } ft::\text{ feasible } t \]
\[\text{and } uu::\text{ unitary } Q \text{ } \Rightarrow \text{ unitary } (u\ Q) \]
\[\text{and } IH:: u\ P \&\& t\ Q \Rightarrow t\ (P\ &\& Q) \]
\[\text{show } wlp\ (\text{body} :: \text{Embed } u::'G::\oplus\text{ Skip} P\ &\&\]
\[wp\ (\text{body} :: \text{Embed } t::'G::\oplus\text{ Skip} Q\ \Rightarrow\]
\[wp\ (\text{body} :: \text{Embed } t::'G::\oplus\text{ Skip} (P\ &\& Q) \]
\[\text{proof}(\text{rule } \text{le-fun1}, \text{simp add:wp-eval exp-conj-def pconj-def}) \]
\[\text{fix } s::'s \]
\[\text{have } «G»\ s\ *\ wp\ \text{body}\ (u\ P)\ s\ +\ (1\ −\ «G»\ s)\ *\ P\ s\ +\]
\[«G»\ s\ *\ wp\ \text{body}\ (t\ Q)\ s\ +\ (1\ −\ «G»\ s)\ *\ Q\ s\ \odot\ 1\ =\]
\[«G»\ s\ *\ wp\ \text{body}\ (u\ P)\ s\ +\ «G»\ s\ *\ wp\ \text{body}\ (t\ Q)\ s\ +\]
\[(1\ −\ «G»\ s)\ *\ P\ s\ +\ (1\ −\ «G»\ s)\ *\ Q\ s\ \odot\ («G»\ s\ +\ (1\ −\ «G»\ s))\]
\[\text{by(\text{simp add:ac-simps})} \]
\[\text{also have } \ldots\ \leq\]
\[«G»\ s\ *\ wp\ \text{body}\ (u\ P)\ s\ +\ «G»\ s\ *\ wp\ \text{body}\ (t\ Q)\ s\ \odot\ «G»\ s\ +\]
\[(1\ −\ «G»\ s)\ *\ P\ s\ +\ (1\ −\ «G»\ s)\ *\ Q\ s\ \odot\ (1\ −\ «G»\ s))\]
\[\text{by(\text{rule } \text{tminus-add-mono})} \]
\[\text{also have } \ldots\ =\]
\[«G»\ s\ *\ (wp\ \text{body}\ (u\ P)\ s\ +\ wp\ \text{body}\ (t\ Q)\ s\ \odot\ 1)\ +\]
\[(1\ −\ «G»\ s)\ *\ (P\ s\ +\ Q\ s\ \odot\ 1)\]
\[\text{by(\text{simp add:tminus-left-distrib distrib-left})} \]
\[\text{also \{} \]
\[\text{from } uP\ uQ\ ft\ uu \]
\[\text{have } wp\ \text{body}\ (u\ P)\ &\&\ wp\ \text{body}\ (t\ Q)\ \Rightarrow\ wp\ \text{body}\ (u\ P\ &\&\ t\ Q)\]
\[\text{by(\text{auto intro:sub-distrib-pconjD}(OF \text{ sdp-body}))} \]
\[\text{also from } IH\ \text{unitary-sound}(OF\ uP]\ \text{unitary-sound}(OF\ uQ)\ ft\]
\[\text{unitary-sound}(OF\ uu)(OF\ uP)]\]
\[\text{have } \ldots\ \leq\ wp\ \text{body}\ (t\ (P\ &\&\ Q))\]
\[\text{by(\text{blast intro:mono-transD}(OF\ healthy-monoD,\ OF\ hwp)\ exp-conj-sound}) \]
\[\text{finally have } wp\ \text{body}\ (u\ P)\ s\ +\ wp\ \text{body}\ (t\ Q)\ s\ \odot\ 1\ \leq\]
\[wp\ \text{body}\ (t\ (\lambda s.\ P\ s\ +\ Q\ s\ \odot\ 1))\ s\]
\[\text{by(\text{auto simp:exp-conj-def pconj-def})} \]
\[\text{hence } «G»\ s\ *\ (wp\ \text{body}\ (u\ P)\ s\ +\ wp\ \text{body}\ (t\ Q)\ s\ \odot\ 1)\ +\]
\[(1\ −\ «G»\ s)\ *\ (P\ s\ +\ Q\ s\ \odot\ 1)\ \leq\]
\[«G»\ s\ *\ wp\ \text{body}\ (t\ (\lambda s.\ P\ s\ +\ Q\ s\ \odot\ 1))\ s\ +\]
\[(1\ −\ «G»\ s)\ *\ (P\ s\ +\ Q\ s\ \odot\ 1)\]
\[\text{by(\text{auto intro:add-right-mono mult-left-mono})} \}
\[\text{finally} \]
\[\text{show } «G»\ s\ *\ wp\ \text{body}\ (u\ P)\ s\ +\ (1\ −\ «G»\ s)\ *\ P\ s\ +\]
\[(«G»\ s\ *\ wp\ \text{body}\ (t\ Q)\ s\ +\ (1\ −\ «G»\ s)\ *\ Q\ s\ \odot\ 1\ \leq\]
\[«G»\ s\ *\ wp\ \text{body}\ (t\ (\lambda s.\ P\ s\ +\ Q\ s\ \odot\ 1))\ s\ +\]
\[(1\ −\ «G»\ s)\ *\ (P\ s\ +\ Q\ s\ \odot\ 1)\ . \]
\[\text{qed} \]
\[\text{next} \]
\[\text{fix } P\ Q::'s \text{ expect and } t\ t'\ u::'s \text{ trans} \]
\[\text{assume } \text{unitary } P\ \text{unitary } Q \]
equiv-trans \ t \ t' \ equiv-utrans \ u \ u'
\ u \ P \ \&\& \ t \ P \ \&\& \ Q
\therefore \ u' \ P \ \&\& \ t' \ P \ \&\& \ Q
\text{by (simp add: equiv-transD unitary-sound equiv-utransD exp-conj-unitary)}
\text{qed}

\text{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.

\text{definition well-def :: 's prog ⇒ bool}
\text{where}
\begin{align*}
\text{well-def prog} & \equiv \text{healthy (wp prog)} \land \text{nearly-healthy (wlp prog)} \\
& \phantom{\equiv \text{healthy (wp prog)}} \land \text{wp-under-wlp prog} \land \text{sub-distrib-pconj prog} \\
& \phantom{\equiv \text{healthy (wp prog)} \land \text{nearly-healthy (wlp prog)}} \land \text{sublinear (wp prog)} \land \text{bd-cts (wp prog)}
\end{align*}

\text{lemma well-defI [intro]:}
\begin{align*}
\text{well-defI prog} & \equiv \text{healthy (wp prog)} \land \text{nearly-healthy (wlp prog)} \\
& \phantom{\equiv \text{healthy (wp prog)}} \land \text{wp-under-wlp prog} \land \text{sub-distrib-pconj prog} \\
& \phantom{\equiv \text{healthy (wp prog)} \land \text{nearly-healthy (wlp prog)}} \land \text{sublinear (wp prog)} \land \text{bd-cts (wp prog)}
\end{align*}
\text{unfolding well-def-def by (simp)}

\text{lemma well-def-wp-healthy [dest]:}
\begin{align*}
\text{well-defI prog} & \equiv \text{healthy (wp prog)}
\end{align*}
\text{unfolding well-def-def by (simp)}

\text{lemma well-def-wlp-nearly-healthy [dest]:}
\begin{align*}
\text{well-defI prog} & \equiv \text{nearly-healthy (wlp prog)}
\end{align*}
\text{unfolding well-def-def by (simp)}

\text{lemma well-def-wp-under [dest]:}
\begin{align*}
\text{well-defI prog} & \equiv \text{wp-under-wlp prog}
\end{align*}
\text{unfolding well-def-def by (simp)}

\text{lemma well-def-sdp [dest]:}
\begin{align*}
\text{well-defI prog} & \equiv \text{sub-distrib-pconj prog}
\end{align*}
\text{unfolding well-def-def by (simp)}

\text{lemma well-def-wp-sublinear [dest]:}
\begin{align*}
\text{well-defI prog} & \equiv \text{sublinear (wp prog)}
\end{align*}
\text{unfolding well-def-def by (simp)}

\text{lemma well-def-wp-cts [dest]:}
4.7. WELL-DEFINED PROGRAMS.

well-def prog \implies bd-cts (wp prog)

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

\textbf{lemmas} \textbf{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

\textbf{lemma} \textbf{wd-Abort}:
well-def Abort
\textbf{by}(blast intro: healthy-wp-Abort nearly-healthy-wlp-Abort
wp-under-wlp-Abort sdp-Abort sublinear-wp-Abort
cts-wp-Abort)

\textbf{lemma} \textbf{wd-Skip}:
well-def Skip
\textbf{by}(blast intro: healthy-wp-Skip nearly-healthy-wlp-Skip
wp-under-wlp-Skip sdp-Skip sublinear-wp-Skip
cts-wp-Skip)

\textbf{lemma} \textbf{wd-Apply}:
well-def (Apply f)
\textbf{by}(blast intro: healthy-wp-Apply nearly-healthy-wlp-Apply
wp-under-wlp-Apply sdp-Apply sublinear-wp-Apply
cts-wp-Apply)

\textbf{lemma} \textbf{wd-Seq}:
[
    \text{well-def a; well-def b}
] \implies
well-def (a ;; b)
\textbf{by}(blast intro: healthy-wp-Seq nearly-healthy-wlp-Seq
wp-under-wlp-Seq sdp-Seq sublinear-wp-Seq
cts-wp-Seq)

\textbf{lemma} \textbf{wd-PC}:
[
    \text{well-def a; well-def b; unitary P}
] \implies
well-def (a \sqcup b)
\textbf{by}(blast intro: healthy-wp-PC nearly-healthy-wlp-PC
wp-under-wlp-PC sdp-PC sublinear-wp-PC
cts-wp-PC)

\textbf{lemma} \textbf{wd-DC}:
[
    \text{well-def a; well-def b}
] \implies
well-def (a \sqcap b)
\textbf{by}(blast intro: healthy-wp-DC nearly-healthy-wlp-DC
wp-under-wlp-DC sdp-DC sublinear-wp-DC
cts-wp-DC)

\textbf{lemma} \textbf{wd-SetDC}:
[
    \forall x. s. x \in S__(s __) \implies well-def (a x);
    \forall s. S s \neq \{\}
    \forall s. finite (S s) \implies well-def (SetDC a S)
] \implies
well-def (SetDC a S)
\textbf{by}(iprover intro: well-defII healthy-wp-SetDC nearly-healthy-wlp-SetDC
wp-under-wlp-SetDC sdp-SetDC sublinear-wp-SetDC cts-wp-SetDC
nonempty-witness
dest:wd-dests)

lemma wd-SetPC:
\[ \forall s. x \in (\operatorname{supp}\ (p\ s)) \implies \operatorname{well-def}\ (a\ x) ; \forall s. \operatorname{unitary}\ (p\ s) ; \forall s. \operatorname{finite}\ (\operatorname{supp}\ (p\ s)) ; \forall s. \operatorname{setsum}\ (p\ s) \leq 1 \implies \operatorname{well-def}\ (\operatorname{SetPC}\ a\ p) \]
by (iprover intro: well-defI healthy-wp-SetPC nearly-healthy-wlp-SetPC
well-under-wlp-SetPC sdp-SetPC sublinear-wp-SetPC cts-wp-SetPC
dest:wd-dests unitary-sound sound-nneg)

lemma wd-Embed:
\[ \textit{fixes} t :: \prime s \textit{ trans} \]
\[ \textit{assumes} ht: \text{healthy}\ t \textit{ and} st: \text{sublinear}\ t \textit{ and} ct: \text{bd-cts} t \]
\[ \textit{shows} \ \operatorname{well-def}\ (\text{Embed}\ t) \]
proof (intro well-defI)
from ht show healthy (wp (Embed t)) nearly-healthy (wlp (Embed t))
by (simp add: wp-def wp-def Embed-def healthy-nearly-healthy+)
from st show sublinear (wp (Embed t)) by (simp add: wp-def wp-def Embed-def)
show wp-under-wlp (Embed t) by (simp add: wp-under-wlp-def wp-eval)
show sub-distrib-pconj (Embed t)
by (rule sub-distrib-pconjI auto intro: le-funI [OF sublinearD [OF st where a=1 and b=1 and c=1, simplified]])
from ct show bd-cts (wp (Embed t))
by (simp add: wp-def Embed-def)
qed

lemma wd-repeat:
\[ \operatorname{well-def}\ a \implies \operatorname{well-def}\ (\text{repeat}\ n\ a) \]
by (blast intro: healthy-wp-repeat nearly-healthy-wlp-repeat
well-under-wlp-repeat sdp-repeat sublinear-wp-repeat cts-wp-repeat)

lemma wd-Bind:
\[ \forall s. \operatorname{well-def}\ (a\ (f\ s)) \implies \operatorname{well-def}\ (\text{Bind}\ f\ a) \]
by (blast intro: healthy-wp-Bind nearly-healthy-wlp-Bind
well-under-wlp-Bind sdp-Bind sublinear-wp-Bind cts-wp-Bind)

lemma wd-loop:
\[ \operatorname{well-def}\ \text{body} \implies \operatorname{well-def}\ (\text{do}\ G\ \rightarrow\ \text{body}\ od) \]
by (blast intro: healthy-wp-loop nearly-healthy-wlp-loop
well-under-wlp-loop sdp-loop sublinear-wp-loop cts-wp-loop)

lemmas wd-intros =
wd-Abort wd-Skip wd-Apply
wd-Embed wd-Seq wd-PC
wd-DC wd-SetPC wd-SetDC
wd-Bind wd-repeat wd-loop
4.8 The Loop Rules

theory Loops imports WellDefined begin

Given a well-defined body, we can annotate a loop using an invariant, just as in the classical setting.

4.8.1 Liberal and Strict Invariants.

A probabilistic invariant generalises a boolean one: it entails itself, given the loop guard.

definition wp-inv :: (′s ⇒ bool) ⇒ ′s prog ⇒ (′s ⇒ real) ⇒ bool
where
wp-inv G body I ←→ (∀ s. ≼G s * I s ≤ wp body I s)

lemma wp-invI:
∀ I. (∀ s. ≼G s * I s ≤ wp body I s) ⇒ wp-inv G body I
by(simp add:wp-inv-def)

definition wlp-inv :: (′s ⇒ bool) ⇒ ′s prog ⇒ (′s ⇒ real) ⇒ bool
where
wlp-inv G body I ←→ (∀ s. ≼G s * I s ≤ wlp body I s)

lemma wlp-invI:
∀ I. (∀ s. ≼G s * I s ≤ wlp body I s) ⇒ wlp-inv G body I
by(simp add:wlp-inv-def)

lemma wlp-invD:
wlp-inv G body I =⇒ ≼G s * I s ≤ wlp body I s
by(simp add:wlp-inv-def)

For standard invariants, the multiplication reduces to conjunction.

lemma wp-inv-stdD:
assumes inv: wp-inv G body «I»
and hb: healthy (wp body)
shows «G» & «I» ⊢ wp body «I»
proof(rule le-funI)
fix s
show («G» & «I») s ≤ wp body «I» s
proof(cases G s)
  case False
  with hb show ?thesis
by(auto simp:exp-conj-def)
next
case True
  hence («G» && «I») s = «G» s * «I» s
  by(simp add:exp-conj-def)
also from inv have «G» s * «I» s ≤ wp body «I» s
  by(simp add:wp-inv-def)
finally show thesis .
qed

4.8.2 Partial Correctness


lemma wlp-Loop:
  assumes wd: well-def body
    and uI: unitary I
    and inv: wlp-inv G body I
  shows I ≤ wp do G −→ body od (λs. «N G» s * I s)
    (is I ≤ wp do G −→ body od ?P)
proof −
  let ?f Q s = «G» s * wlp body Q s + «N G» s * ?P s
  have I ⊢ ⊢ gfp-exp ?f
    proof
      (rule gfp-exp-upperbound[OF _ uI]
        have I = (λs. («G» s + «N G» s) * I s) by(simp add:negate-embed)
        also have ... = (λs. «G» s * I s + «N G» s * I s)
          by(simp add:algebra-simps)
        also have ... = (λs. «G» s * («G» s * I s) + «N G» s * («N G» s * I s))
          by(simp add:embed-bool-idem algebra-simps)
        also have ... ⊢ (λs. «G» s * wp body I s + «N G» s * («N G» s * I s))
          using inv by(auto dest:wp-invD intro:add-mono mult-left-mono)
        finally show I ⊢ (λs. «G» s * wp body I s + «N G» s * («N G» s * I s)) .
        qed
      also from uI well-def-wlp-nearly-healthy[OF wd] have ... = wp do G −→ body od ?P
        by(auto intro!:wlp-Loop1[symmetric] unitary-intros)
      finally show thesis .
    qed

4.8.3 Total Correctness

The first total correctness lemma for loops which terminate with probability 1[McIver and Morgan, 2004, Lemma 7.3.1, §7, p. 186].

lemma wp-Loop:
  assumes wd: well-def body
    and inv: wp-inv G body I
    and unit: unitary I
4.8. THE LOOP RULES

shows \( I \land \& \& wp (do G \rightarrow body \od) (\lambda s. 1) \vdash wp (do G \rightarrow body \od) (\lambda s. \langle N \rangle s \star I s) \)
\( (is I \land \& \& ?T \vdash wp \; ?loop \; ?X) \)

proof —

We first appeal to the liberal loop rule:

from assms have \( I \land \& \& ?T \vdash wp \; ?loop \; ?X \land \& \& ?T \)
by (blast intro: exp-conj-mono-left wp-Loop)

Next, by sub-conjunctivity:

also {
  from wd have sdp-loop: sub-distrib-pconj (do G \rightarrow body \od)
  by (blast intro: sdp-intros)
  from wd unit have wp \; ?loop \; ?X \land \& \& ?T \vdash wp \; ?loop \; (?X \land \& \& (\lambda s. 1))
  by (blast intro: sub-distrib-pconjD sdp-intros unitary-intros)
}

Finally, the conjunction collapses:

finally show \( ?thesis \)
by (simp add: exp-conj-1-right sound-intros sound-nneg unit unitary-sound)
qed

4.8.4 Unfolding

lemma wp-loop-unfold:
fixes body :: 's prog
assumes \( sP: \text{sound } P \)
and \( h: \text{healthy } (wp \text{ body}) \)
shows wp (do G \rightarrow body \od) P =
\( (\lambda s. \langle N \rangle G \ s \star P \ s + \langle G \rangle s \star wp \text{ body } (wp \ (do G \rightarrow body \od) \ P) \ s) \)

unfolding wp-eval

proof —
let \( ?X \ t = wp \ (\text{body };; \text{Embed } t \ s G \oplus \text{Skip}) \)
have equiv-trans (lfp-trans \ ?X)
  (wp \ (\text{body };; \text{Embed } (lfp-trans \ ?X) \ s G \oplus \text{Skip}))
proof (intro lfp-trans-unfold)
  fix \( t::'s \text{ trans and } P::'s \text{ expect} \)
  assume \( st: \bigwedge Q \ . \text{sound } Q \implies \text{sound } (t \ Q) \)
  and \( sP: \text{sound } P \)
with \( h \) show \( \text{sound } (?X \ t \ P) \)
by (rule wp-loop-step-sound)

next
fix \( t \ u::'s \text{ trans} \)
assume le-trans \( t \ u \ (\bigwedge P \ . \text{sound } P \implies \text{sound } (t \ P)) \)
\( (\bigwedge P \ . \text{sound } P \implies \text{sound } (u \ P)) \)
with \( h \) show le-trans (wp \ (\text{body };; \text{Embed } t \ s G \oplus \text{Skip}))
  (wp \ (\text{body };; \text{Embed } u \ s G \oplus \text{Skip}))
by (iprover intro: wp-loop-step-mono)
CHAPTER 4. THE PGCL LANGUAGE

next
let \(?v = \lambda P \cdot \text{bound-of } P\)
from h show le-trans (wp (body ;; Embed ?v :: G \oplus \text{Skip})) ?v
  by (intro le-transI, simp add: wp-eval lfp-loop-fp [unfolded negate-embed])
fix P::'s expect
assume sound P thus sound (?v P) by (auto)
qed
also have equiv-trans ...
  (\lambda P s. «N G» s \ast P s + «G» s \ast 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 =
  (\lambda s. «N G» s \ast P s + «G» s \ast wp body (lfp-trans ?X P) s)
  using sP unfolding wp-eval by (blast)
qed

lemma wp-loop-nguard:
[ [ healthy (wp body); sound P; \neg G s ] \implies wp do G \rightarrow body od P s = P s
by (subst wp-loop-unfold, simp-all) ]

lemma wp-loop-guard:
[ [ healthy (wp body); sound P; G s ] \implies wp do G \rightarrow body od P s = wp (body ;; do G \rightarrow body od) P s
by (subst wp-loop-unfold, simp-all add: wp-eval) ]
end

4.9 The Algebra of pGCL

theory Algebra imports WellDefined begin

Programs in pGCL have a rich algebraic structure, largely mirroring that for GCL. We show that programs form a lattice under refinement, with a \(\sqcap\) \(b\) and \(a \sqcup b\) as the meet and join operators, respectively. We also take advantage of the algebraic structure to establish a framework for the modular decomposition of proofs.

4.9.1 Program Refinement

Refinement in pGCL relates to refinement in GCL exactly as probabilistic entailment relates to implication. It turns out to have a very similar algebra, the rules of which we establish shortly.

definition refines :: 's prog \Rightarrow 's prog \Rightarrow bool (infix \sqsubseteq 70)
where
  prog \sqsubseteq prog' \equiv \forall P. sound P \implies wp prog P \Rightarrow wp prog' P

lemma refinesI[intro]:
4.9. THE ALGEBRA OF PGCL

\[ \wedge P, \text{sound } P \Rightarrow wp \text{ prog } P \vdash wp \text{ prog'} P \implies \text{prog } \sqsubseteq \text{prog'} \]

unfolding refines-def by(simp)

\textbf{lemma refinesD[dest]:}
\[ [ \text{prog } \sqsubseteq \text{prog'}; \text{sound } P ] \implies wp \text{ prog } P \vdash wp \text{ prog'} P \]

unfolding refines-def by(simp)

The equivalence relation below will turn out to be that induced by refinement. It is also the application of \textit{equiv-trans} to the weakest precondition.

\textbf{definition pequiv :: 's prog \Rightarrow 's prog \Rightarrow bool (infix \simeq 70)}

where
\[ \text{prog } \simeq \text{prog'} \equiv \forall P. \text{sound } P \Rightarrow wp \text{ prog } P = wp \text{ prog'} P \]

\textbf{lemma pequivI[intro]:}
\[ [ \wedge P, \text{sound } P \Rightarrow wp \text{ prog } P = wp \text{ prog'} P ] \implies \text{prog } \simeq \text{prog'} \]

unfolding pequiv-def by(simp)

\textbf{lemma pequivD[dest,simp]:}
\[ [ \text{prog } \simeq \text{prog'}; \text{sound } P ] \implies wp \text{ prog } P = wp \text{ prog'} P \]

unfolding pequiv-def by(simp)

\textbf{lemma pequiv-equiv-trans:}
\[ a \simeq b \iff \text{equiv-trans } (wp \ a) (wp \ b) \]

by(auto)

4.9.2 Simple Identities

The following identities involve only the primitive operations as defined in Section 4.1.1, and refinement as defined above.

\textbf{Laws following from the basic arithmetic of the operators separately}

\textbf{lemma DC-comm[ac-simps]:}
\[ a \sqcap b = b \sqcap a \]

unfolding DC-def by(simp add:ac-simps)

\textbf{lemma DC-assoc[ac-simps]:}
\[ a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c \]

unfolding DC-def by(simp add:ac-simps)

\textbf{lemma DC-idem:}
\[ a \sqcap a = a \]

unfolding DC-def by(simp)

\textbf{lemma AC-comm[ac-simps]:}
\[ a \sqcup b = b \sqcup a \]
unfolding AC-def by (simp add: ac-simps)

lemma AC-assoc [ac-simps]:
\[ a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c \]
unfolding AC-def by (simp add: ac-simps)

lemma AC-idem:
\[ a \sqcup a = a \]
unfolding AC-def by (simp)

lemma PC-quasi-comm:
\[ a \oplus b = b (\lambda s. t - p s) \oplus a \]
unfolding PC-def by (simp add: algebra-simps)

lemma PC-idem:
\[ a \oplus a = a \]
unfolding PC-def by (simp add: algebra-simps)

lemma Seq-assoc [ac-simps]:
\[ A ;; (B ;; C) = A ;; B ;; C \]
by (simp add: Seq-def o-def)

lemma Abort-refines [intro]:
\[ \text{well-def } a \Rightarrow \text{Abort } \sqsubseteq a \]
by (rule refinesI, unfold wp-eval, auto dest: well-def-wp-healthy)

Laws relating demonic choice and refinement

lemma left-refines-DC:
\[ (a \sqcap b) \sqsubseteq a \]
by (auto intro!: refinesI simp: wp-eval)

lemma right-refines-DC:
\[ (a \sqcap b) \sqsubseteq b \]
by (auto intro!: refinesI simp: wp-eval)

lemma DC-refines:
fixes a::'s prog and b and c
assumes rab: a \sqsubseteq b and rac: a \sqsubseteq c
shows a \sqsubseteq (b \sqcap c)
proof
  fix P::'s \Rightarrow \text{real } assume sP: sound P
  with assms have wp a P \vdash wp b P and wp a P \vdash wp c P
    by (auto dest: refinesD)
  thus wp a P \vdash wp (b \sqcap c) P
    by (auto simp: wp-eval intro:min.boundedI)
qed

lemma DC-mono:
4.9. THE ALGEBRA OF PGCL

fixes $a::s$ prog
assumes $rab$: $a \subseteq b$ and $rcd$: $c \subseteq d$
shows $(a \bigcup c) \subseteq (b \bigcup d)$

proof (rule refinesI, unfold wp-eval, rule le-funI)
  fix $P::s$ ⇒ real and $s::s$
  assume $sP$: sound $P$
  with assms have $wp a P s \leq wp b P s$ and $wp c P s \leq wp d P s$
    by (auto)
  thus $\min (wp a P s) (wp c P s) \leq \min (wp b P s) (wp d P s)$
    by (auto)
qed

Laws relating angelic choice and refinement

lemma left-refines-AC:
  $a \subseteq (a \bigcup b)$
  by (auto intro!: refinesI simp: wp-eval)

lemma right-refines-AC:
  $b \subseteq (a \bigcup b)$
  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 $(a \bigcup b) \subseteq c$

proof
  fix $P::s$ ⇒ real assume $sP$: sound $P$
  with assms have $\forall s. wp a P s \leq wp c P s$
    and $\forall s. wp b P s \leq wp c P s$
    by (auto dest!: refinesD)
  thus $wp (a \bigcup b) P \vdash wp c P$
    unfolding wp-eval by (auto)
qed

lemma AC-mono:
  fixes $a::s$ prog
  assumes rac: $a \subseteq b$ and rcd: $c \subseteq d$
  shows $(a \bigcup c) \subseteq (b \bigcup d)$

proof (rule refinesI, unfold wp-eval, rule le-funI)
  fix $P::s$ ⇒ real and $s::s$
  assume $sP$: sound $P$
  with assms have $wp a P s \leq wp b P s$ and $wp c P s \leq wp d P s$
    by (auto)
  thus $\max (wp a P s) (wp c P s) \leq \max (wp b P s) (wp d P s)$
    by (auto)
qed
Laws depending on the arithmetic of $a \oplus b$ and $a \sqcup b$ together

**Lemma DC-refines-PC:**
assumes unit: unitary $p$
shows $(a \sqcup b) \sqsubseteq (a \oplus b)$

**Proof:**
\begin{align*}
& \text{fix } s \text{ and } P::'a \Rightarrow \mathbb{R} \\
& \text{assume sound: sound } P \\
& \text{from } \text{unit have } \mathrm{nn-p}: 0 \leq p \ s \ \text{by(\text{blast})} \\
& \text{from } \text{unit have } p \ s \leq 1 \ \text{by(\text{blast})} \\
& \text{hence } \text{nn-np}: 0 \leq 1 - p \ s \ \text{by(\text{simp})} \\
& \text{show } \min (\text{wp } a \ P \ s) (\text{wp } b \ P \ s) \leq p \ s \ast \text{wp } a \ P \ s + (1 - p \ s) \ast \text{wp } b \ P \ s \\
\end{align*}

**Proof:**
\begin{align*}
& \text{cases } \text{wp } a \ P \ s \leq \text{wp } b \ P \ s, \\
& \text{simp-all add:min.absorb1 min.absorb2} \\
& \text{case True note } \leq = \text{this} \\
& \text{have } \text{wp } a \ P \ s = (p \ s + (1 - p \ s)) \ast \text{wp } a \ P \ s \ \text{by(\text{simp})} \\
& \text{also have } \ldots = p \ s \ast \text{wp } a \ P \ s + (1 - p \ s) \ast \text{wp } a \ P \ s \\
& \text{by(\text{simp only: distrib-right})} \\
& \text{also } \{ \\
& \text{from } \leq \text{and } \text{nn-np have } (1 - p \ s) \ast \text{wp } a \ P \ s \leq (1 - p \ s) \ast \text{wp } b \ P \ s \\
& \text{by(\text{rule mult-left-mono})} \\
& \text{hence } p \ s \ast \text{wp } a \ P \ s + (1 - p \ s) \ast \text{wp } a \ P \ s \leq \\
& p \ s \ast \text{wp } a \ P \ s + (1 - p \ s) \ast \text{wp } b \ P \ s \\
& \text{by(\text{rule add-left-mono})} \\
& \} \\
& \text{finally show } \text{wp } a \ P \ s \leq p \ s \ast \text{wp } a \ P \ s + (1 - p \ s) \ast \text{wp } b \ P \ s. \\
\text{next} \\
& \text{case False} \\
& \text{then have } \leq: \text{wp } b \ P \ s \leq \text{wp } a \ P \ s \ \text{by(\text{simp})} \\
& \text{have } \text{wp } b \ P \ s = (p \ s + (1 - p \ s)) \ast \text{wp } b \ P \ s \ \text{by(\text{simp})} \\
& \text{also have } \ldots = p \ s \ast \text{wp } b \ P \ s + (1 - p \ s) \ast \text{wp } b \ P \ s \\
& \text{by(\text{simp only: distrib-right})} \\
& \text{also } \{ \\
& \text{from } \leq \text{and } \text{nn-p have } p \ s \ast \text{wp } b \ P \ s \leq p \ s \ast \text{wp } a \ P \ s \\
& \text{by(\text{rule mult-left-mono})} \\
& \text{hence } p \ s \ast \text{wp } b \ P \ s + (1 - p \ s) \ast \text{wp } b \ P \ s \leq \\
& p \ s \ast \text{wp } a \ P \ s + (1 - p \ s) \ast \text{wp } b \ P \ s \\
& \text{by(\text{rule add-right-mono})} \\
& \} \\
& \text{finally show } \text{wp } b \ P \ s \leq p \ s \ast \text{wp } a \ P \ s + (1 - p \ s) \ast \text{wp } b \ P \ s. \\
& \text{qed} \\
& \text{qed}
\end{align*}

Laws depending on the arithmetic of $a \oplus b$ and $a \sqcup b$ together

**Lemma PC-refines-AC:**
assumes unit: unitary $p$
shows $(a \oplus b) \sqsubseteq (a \sqcup b)$

**Proof:**
\begin{align*}
& \text{fix } s \text{ and } P::'a \Rightarrow \mathbb{R} \\
& \text{assume sound: sound } P \\
& \text{from } \text{unit have } \mathrm{nn-p}: 0 \leq p \ s \ \text{by(\text{blast})} \\
& \text{from } \text{unit have } p \ s \leq 1 \ \text{by(\text{blast})} \\
& \text{hence } \text{nn-np}: 0 \leq 1 - p \ s \ \text{by(\text{simp})} \\
& \text{show } \min (\text{wp } a \ P \ s) (\text{wp } b \ P \ s) \leq p \ s \ast \text{wp } a \ P \ s + (1 - p \ s) \ast \text{wp } b \ P \ s \\
\end{align*}
from unit have \( p \cdot s \leq 1 \) by (blast)

hence \( nn-np: 0 \leq 1 - p \cdot s \) by (simp)

show \( p \cdot s \cdot \wp a \cdot P \cdot s + (1 - p \cdot s) \cdot \wp b \cdot P \cdot s \leq \max (\wp a \cdot P \cdot s) (\wp b \cdot P \cdot s) \)

proof (cases \( \wp a \cdot P \cdot s \leq \wp b \cdot P \cdot s \))
  case True

  note leab = this

  with unit nn-np

  have \( p \cdot s \cdot \wp a \cdot P \cdot s + (1 - p \cdot s) \cdot \wp b \cdot P \cdot s \leq \)

  \( p \cdot s \cdot \wp b \cdot P \cdot s + (1 - p \cdot s) \cdot \wp b \cdot P \cdot s \)

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

  also have \( \ldots = \wp b \cdot P \cdot s \)

  by (auto simp: field-simps)

  also from leab

  have \( \ldots = \max (\wp a \cdot P \cdot s) (\wp b \cdot P \cdot s) \)

  by (auto)

  finally show ?thesis .

next

case False

  note leba = this

  with unit nn-np

  have \( p \cdot s \cdot \wp a \cdot P \cdot s + (1 - p \cdot s) \cdot \wp b \cdot P \cdot s \leq \)

  \( p \cdot s \cdot \wp a \cdot P \cdot s + (1 - p \cdot s) \cdot \wp a \cdot P \cdot s \)

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

  also have \( \ldots = \wp a \cdot P \cdot s \)

  by (auto simp: field-simps)

  also from leba

  have \( \ldots = \max (\wp a \cdot P \cdot s) (\wp b \cdot P \cdot s) \)

  by (auto)

  finally show ?thesis .

qed

Laws depending on the arithmetic of \( a \bigcup b \) and \( a \bigcap b \) together

lemma DC-refines-AC:

\( (a \bigcap b) \subseteq (a \bigcup b) \)

by (auto intro!: refinesI simp: wp-eval)

Laws Involving Refinement and Equivalence

lemma pr-trans[trans]:

fixes \( A::'a \ prog \)

assumes \( prAB: A \sqsubseteq B \)

and \( prBC: B \sqsubseteq C \)

shows \( A \sqsubseteq C \)

proof

fix \( P::'a \Rightarrow real \)

assume \( sP: \ sound \ P \)

with \( prAB \) have \( \wp A \cdot P \vdash \wp B \cdot P \) by (blast)

also from \( sP \) and \( prBC \) have \( \ldots \vdash \wp C \cdot P \) by (blast)

finally show \( \wp A \cdot P \vdash \ldots \).

qed
CHAPTER 4. THE PGCL LANGUAGE

qed

lemma pequiv-refl[intro,simp]:
  \( a \simeq a \)
  by(auto)

lemma pequiv-comm[ac-simps]:
  \( a \simeq b \iff b \simeq a \)
  unfolding pequiv-def
  by(rule iffI, safe, simp-all)

lemma pequiv-pr[dest]:
  \( a \simeq b \Rightarrow a \subseteq b \)
  by(auto)

lemma pequiv-trans[intro,trans]:
  \( [a \simeq b; b \simeq c] \Rightarrow a \simeq c \)
  unfolding pequiv-def by(auto intro!:order-trans)

lemma pequiv-pr-trans[intro,trans]:
  \( [a \subseteq b; b \simeq c] \Rightarrow a \subseteq c \)
  unfolding pequiv-def by(simp)

lemma pr-pequiv-trans[intro,trans]:
  \( [a \subseteq b; b \simeq c] \Rightarrow a \subseteq c \)
  unfolding pequiv-def by(simp)

Refinement induces equivalence by antisymmetry:

lemma pequiv-antisym:
  \( [a \subseteq b; b \subseteq a] \Rightarrow a \simeq b \)
  by(auto intro:antisym)

lemma pequiv-DC:
  \( [a \simeq c; b \simeq d] \Rightarrow (a \sqcap b) \simeq (c \sqcap d) \)
  by(auto intro!:DC-mono pequiv-antisym simp:ac-simps)

lemma pequiv-AC:
  \( [a \simeq c; b \simeq d] \Rightarrow (a \sqcup b) \simeq (c \sqcup d) \)
  by(auto intro!:AC-mono pequiv-antisym simp:ac-simps)

4.9.3 Deterministic Programs are Maximal

Any sub-additive refinement of a deterministic program is in fact an equivalence. Deterministic programs are thus maximal (under the refinement order) among sub-additive programs.

lemma refines-determ:
  fixes a::'s prog
  assumes da: determ (wp a)
4.9. THE ALGEBRA OF PGCL

and \( wa \): well-def \( a \)
and \( wb \): well-def \( b \)
and \( dr: a \sqsubseteq b \)
shows \( a \simeq b \)

Proof by contradiction.

**proof** (rule pequivI, rule contrapos-pp)
from \( wb \) have feasible (wp \( b \)) by(auto)
with \( wb \) have \( \lambda s. \text{sub-add}(wp b) \)
  by(auto dest: sublinear-subadd[OF well-def-wp-sublinear])
fix \( P::\text{s} \Rightarrow \text{real} \)
assume \( sP \) sound \( P \)

Assume that \( a \) and \( b \) are not equivalent:

**assume** \( ne: wp a P \neq wp b P \)

Find a point at which they differ. As \( a \sqsubseteq b \), \( wp b P \) must by strictly greater than \( wp a P \) 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 \( wp b P \ P \vdash wp a P \) by(auto)
moreover from \( sP \) dr have \( wp a P \ P \vdash wp b P \) by(auto)
ultimately show \( wp a P = wp b P \) by(auto)
qed
then obtain \( s \) where \( \text{less: wp a P \ s < wp b P \ s} \) by(blast)

Take a carefully constructed expectation:

let \( ?Pc = \lambda s. \text{bound-of } P - P \ s \)
have \( sPc \) sound \( ?Pc \)
**proof** (rule soundI)
from \( sP \) have \( \forall s. 0 \leq P \ s \) by(auto)
  hence \( \forall s. ?Pc \ s \leq \text{bound-of } P \) by(auto)
  thus bounded \( ?Pc \) by(blast)
from \( sP \) have \( \forall s. P \ s \leq \text{bound-of } P \) by(auto)
  hence \( \forall s. 0 \leq ?Pc \ s \) by(auto simp:sig-simps)
  thus nneg \( ?Pc \) by(auto)
qed

We then show that \( wp b \) violates feasibility, and thus healthiness.

from \( sP \) have \( 0 \leq \text{bound-of } P \) by(auto)
with \( da \) have \( \text{bound-of } P = wp a (\lambda s. \text{bound-of } P) \ s \)
  by(simp add:maximalD determ-maximalD)
also have \( ... = wp a (\lambda s. ?Pc \ s + P \ s) \) by(simp)
also from \( da \ \ sPc \) have \( ... = wp a ?Pc \ s + wp a P \ s \)
  by(subst additiveD[OF determ-additiveD], simp add: add:P sPc)
also from \( sPc \) dr have \( ... \leq wp b ?Pc \ s + wp a P \ s \)
  by(auto)
also from \texttt{less} have ... \( < \text{wp} \ b \ ?Pc \ s + \text{wp} \ b \ P \ s \)

by(\texttt{auto})

also from \texttt{sab} \( sPc \) have ... \( \leq \text{wp} \ b \ (\lambda s. \ ?Pc \ s + P \ s) \ s \)

by(\texttt{blast})

finally have \( \neg \text{wp} \ b \ (\lambda s. \ \text{bound-of} \ P) \ s \ \leq \text{bound-of} \ P \)

by(\texttt{simp})

thus \( \neg \text{bounded-by} \ (\text{bound-of} \ P) \ (\text{wp} \ b \ (\lambda s. \ \text{bound-of} \ P)) \)

by(\texttt{auto})

next

However,

\texttt{fix} \( P::'s \Rightarrow \text{real} \ assume \ sP: \text{sound} \ P \)

\texttt{hence} \( \neg \text{neq} \ (\lambda s. \ \text{bound-of} \ P) \ \text{by(auto)} \)

moreover have \( \text{bounded-by} \ (\text{bound-of} \ P) \ (\lambda s. \ \text{bound-of} \ P) \ \text{by(auto)} \)

ultimately

\texttt{show} \( \text{bounded-by} \ (\text{bound-of} \ P) \ (\text{wp} \ b \ (\lambda s. \ \text{bound-of} \ P)) \)

by(\texttt{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 \textit{Abort} is bottom, and \( a \sqcup b \) is \textit{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.

\texttt{quotient-type} \( 's \) \( \text{program} = \)

\( 's \ \text{prog} / \text{partial} : \lambda a \ b. \ a \simeq b \land \text{well-def} \ a \land \text{well-def} \ b \)

\texttt{proof}(\texttt{rule part-equivpI})

\texttt{have} \( \text{Skip} \simeq \text{Skip} \ \text{and} \ \text{well-def} \ \text{Skip} \ \text{by(auto intro:wd-intros}) \)

\texttt{thus} \( \exists x. \ x \simeq x \land \text{well-def} \ x \land \text{well-def} \ x \ \text{by(blast)} \)

\texttt{show} \( \text{symp} \ (\lambda a \ b. \ a \simeq b \land \text{well-def} \ a \land \text{well-def} \ b) \)

\texttt{proof}(\texttt{rule sympI, safe})

\texttt{fix} \( a::'a \ \text{prog} \ \text{and} \ b \)

\texttt{assume} \( a \simeq b \)

\texttt{hence} \( \text{equi-trans} \ (\text{wp} \ a) \ (\text{wp} \ b) \)

\texttt{by}(\texttt{simp add:pequiv-equiv-trans})

\texttt{thus} \( b \simeq a \ \text{by}(\texttt{simp add:ac-simps pequiv-equiv-trans}) \)

\texttt{qed}

\texttt{show} \( \text{transp} \ (\lambda a \ b. \ a \simeq b \land \text{well-def} \ a \land \text{well-def} \ b) \)

\texttt{by}(\texttt{rule transpI, safe, rule pequiv-trans})

\texttt{qed}

\texttt{instantiation} \( \text{program} :: \ (\text{type}) \ \text{semilattice-inf} \ \text{begin} \)

\texttt{lift-definition} \( \text{less-eq-program} :: 'a \ \text{program} \Rightarrow 'a \ \text{program} \Rightarrow \text{bool} \ \text{is} \ \text{refines} \)

\texttt{proof}(\texttt{safe})

\texttt{fix} \( a::'a \ \text{prog} \ \text{and} \ b \ c \ d \)


assume $a \simeq b$ hence $b \simeq a$ by $(\text{simp add: ac-simps})$
also assume $a \sqsubseteq c$
also assume $c \simeq d$
finally show $b \sqsubseteq d$.
next
  fix $a :: \text{'}a \text{ prog}$ and $b c d$
  assume $a \simeq b$
  also assume $b \sqsubseteq d$
  also assume $c \simeq d$ hence $d \simeq c$ by $(\text{simp add: ac-simps})$
  finally show $a \sqsubseteq c$.

lift-definition
  \[ \text{less-program} :: 'a \text{ program} \Rightarrow 'a \text{ program} \Rightarrow \text{ bool} \]
is $\lambda a b. \ a \sqsubseteq b \land \neg \ b \sqsubseteq a$
proof (safe)
  fix $a :: \text{'}a \text{ prog}$ and $b c d$
  assume $a \simeq b$ hence $b \simeq a$ by $(\text{simp add: ac-simps})$
  also assume $a \sqsubseteq c$
  also assume $c \simeq d$
  finally show $b \sqsubseteq d$.
next
  fix $a :: \text{'}a \text{ prog}$ and $b c d$
  assume $a \simeq b$
  also assume $b \sqsubseteq d$
  also assume $c \simeq d$ hence $d \simeq c$ by $(\text{simp add: ac-simps})$
  finally show $a \sqsubseteq c$.

next
  fix $a b$ and $c :: \text{'}a \text{ prog}$ and $d$
  assume $c \simeq d$
  also assume $d \sqsubseteq b$
  also assume $a \simeq b$ hence $b \simeq a$ by $(\text{simp add: ac-simps})$
  finally have $c \sqsubseteq a$.
  moreover assume $\neg c \sqsubseteq a$
  ultimately show $\text{False}$ by (auto)
next
  fix $a b$ and $c :: \text{'}a \text{ prog}$ and $d$
  assume $c \simeq d$ hence $d \simeq c$ by $(\text{simp add: ac-simps})$
  also assume $c \sqsubseteq a$
  also assume $a \simeq b$
  finally have $d \sqsubseteq b$.
  moreover assume $\neg d \sqsubseteq b$
  ultimately show $\text{False}$ by (auto)
qed

lift-definition
  \[ \text{inf-program} :: 'a \text{ program} \Rightarrow 'a \text{ program} \Rightarrow 'a \text{ program} \text{ is DC} \]
proof (safe)
  fix $a b c d :: \text{'}s \text{ prog}$
assume \( a \simeq b \) and \( c \simeq d \)
thus \( (a \sqcap c) \simeq (b \sqcap d) \) by (rule pequiv-DC)

next
fix \( a \) ::'s prog
assume well-def \( a \) well-def \( c \)
thus well-def \( (a \sqcap c) \) by (rule wd-intros)

next
fix \( a \) ::'s prog
assume well-def \( a \) well-def \( c \)
thus well-def \( (a \sqcap c) \) by (rule wd-intros)

qed

instance
proof
fix \( x \) \( y \) ::'a program
show \( (x < y) = (x \leq y \land \neg y \leq x) \)
by (transfer, simp)
show \( x \leq x \)
by (transfer, auto)
show \( \inf x y \leq x \)
by (transfer, rule left-refines-DC)
show \( \inf x y \leq y \)
by (transfer, rule right-refines-DC)
assume \( x \leq y \) and \( y \leq x \) thus \( x = y \)
by (transfer, iprover intro:pequiv-antisym)

next
fix \( x \) \( y \) \( z \) ::'a program
assume \( x \leq y \) and \( y \leq z \)
thus \( x \leq z \)
by (transfer, iprover intro:pr-trans)

next
fix \( x \) \( y \) \( z \) ::'a program
assume \( x \leq y \) and \( x \leq z \)
thus \( x \leq \inf y z \)
by (transfer, iprover intro:DC-refines)

qed

end

instantiation program :: (type) bot begin

lift-definition
bot-program :: 'a program is Abort
by (auto intro:wd-intros)

instance ..

end

lemma eq-det: \( \forall a \ b::'s \ prog. \ [ a \simeq b; \ determ (wp \ a) ] \imp determ (wp \ b) \)
proof (intro determI additiveI maximalI)
fix \( a \) \( b::'s \ prog \) and \( P::'s \imp real \)
4.9. THE ALGEBRA OF PGCL

and Q::'s ⇒ real and s::'s
assume da: determ (wp a)
assume sP: sound P and sQ: sound Q
and eq: a ≃ b
hence wp b (λs. P s + Q s) s =
  wp a (λs. 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 (λs. P s + Q s) s = wp b P s + wp b Q s.
next
fix a b::'s prog and c::real
assume a: determ (wp a)
assume a ≃ b hence b ≃ a by(simp add:ac-simps)
moreover assume nn: 0 ≤ c
ultimately have wp b (λ.. c) = wp a (λ.. c)
  by(simp add:pequivD const-sound)
also from da nn have ... = (λ.. c)
  by(simp add:determ-maximalD maximalD)
finally show wp b (λ.. c) = (λ.. c).
qed

lift-definition
  pdeterm :: 's program ⇒ bool
  is λa. determ (wp a)
proof(safe)
fix a b::'s prog
assume a ≃ b and determ (wp a)
thus determ (wp b) by(rule eq-det)
next
fix a b::'s prog
assume a ≃ b hence b ≃ 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 ≤ x ] ⇒ a = x
  by(transfer, auto intro:refines-determ)

4.9.5 Data Refinement

A projective data refinement construction for pGCL. By projective, we mean that the abstract state is always a function (φ) of the concrete state. Refinement may be predicated (G) on the state.
definition

drefines :: ('b ⇒ 'a) ⇒ ('b ⇒ bool) ⇒ 'a prog ⇒ 'b prog ⇒ bool

where

drefines ϕ G A B ≡ ∀ P Q. (unitary P ∧ unitary Q ∧ (P ⊢ wp A Q)) →
(«G» & (P o ϕ) ⊢ wp B (Q o ϕ))

lemma drefinesD[dest]:

[ drefines ϕ G A B; unitary P; unitary Q; P ⊢ wp A Q ] →
«G» & (P o ϕ) ⊢ wp B (Q o ϕ)

unfolding drefines-def by(blast)

We can alternatively use G as an assumption:

lemma drefinesD2:

assumes dr: drefines ϕ G A B

and uP: unitary P

and uQ: unitary Q

and wpA: P ⊢ wp A Q

and G: G s

shows (P o ϕ) s ≤ wp B (Q o ϕ) s

proof –

from uP have 0 ≤ (P o ϕ) s unfolding o-def by(blast)

with G have (P o ϕ) s = («G» & (P o ϕ)) s

by(simp add:exp-conj-def)

also from assms have ... ≤ wp B (Q o ϕ) s by(blast)

finally show (P o ϕ) s ≤ ...

qed

This additional form is sometimes useful:

lemma drefinesD3:

assumes dr: drefines ϕ G a b

and G: G s

and uQ: unitary Q

and wa: well-def a

shows wp a Q (ϕ s) ≤ wp b (Q o ϕ) s

proof –

let ?L s' = wp a Q s'

from uQ wa have sL: sound ?L by(blast)

from uQ wa have bL: bounded-by 1 ?L by(blast)

have ?L ⊢ ?L by(simp)

with sL and bL and assms

show ?thesis

by(blast intro:drefinesD2[OF dr, where P=?L, simplified])

qed

lemma drefinesI[intro]:

[ ∃ P Q. [ unitary P; unitary Q; P ⊢ wp A Q ] →
«G» & (P o ϕ) ⊢ wp B (Q o ϕ) ] →

drefines ϕ G A B
unfolding drefines-def by(blast)

Use G as an assumption, when showing refinement:

lemma drefinesI2:
fixes A::'a prog
and B::'b prog
and ϕ::'b ⇒ 'a
and G::'b ⇒ bool
assumes wB: well-def B
and withAs: ⋀ P Q s. unitary P; unitary Q;
G s; P ⊢ wp A Q ⟹ (P o ϕ) s ≤ wp B (Q o ϕ) s
shows drefines ϕ G A B
proof
fix P and Q
assume uP: unitary P
and uQ: unitary Q
and wpA: P ⊢ wp A Q

hence ⋀ s. G s ⟹ (P o ϕ) s ≤ wp B (Q o ϕ) s
using withAs by(blast)
moreover
from uQ have unitary (Q o ϕ)
unfolding o-def by(blast)
moreover
from uP have unitary (P o ϕ)
unfolding o-def by(blast)
ultimately
show «G» && (P o ϕ) ⊢ wp B (Q o ϕ)
using wB by(blast intro:entails-pconj-assumption)
qed

lemma dr-strengthen-guard:
fixes a::'s prog and b::'t prog
assumes fg: ⋀ s. F s ⟹ G s
and drab: drefines ϕ G a b
shows drefines ϕ F a b
proof(intro drefinesI)
fix P Q::'s expect
assume uP: unitary P and uQ: unitary Q
and wp: P ⊢ wp a Q
from fg have ⋀ s. «F» s ≤ «G» s by(simp add:embed-bool-def)
hence («F» && (P o ϕ)) ⊢ («G» && (P o ϕ)) by(auto intro:pconj-mono le-funI
simp:exp-conj-def)
also from drab uP uQ wp have ... ⊢ wp b (Q o ϕ) by(auto)
finally show «F» && (P o ϕ) ⊢ wp b (Q o ϕ)
qed

Probabilistic correspondence, pcorres, is equality on distribution transform-
ers, modulo a guard. It is the analogue, for data refinement, of program equivalence for program refinement.

**Definition**

\[
\text{pcorres} :: ('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow 'a \text{ prog} \Rightarrow 'b \text{ prog} \Rightarrow \text{bool}
\]

**where**

\[
\text{pcorres} \ \varphi \ G \ A \ B \longleftrightarrow (
\forall Q. \ \text{unitary} \ Q \longrightarrow \langle G \rangle &\& (wp \ A \ Q \ o \ \varphi) = \langle G \rangle &\& wp \ B \ (Q \ o \ \varphi)
\)

**Lemma** pcorres1:

\[
\langle Q. \ \text{unitary} \ Q \longrightarrow \langle G \rangle &\& (wp \ A \ Q \ o \ \varphi) = \langle G \rangle &\& wp \ B \ (Q \ o \ \varphi) \rangle \Longrightarrow
\]

\[
\text{pcorres} \ \varphi \ G \ A \ B
\]

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

Often easier to use, as it allows one to assume the precondition.

**Lemma** pcorres2[intro]:

**Fixes** A::'a prog and B::'b prog

**Assumes** withG: \[
\forall Q \ s. [ \ \text{unitary} \ Q; \ G \ s] \Longrightarrow wp \ A \ Q \ (\varphi \ s)= wp \ B \ (Q \ o \ \varphi) \ s
\]

**and** wA: well-def A

**and** wB: well-def B

**Shows** pcorres \ \varphi \ G \ A \ B

**Proof**(rule pcorresI, rule ext)

**Fix** Q::'a \Rightarrow \text{real} and s::'b

**Assume** uQ: \text{unitary} Q

**Hence** uQ:\ \varphi: \text{unitary} (Q o \ \varphi) **by**(auto)

**Show** (\langle G \rangle &\& (wp \ A \ Q \ o \ \varphi)) s = (\langle G \rangle &\& wp \ B \ (Q \ o \ \varphi)) s

**Proof**(cases G s)

**Case** True **note** this

**Moreover**

from well-def-wp-healthy[OF wA] uQ have \(0 \leq wp \ A \ Q \ (\varphi \ s)\) **by**(blast)

**Moreover**

from well-def-wp-healthy[OF wB] uQ\ \varphi \ have \(0 \leq wp \ B \ (Q \ o \ \varphi) \ s\) **by**(blast)

**Ultimately show** \?thesis

using uQ **by**(simp add:exp-conj-def withG)

**Next**

**Case** False **note** this

**Moreover**

from well-def-wp-healthy[OF wA] uQ have \(wp \ A \ Q \ (\varphi \ s) \leq 1\) **by**(blast)

**Moreover**

from well-def-wp-healthy[OF wB] uQ\ \varphi \ have \(wp \ B \ (Q \ o \ \varphi) \ s \leq 1\)

**by**(blast dest!:healthy-bounded-byD intro:sound-nneg)

**Ultimately show** \?thesis **by**(simp add:exp-conj-def)

**Qed**

**Qed**

**Lemma** pcorresD:

\[
\langle \ \text{pcorres} \ \varphi \ G \ A \ B; \ \text{unitary} \ Q \rangle \Longrightarrow \langle G \rangle &\& (wp \ A \ Q \ o \ \varphi) = \langle G \rangle &\& wp \ B \ (Q \ o \ \varphi)
\]

**Unfolding** pcorres-def **by**(simp)
4.9. **THE ALGEBRA OF PGCL**

Again, easier to use if the precondition is known to hold.

**Lemma pcorresD2**:  
assumes pc: pcorres \( \varphi \) \( G \) \( A \) \( B \)  
and uQ: unitary \( Q \)  
and wA: well-def \( A \) and wB: well-def \( B \)  
and G: \( G \) \( s \)  
shows wp \( A \) \( Q \) (\( \varphi \) \( s \)) = wp \( B \) (\( Q \circ \varphi \) \( s \))

**Proof**  
from \( uQ \) well-def-wp-healthy[OF wa] have \( \theta \leq wp A \) (\( \varphi \) \( s \)) by(auto)  
with G have wp \( A \) \( Q \) (\( \varphi \) \( s \)) = «\( G \)» \( s \) & wp \( A \) \( Q \) (\( \varphi \) \( s \)) by(simp)  
also {  
from pc uQ have «\( G \)» & & (wp \( A \) \( Q \circ \varphi \)) = «\( G \)» & & wp \( B \) (\( Q \circ \varphi \) ) by(rule pcorresD)  
hence «\( G \)» \( s \) & wp \( A \) \( Q \) (\( \varphi \) \( s \)) = «\( G \)» \( s \) & wp \( B \) (\( Q \circ \varphi \) \( s \))  
unfolding exp-conj-def o-def by(rule fun-cong)
}
also {  
from uQ have sound \( Q \) by(auto)  
hence sound (\( Q \circ \varphi \)) by(auto intro:sound-intros)  
with well-def-wp-healthy[OF wb] have \( \theta \leq wp B \) (\( Q \circ \varphi \) \( s \)) by(auto)  
with G have «\( G \)» \( s \) & wp \( B \) (\( Q \circ \varphi \) \( s \)) = wp \( B \) (\( Q \circ \varphi \) \( s \)) by(simp)  
}
finally show ?thesis .  
qed

4.9.6 **The Algebra of Data Refinement**

Program refinement implies a trivial data refinement:

**Lemma refines-drefines**:  
fixes \( a::'s \) prog  
assumes rab: \( a \subseteq b \) and wb: well-def \( b \)  
shows drefines (\( \lambda s. s \)) \( G \) \( a \) \( b \)  
**Proof** (intro drefinesI2 wb, simp add:o-def)  
fix \( P::'s \Rightarrow real \) and \( Q::'s \Rightarrow real \) and \( s::'s \)  
assume sQ: unitary \( Q \)  
assume \( P \vdash wp a \) \( Q \) hence \( P \) \( s \) \( \leq wp a \) \( Q \) \( s \) by(auto)  
also from rab sQ have ... \( \leq wp b \) \( Q \) \( s \) by(auto)  
finally show \( P \) \( s \) \( \leq wp b \) \( Q \) \( s \) .  
qed

Data refinement is transitive:

**Lemma dr-trans**:  
fixes \( A::'a \) prog and \( B::'b \) prog and \( C::'c \) prog  
assumes drAB: drefines \( \varphi \) \( G \) \( A \) \( B \)  
and drBC: drefines \( \varphi' \) \( G' \) \( B \) \( C \)  
and Gimp: \( \forall s. \ G' \ s \implies G \ (\varphi' \ s) \)  
shows drefines (\( \varphi \circ \varphi' \)) \( G' \) \( A \) \( C \)  
**Proof** (rule drefinesI)
fix $P::'a \Rightarrow \text{real}$ and $Q::'a \Rightarrow \text{real}$ and $s::'a$

assume $uP$: unitary $P$ and $uQ$: unitary $Q$

and $wpA$: $P \vdash wp A Q$

have $\langle G' \rangle \& \& \langle G \circ \varphi' \rangle = \langle G' \rangle$

proof (rule ext, unfold exp-conj-def)

fix $x$

show $\langle G' \rangle x \& \& \langle G \circ \varphi' \rangle x = \langle G' \rangle x$ (is $?X$)

proof (cases $G' x$)

case False

then show $?X$ by (simp)

next

moreover

with $Gimp$ have $(G \circ \varphi') x$ by (simp add: $o$-def)

ultimately

show $?X$ by (simp)

qed

with $uP$

have $\langle G' \rangle \& \& (P \circ (\varphi \circ \varphi')) = \langle G' \rangle \& \& ((\langle G \rangle \& \& (P \circ \varphi)) \circ \varphi')$

by (simp add: exp-conj-assoc $o$-assoc)

also {

from $uP$ $uQ$ $wpA$ and $drAB$

have $\langle G \rangle \& \& (P \circ \varphi) \vdash wp B (Q \circ \varphi)$

by (blast intro: drefinesD)

with $drBC$ and $uP$ $uQ$

have $\langle G' \rangle \& \& ((\langle G \rangle \& \& (P \circ \varphi)) \circ \varphi') \vdash wp C ((Q \circ \varphi) \circ \varphi')$

by (blast intro: unitary-intros drefinesD)

}

finally

show $\langle G' \rangle \& \& (P \circ (\varphi \circ \varphi')) \vdash wp C (Q \circ (\varphi \circ \varphi'))$

by (simp add: $o$-assoc)

qed

Data refinement composes with program refinement:

lemma pr-dr-trans[trans]:

assumes $prAB$: $A \sqsubseteq B$

and $drBC$: drefines $\varphi$ $G$ $B$ $C$

shows drefines $\varphi$ $G$ $A$ $C$

proof (rule drefinesI)

fix $P$ and $Q$

assume $uP$: unitary $P$

and $uQ$: unitary $Q$

and $wpA$: $P \vdash wp A Q$
4.9. THE ALGEBRA OF PGCL

note wpA
also from uQ and prAB have wp A Q \vdash wp B Q by (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
assume uP: unitary P
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 \implies 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 corres: pcorres \varphi G A C
and wC: well-def C
shows drefines \varphi G A C
proof
fix $P$ and $Q$
assume $uP$: unitary $P$ and $uQ$: unitary $Q$
and $wpA$: $P \vdash wp A Q$

from $wpA$ have $P \circ \varphi \vdash wp A Q \circ \varphi$ by (simp add: o-def le-fun-def)
hence $\langle G \rangle \& \langle P \circ \varphi \rangle \vdash \langle G \rangle \& \langle wp A Q \circ \varphi \rangle$
by (rule exp-conj-mono-right)
also from $corres uQ$
have $\ldots = \langle G \rangle \& \langle wp C (Q \circ \varphi) \rangle$ by (rule $pcorresD$)
also
have $\ldots \vdash wp C (Q \circ \varphi)$
proof (rule le-funI)
fix $s$
from $uQ$ have unitary $(Q \circ \varphi)$ by (rule unitary-intros)
with well-def-wp-healthy[OF $wC$] have nn-wpC: $0 \leq wp C (Q \circ \varphi) s$ by (blast)
show $(\langle G \rangle \& \langle wp C (Q \circ \varphi) \rangle) s \leq wp C (Q \circ \varphi) s$
proof (cases $G \ s$)
case True
with nn-wpC show $?thesis$ by (simp add: exp-conj-def)
next
case False note this
moreover {
from $uQ$ have unitary $(Q \circ \varphi)$ by (simp)
with well-def-wp-healthy[OF $wC$] have $wp C (Q \circ \varphi) s \leq 1$ by (auto)
}
moreover note nn-wpC
ultimately show $?thesis$ by (simp add: exp-conj-def)
qed
qed
finally show $\langle G \rangle \& \langle P \circ \varphi \rangle \vdash wp C (Q \circ \varphi)$.
qed

Any data refinement of a deterministic program is correspondence. This is the analogous result to that relating program refinement and equivalence.

lemma $drefines-determ$:
fixes $a$::$\mathcal{A}$ prog and $b$::$\mathcal{B}$ prog
assumes $da$: determ ($wp \ a$)
and $wa$: well-def $a$
and $wb$: well-def $b$
and $dr$: $drefines \varphi \ G \ a \ b$
shows $pcorres \varphi \ G \ a \ b$

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)
from $wb$ show feasible ($wp \ b$) by (auto)

note $ha$ = well-def-wp-healthy[OF $wa$]
note $hb$ = well-def-wp-healthy[OF $wb$]
4.9. THE ALGEBRA OF PGCL

From refinement, "previous result, the second must be somewhere strictly larger than the first:

If the programs do not correspond, the terms must differ somewhere, and given the

 Transformers themselves must differ at this point:

\[ \begin{align*}
\text{fix } Q 
\text{ a \Rightarrow real } \\
\text{assume } uQ : \text{unitary } Q \\
\text{hence } uQ \varphi : \text{unitary } (Q \circ \varphi) \text{ by(auto)} \\
\text{assume } ncl : \langle G \rangle \land \& \langle wp a Q \circ \varphi \rangle \neq \langle G \rangle \land \& \langle wp b Q \circ \varphi \rangle \\
\text{hence } ncl' : \langle wp a Q \circ \varphi \rangle \neq \langle wp b Q \circ \varphi \rangle \text{ by(auto)} \\
\end{align*} \]

From refinement, "G" \land \& \langle wp a Q \circ \varphi \rangle lies below "G" \land \& \langle wp b Q \circ \varphi \rangle.

\[ \begin{align*}
\text{from } ha uQ \\
\text{have gle : \langle G \rangle \land \& \langle wp a Q \circ \varphi \rangle \vdash wp b (Q \circ \varphi) \text{ by(blast intro:drefinesD[OF dr])} \\
\text{have le : \langle G \rangle \land \& \langle wp a Q \circ \varphi \rangle \dashv wp b (Q \circ \varphi) \text{ unfolding exp-conj-def} \\
\text{proof (rule le-funI)} \\
\text{fix } s \\
\text{from gle have \langle G \rangle s \land \langle wp a Q \circ \varphi \rangle s \leq wp b (Q \circ \varphi) s \\
\text{unfolding exp-conj-def by(auto)} \\
\text{hence \langle G \rangle s \land \langle wp a Q \circ \varphi \rangle s \leq \langle G \rangle s \land \langle wp b (Q \circ \varphi) s \\
\text{by(auto intro:pconj-mono)} \\
\text{moreover from uQ ha have \langle wp a Q \circ \varphi \rangle s \leq 1 } \text{ by(dest:healthy-bounded-byD)} \\
\text{moreover from uQ ha have } \theta \leq wp a (Q \circ \varphi) s \text{ by(auto) } \\
\text{ultimately } \\
\text{show \langle G \rangle s \land \langle wp a Q \circ \varphi \rangle s \leq \langle G \rangle s \land \langle wp b (Q \circ \varphi) s \\
\text{by(simp add:pconj-assoc) } \\
\text{qed} \\
\end{align*} \]

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:

\[ \begin{align*}
\text{have nle : } \exists s . (\langle G \rangle \land \& \langle wp a Q \circ \varphi \rangle) s < (\langle G \rangle \land \& \langle wp b (Q \circ \varphi) s \\
\text{proof (rule contrapos-np[OF ne], rule ext, rule antisym)} \\
\text{fix } s \\
\text{from le show } \langle G \rangle \land \& \langle wp a Q \circ \varphi \rangle) s \leq (\langle G \rangle \land \& \langle wp b (Q \circ \varphi) s \\
\text{by(blast) } \\
\text{next } \\
\text{fix } s \\
\text{assume } \neg (\exists s . (\langle G \rangle \land \& \langle wp a Q \circ \varphi \rangle) s < (\langle G \rangle \land \& \langle wp b (Q \circ \varphi) s \\
\text{thus } (\langle G \rangle \land \& \langle wp b (Q \circ \varphi) s \leq (\langle G \rangle \land \& \langle wp a (Q \circ \varphi) s \\
\text{by(simp add:not-less) } \\
\text{qed} \\
\text{from this obtain } s \text{ where less-s: } \\
\langle G \rangle \land \& \langle wp a Q \circ \varphi \rangle) s < (\langle G \rangle \land \& \langle wp b (Q \circ \varphi) s \\
\text{by(blast) } \\
\end{align*} \]

The transformers themselves must differ at this point:
hence larger: \( wp \ a \ Q \ (\varphi \ s) < wp \ b \ (Q \circ \varphi) \ s \)

proof (cases \( G \ s \))

- case True
  moreover from \( ha \ uQ \) have \( 0 \leq wp \ a \ Q \ (\varphi \ s) \)
    by(blast)
  moreover from \( hb \ uQ \varphi \) have \( 0 \leq wp \ b \ (Q \circ \varphi) \ s \)
    by(blast)
  moreover note less-s
  ultimately show \( ?thesis \) by(simp add:exp-conj-def)

- case False
  moreover from \( ha \ uQ \) have \( wp \ a \ Q \ (\varphi \ s) \leq 1 \)
    by(blast)
  moreover {
    from \( uQ \) have bounded-by 1 \( (Q \circ \varphi) \)
      by(blast)
    moreover from unitary-sound[OF \( uQ \)]
      have sound \( (Q \circ \varphi) \) by(auto)
    ultimately have \( wp \ b \ (Q \circ \varphi) \ s \leq 1 \)
      using \( hb \) by(auto)
  }
  moreover note less-s
  ultimately show \( ?thesis \) by(simp add:exp-conj-def)

qed

G must also hold, as otherwise both would be zero.

hence \( G-s: G \ s \)

proof (rule contrapos-np)

- assume \( nG: \neg G \ s \)
  moreover from \( ha \ uQ \) have \( wp \ a \ Q \ (\varphi \ s) \leq 1 \)
    by(blast)
  moreover {
    from \( uQ \) have bounded-by 1 \( (Q \circ \varphi) \)
      by(blast)
    moreover from unitary-sound[OF \( uQ \)]
      have sound \( (Q \circ \varphi) \) by(auto)
    ultimately have \( wp \ b \ (Q \circ \varphi) \ s \leq 1 \)
      using \( hb \) by(auto)
  }
  ultimately
  show \( (\langle G \rangle \& \& (wp \ a \ Q \circ \varphi)) \ s \neq (\langle G \rangle \& \& wp \ b \ (Q \circ \varphi)) \ s \)
    by(force)

qed

Take a carefully constructed expectation:

let \( \tilde{Q}c = \lambda s. \text{bound-of} \ Q \ s \)

have \( b\tilde{Q}c: \text{bounded-by} \ 1 \ \tilde{Q}c \)
4.9. THE ALGEBRA OF PGCL

proof (rule bounded-byI)
fix s
from uQ have bound-of Q ≤ 1 and 0 ≤ Q s by (auto)
thus bound-of Q − Q s ≤ 1 by (auto)
qed
have sQc: sound ?Qc
proof (rule soundI)
from bQc show bounded ?Qc by (auto)
show nneg ?Qc
proof (rule nnegI)
fix s
from uQ have bound-of Q ≤ 1 by (auto)
thus 0 ≤ bound-of Q − Q s by (auto)
qed
qed

By the maximality of wp a, wp b must violate feasibility, by mapping s to something strictly greater than bound-of Q.

from uQ have 0 ≤ bound-of Q by (auto)
with da have bound-of Q = wp a (λs. bound-of Q) (ϕ s)
by (simp add: maximalD determ-maximalD)
also have wp a (λs. bound-of Q) (ϕ s) = wp a (λs. Q s + ?Qc s) (ϕ s)
by (simp)
also {
from da have additive (wp a) by (blast)
with uQ sQc
have wp a (λs. Q s + ?Qc s) (ϕ s) =
wp a (Q s + wp a ?Qc (ϕ s)) by (subst additiveD, blast+)
}
also {
from ha and sQc and bQc
have «G» & & (wp a (?Qc o ϕ) △ wp b (?Qc o ϕ))
by (blast intro: drefinesD OF dr)
hence («G» & & (wp a ?Qc o ϕ)) s ≤ wp b (?Qc o ϕ) s
by (blast)
moreover from sQc and ha
have 0 ≤ wp a (λs. bound-of Q − Q s) (ϕ s)
by (blast)
ultimately
have wp a ?Qc (ϕ s) ≤ wp b (?Qc o ϕ) s
using G-s by (simp add: exp-conj-def)
hence wp a Q (ϕ s) + wp a ?Qc (ϕ s) ≤ wp a Q (ϕ s) + wp b (?Qc o ϕ) s
by (rule add-left-mono)
also with larger
have wp a Q (ϕ s) + wp b (?Qc o ϕ) s <
wbp (Q o ϕ) s + wp b (?Qc o ϕ) s
by (auto)
finally
have \( \wp a Q (\varphi s) + \wp a ?Qc (\varphi s) < \)

\( \wp b (Q o \varphi) s + \wp b (?Qc o \varphi) s . \)

}\n
also from \( sab \) and unitary-sound \([OF uQ]\) and \( sQc \)

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\( (\text{blast}) \)

also have \( \ldots = \wp b (\lambda s. \text{bound-of } Q) s \)

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

finally

drop \( \neg \text{feasible} (\wp b) \)

drop (rule contrapos-pn)

assume \( fb: \text{feasible} (\wp b) \)

have \( \text{bounded-by} (\text{bound-of } Q) (\lambda s. \text{bound-of } Q) \) by\( (\text{blast}) \)

hence \( \text{bounded-by} (\text{bound-of } Q) (\wp b (\lambda s. \text{bound-of } Q)) \)

using \( uQ \) by\( (\text{blast intro:feasible-boundedD}[OF fb]) \)

hence \( \wp b (\lambda s. \text{bound-of } Q) s \leq \text{bound-of } Q \) by\( (\text{blast}) \)

thus \( \neg \text{bound-of } Q < \wp b (\lambda s. \text{bound-of } Q) s \) by\( (\text{simp}) \)

qed

qed

4.9.7 Structural Rules for Correspondence

**Lemma** \( \text{pcorres-Skip} \):

\( \text{pcorres } \varphi G \text{ Skip Skip} \)

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

Correspondence composes over sequential composition.

**Lemma** \( \text{pcorres-Seq} \):

fixes \( A::'b \text{ prog} \) and \( B::'c \text{ prog} \)

and \( C::'b \text{ prog} \) and \( D::'c \text{ prog} \)

and \( \varphi::'c \Rightarrow 'b \)

assumes \( \text{pcAB: pcorres } \varphi G A B \)

and \( \text{pcCD: pcorres } \varphi H C D \)

and \( \text{wA: well-def A} \) and \( \text{wB: well-def B} \)

and \( \text{wC: well-def C} \) and \( \text{wD: well-def D} \)

and \( \text{p3p2: } Q. \text{ unitary } Q \Rightarrow \{\text{I}\} \& \& \wp B Q = \wp B (\{\text{H}\} \& \& Q) \)

and \( \text{p1p3: } \lambda s. G s \Rightarrow I s \)

shows \( \text{pcorres } \varphi G (A;C) (B;D) \)

proof (rule pcorresI)

fix \( Q::'b \Rightarrow \text{real} \)

assume \( uQ: \text{unitary } Q \)

with \( \text{well-def-wp-healthy}[OF wC] \) have \( uCQ: \text{unitary} (\wp C Q) \) by\( (\text{auto}) \)

from \( uQ \) \text{ well-def-wp-healthy}[OF wD] have \( uDQ: \text{unitary} (\wp D (Q o \varphi)) \)

by\( (\text{auto dest:unitary-comp}) \)

have \( \text{p3p1: } \lambda R S. [ \text{unitary R; unitary S; } \{\text{I}\} \& \& R = \{\text{I}\} \& \& S ] \Rightarrow \)

\( \{\text{G}\} \& \& R = \{\text{G}\} \& \& S \)

proof (rule ext)
4.9. THE ALGEBRA OF PGCL

fix \( R : \cdot c \Rightarrow \text{real} \) \& \& \( S : \cdot c \Rightarrow \text{real} \) \& \& \( s : \cdot c \)
assume \( a3 : \langle I \rangle \& \& R = \langle I \rangle \& \& S \)
and \( uR : \text{unitary} \ R \) \& \& \( uS : \text{unitary} \ S \)
show \( (\ast G \ast \& \& R) s = (\ast G \ast \& \& S) s \)
proof(simp add:exp-conj-def, cases \( G \) \& \& \( s \))
  case False
    note this
    moreover from \( uR \) have \( R s \leq 1 \) by(blast)
    moreover from \( uS \) have \( S s \leq 1 \) by(blast)
    ultimately show \( \langle G \rangle s \ast R s = \langle G \rangle s \ast S s \)
      by(simp)
  next
  case True
    note \( p1 = \) this
    with \( p1p3 \) have \( I s \) by(blast)
    with \( \text{fun-cong} \) \( \{ \langle a3 \rangle, \text{where} \ x = s \} \) have \( 1 \ast R s = 1 \ast S s \)
      by(simp add:exp-conj-def)
    with \( p1 \) show \( \langle G \rangle s \ast R s = \langle G \rangle s \ast S s \)
      by(simp)
  qed
qed

4.9.8 Structural Rules for Data Refinement

lemma dr-Skip:
  fixes \( \varphi : \cdot c \Rightarrow \cdot b \)
  shows drefines \( \varphi \ G \) \& \& \( \text{Skip} \) \& \& \( \text{Skip} \)
proof(intro drefinesI2 wd-intros)

```plaintext
4.9. THE ALGEBRA OF PGCL

fix \( R : \cdot c \Rightarrow \text{real} \) \& \& \( S : \cdot c \Rightarrow \text{real} \) \& \& \( s : \cdot c \)
assume \( a3 : \langle I \rangle \& \& R = \langle I \rangle \& \& S \)
and \( uR : \text{unitary} \ R \) \& \& \( uS : \text{unitary} \ S \)
show \( (\ast G \ast \& \& R) s = (\ast G \ast \& \& S) s \)
proof(simp add:exp-conj-def, cases \( G \) \& \& \( s \))
  case False
    note this
    moreover from \( uR \) have \( R s \leq 1 \) by(blast)
    moreover from \( uS \) have \( S s \leq 1 \) by(blast)
    ultimately show \( \langle G \rangle s \ast R s = \langle G \rangle s \ast S s \)
      by(simp)
  next
  case True
    note \( p1 = \) this
    with \( p1p3 \) have \( I s \) by(blast)
    with \( \text{fun-cong} \) \( \{ \langle a3 \rangle, \text{where} \ x = s \} \) have \( 1 \ast R s = 1 \ast S s \)
      by(simp add:exp-conj-def)
    with \( p1 \) show \( \langle G \rangle s \ast R s = \langle G \rangle s \ast S s \)
      by(simp)
  qed
qed

show \( \langle G \rangle \& \& \langle \text{wp} \ (A ; B) \rangle Q \circ \varphi \) = \( \langle G \rangle \& \& \langle \text{wp} \ D \rangle (Q \circ \varphi) \)
proof(simp add:wp-eval)
  from \( uCQ \) \( \text{pcAB} \) have \( \langle G \rangle \& \& \langle \text{wp} \ A \ \langle \text{wp} \ C Q \rangle \circ \varphi \) = \( \langle G \rangle \& \& \langle \text{wp} \ D \rangle \langle (\text{wp} \ C Q) \circ \varphi \rangle \)
    by(auto dest:\( \text{pcorresD} \)
  also have \( \langle G \rangle \& \& \langle \text{wp} \ D \rangle \langle Q \circ \varphi \rangle \) = \( \langle G \rangle \& \& \langle \text{wp} \ (Q \circ \varphi) \rangle \)
    proof(rule p3p1)
  from \( uCQ \) \( \text{well-def-wp-healthy} \) \( \{ \langle OF \ wB \rangle \) show \( \text{unitary} \ (\langle \text{wp} \ B \ \langle \text{wp} \ D \rangle (Q \circ \varphi) \rangle) \)
    by(auto intro:\( \text{unitary-comp} \))
  from \( uDQ \) \( \text{well-def-wp-healthy} \) \( \{ \langle OF \ wB \rangle \) show \( \text{unitary} \ (\langle \text{wp} \ B \ \langle \text{wp} \ D \rangle (Q \circ \varphi) \rangle) \)
    by(auto)
  from \( uQ \) have \( \langle H \rangle \& \& \langle \text{wp} \ C Q \circ \varphi \rangle = \langle H \rangle \& \& \langle \text{wp} \ D \rangle (Q \circ \varphi) \)
    by(blast intro:\( \text{pcorresD} \) \( \{ \langle \text{pcCD} \rangle \))
    thus \( \langle I \rangle \& \& \langle \text{wp} \ (\langle \text{wp} \ C Q \rangle \circ \varphi) \rangle = \langle I \rangle \& \& \langle \text{wp} \ B \ \langle \text{wp} \ D \rangle (Q \circ \varphi) \rangle \)
      by(simp add:p3p2 uCQ uDQ)
    qed
  finally show \( \langle G \rangle \& \& \langle \text{wp} \ A \ \langle \text{wp} \ C Q \rangle \circ \varphi \rangle = \langle G \rangle \& \& \langle \text{wp} \ B \ \langle \text{wp} \ D \rangle (Q \circ \varphi) \rangle \)
    .
  qed
qed

4.9.8 Structural Rules for Data Refinement

lemma dr-Skip:
  fixes \( \varphi : \cdot c \Rightarrow \cdot b \)
  shows drefines \( \varphi \ G \) \& \& \( \text{Skip} \) \& \& \( \text{Skip} \)
proof(intro drefinesI2 wd-intros)
```
fix $P::'b \Rightarrow \text{real}$ and $Q::'b \Rightarrow \text{real}$ and $s:.'c$

assume $P \vdash \text{wp } \text{Skip } Q$

def i

hence $(P \circ \varphi) s \leq \text{wp } \text{Skip } Q (\varphi s)$ by(simp, blast)

thus $(P \circ \varphi) s \leq \text{wp } \text{Skip } (Q \circ \varphi) s$ by(simp add:wp-eval)

qed

lemma dr-Abort:

fixes $\varphi::.'c \Rightarrow 'b$

def i

shows drefines $\varphi$ G Abort Abort

proof intro drefinesI2 wd-intros

fix $P::'b \Rightarrow \text{real}$ and $Q::'b \Rightarrow \text{real}$ and $s::.'c$

assume $P \vdash \vdash \text{wp } \text{Abort } Q$

def i

hence $(P \circ \varphi) s \leq \text{wp } \text{Abort } Q (\varphi s)$ by(auto)

thus $(P \circ \varphi) s \leq \text{wp } \text{Abort } (Q \circ \varphi) s$ by(simp add:wp-eval)

qed

lemma dr-Apply:

fixes $\varphi::.'c \Rightarrow 'b$

assumes commutes: $f \circ \varphi = \varphi \circ g$

shows drefines $\varphi$ G (Apply f) (Apply g)

proof intro drefinesI2 wd-intros

fix $P::'b \Rightarrow \text{real}$ and $Q::'b \Rightarrow \text{real}$ and $s::.'c$

assume wp: $P \vdash \text{wp } (\text{Apply } f) Q$

def i

hence $P \vdash (Q \circ f)$ by(simp add:wp-eval)

hence $P \circ (\varphi s) \leq (Q \circ f) (\varphi s)$ by(blast)

also have $... = Q ((f \circ \varphi) s)$ by(simp)

also with commutes

have $... = (((Q \circ \varphi) \circ g) s)$ by(simp)

also have $... = \text{wp } (\text{Apply } g) (Q \circ \varphi) s$

by(simp add:wp-eval)

finally show $(P \circ \varphi) s \leq \text{wp } (\text{Apply } g) (Q \circ \varphi) s$ by(simp)

qed

lemma dr-Seq:

assumes drAB: drefines $\varphi$ P A B

and drBC: drefines $\varphi$ Q C D

and wpB: «P» $\vdash$ 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

fix R and S

assume uR: unitary R and uS: unitary S

and wpAC: R $\vdash$ wp (A;;C) S

from uR

have «P» & & (R o \varphi) = «P» & & («P» & & (R o \varphi))
4.10. STRUCTURED REASONING

by (simp add: exp-conj-assoc)

also {
  from well-def-wp-healthy [OF wC] uR uS
  and wpAC [unfolded eval-wp-Seq o-def]
  have «P» & & (R o ϕ) ⊢ wp B {wp C S o ϕ}
    by (auto intro: drefinesD [OF drAB])
  with wpB well-def-wp-healthy [OF wC] uS
    sublinear-sub-conj [OF well-def-wp-sublinear, OF wB]
  have «P» & & («P» & & (R o ϕ)) ⊢ wp B («Q» & & (wp C S o ϕ))
    by (auto intro!: entails-combine dest!: unitary-sound)
}

also {
  from uS well-def-wp-healthy [OF wC]
  have «Q» & & (wp C S o ϕ) ⊢ wp D (S o ϕ)
    by (auto intro!: drefinesD [OF drBC])
  with well-def-wp-healthy [OF wB] well-def-wp-healthy [OF wC]
    well-def-wp-healthy [OF wD] and unitary-sound [OF uS]
  have wp B («Q» & & (wp C S o ϕ)) ⊢ wp B (wp D (S o ϕ))
    by (blast intro!: mono-transD)
}

finally
show «P» & & (R o ϕ) ⊢ ⊢ wp (B;;D) (S o ϕ)
  unfolding wp-eval o-def .
qed

lemma dr-repeat:
  fixes ϕ :: 'a ⇒ 'b
  assumes dr-ab: drefines ϕ G a b
      and Gpr: «G» ⊢ wp b «G»
      and wa: well-def a
      and wb: well-def b
  shows drefines ϕ G (repeat n a) (repeat n b) (is ?X n)
proof (induct n)
  show ?X 0 by (simp add: dr-Skip)

  fix n
  assume IH: ?X n
  thus ?X (Suc n) by (auto intro!: dr-Seq Gpr assms wd-intros)
qed

end

4.10 Structured Reasoning

theory StructuredReasoning imports Algebra begin
By linking the algebraic, the syntactic, and the semantic views of computation, we derive a set of rules for decomposing expectation entailment proofs, firstly over the syntactic structure of a program, and secondly over the refinement relation. These rules also form the basis for automated reasoning.

### 4.10.1 Syntactic Decomposition

**Lemma wp-Abort:**

\((\lambda s. 0) \vdash \text{wp Abort } Q\)

*unfolding wp-eval by(simp)*

**Lemma wlp-Abort:**

\((\lambda s. 1) \vdash \text{wlp Abort } Q\)

*unfolding wp-eval by(simp)*

**Lemma wp-Skip:**

\(P \vdash \text{wp Skip } P\)

*unfolding wp-eval by(blast)*

**Lemma wlp-Skip:**

\(P \vdash \text{wlp Skip } P\)

*unfolding wp-eval by(blast)*

**Lemma wp-Apply:**

\(Q \circ f \vdash \text{wp (Apply f) } Q\)

*unfolding wp-eval by(simp)*

**Lemma wlp-Apply:**

\(Q \circ f \vdash \text{wlp (Apply f) } Q\)

*unfolding wp-eval by(simp)*

**Lemma wp-Seq:**

assumes ent-a: \(P \vdash \text{wp a } Q\)

and ent-b: \(Q \vdash \text{wp b } R\)

and wa: well-def a

and wb: well-def b

and s-Q: sound Q

and s-R: sound R

shows \(P \vdash \text{wp (a ;; b) } R\)

**Proof**

- note ha = well-def-wp-healthy[OF wa]
- note hb = well-def-wp-healthy[OF wb]
- note ent-a
- also from ent-b ha hb s-Q s-R have wp a Q \(\vdash \text{wp a (wp b } R)\)
  by(blast intro:healthy-monoD2)
- finally show ?thesis by(simp add:wp-eval)

qed

**Lemma wlp-Seq:**
assumes $\text{ent-a}: P \vdash \text{wlp a Q}$
and $\text{ent-b}: Q \vdash \text{wlp b R}$
and $\text{wa}: \text{well-def a}$
and $\text{wb}: \text{well-def b}$
and $\text{u-Q}: \text{unitary Q}$
and $\text{u-R}: \text{unitary R}$
shows $P \vdash \text{wlp (a ; b) R}$

proof

- note $h_a = \text{well-def-wlp-nearly-healthy}[OF \text{wa}]$
- note $h_b = \text{well-def-wlp-nearly-healthy}[OF \text{wb}]$
- also from $\text{ent-b} \ h_a \ h_b \ \text{u-Q} \ \text{u-R}$ have $\text{wlp a Q} \vdash \text{wlp a (wlp b R)}$
  by (blast intro: nearly-healthy-monoD[OF $h_a$])

finally show $\text{thesis}$ by (simp add: wp-eval)

qed

lemma $\text{wp-PC}$:
$$(\lambda s. P s \cdot \text{wp a Q s} \cdot (1 - P s) \cdot \text{wp b Q s}) \vdash \text{wp (a (\lambda s. p) \oplus b) Q}$$
by (simp add: wp-eval)

lemma $\text{wlp-PC}$:
$$(\lambda s. P s \cdot \text{wp a Q s} \cdot (1 - P s) \cdot \text{wlp b Q s}) \vdash \text{wlp (a (\lambda s. p) \oplus b) Q}$$
by (simp add: wp-eval)

A simpler rule for when the probability does not depend on the state.

lemma $\text{PC-fixed}$:
assumes $\text{wp-a}: P \vdash a ab R$
and $\text{wp-b}: Q \vdash b ab R$
and $\text{np}: 0 \leq p$ and $\text{bp}: p \leq 1$
shows $$(\lambda s. p \cdot P s + (1 - P s) \cdot Q s) \vdash (a (\lambda s. p) \oplus b) ab R$$
unfolding $\text{PC-def}$

proof (rule le-funI)
fix $s$
from $\text{wp-a}$ and $\text{np}$ have $p \cdot P s \leq p \cdot a ab R s$
by (auto intro: mult-left-mono)
moreover {
from $\text{bp}$ have $0 \leq 1 - p$ by (simp)
with $\text{wp-b}$ have $$(1 - p) \cdot Q s \leq (1 - p) \cdot b ab R s$$
by (auto intro: mult-left-mono)
}
ultimately show $$p \cdot P s + (1 - p) \cdot Q s \leq$$
$$p \cdot a ab R s + (1 - p) \cdot b ab R s$$
by (rule add-mono)
qed

lemma $\text{wp-PC-fixed}$:
$$[ P \vdash \text{wp a R}; Q \vdash \text{wp b R}; 0 \leq p; p \leq 1 ] \implies$$
$$(\lambda s. p \cdot P s + (1 - p) \cdot Q s) \vdash \text{wp (a (\lambda s. p) \oplus b) R}$$
by (simp add: wp-def PC-fixed)
lemma \textit{wlp-PC-fixed}:\[
\begin{align*}
& [ \; P \vdash \text{wlp } a \; R; \; Q \vdash \text{wlp } b \; R; \; 0 \leq p; \; p \leq 1 \; ] \implies \\
& (\lambda s.\ p \ast P \ s + (1 - p) \ast Q \ s) \vdash \text{wlp } (a \ (\lambda s.\ p) \oplus b) \; R \\
& \text{by:\ (simp add:}\ wlp\text{-def }\text{PC-fixed)}
\end{align*}
\]

lemma \textit{wp-DC}:\[
(\lambda s.\ \text{min } (wp \ a \ Q \ s)) \vdash \text{wp } (a \ [b]) \; Q \\
\text{unfolding } \text{wp-eval by:}\ (\text{simp})
\]

lemma \textit{wlp-DC}:\[
(\lambda s.\ \text{min } (wlp \ a \ Q \ s)) \vdash \text{wlp } (a \ [b]) \; Q \\
\text{unfolding } \text{wp-eval by:}\ (\text{simp})
\]

Combining annotations for both branches:

lemma \textit{DC-split}:\[
\begin{align*}
& \text{fixes } a::'s \text{ prog and } b \\
& \text{assumes } wp\text{a: } P \vdash a \ ab \ R \\
& \quad \text{and } wp\text{b: } Q \vdash b \ ab \ R \\
& \text{shows } (\lambda s.\ \text{min } (P \ s) \ (Q \ s)) \vdash (a \ [b]) \ ab \ R \\
& \text{unfolding } \text{DC-def}
\end{align*}
\]

proof (rule \text{le-funI})\[
\begin{align*}
& \text{fix } s \\
& \text{from wp\text{a wpb}} \\
& \text{have } P \ s \leq a \ ab \ R \ s \text{ and } Q \ s \leq b \ ab \ R \ s \text{ by (auto)} \\
& \text{thus } \text{min } (P \ s) \ (Q \ s) \leq \text{min } (a \ ab \ R \ s) \ (b \ ab \ R \ s) \text{ by (auto)} \\
& \text{qed}
\end{align*}
\]

lemma \textit{wp-DC-split}:\[
\begin{align*}
& [ \; P \vdash \text{wp } \text{prog } \ R; \; Q \vdash \text{wp } \text{prog}' \ R \; ] \implies \\
& (\lambda s.\ \text{min } (P \ s) \ (Q \ s)) \vdash \text{wp } (\text{prog } [\text{prog}') \ R \\
& \text{by:}(\text{simp add:}\ \text{wp-def }\text{DC-split)}
\end{align*}
\]

lemma \textit{wlp-DC-split}:\[
\begin{align*}
& [ \; P \vdash \text{wlp } \text{prog } \ R; \; Q \vdash \text{wlp } \text{prog}' \ R \; ] \implies \\
& (\lambda s.\ \text{min } (P \ s) \ (Q \ s)) \vdash \text{wlp } (\text{prog } [\text{prog}') \ R \\
& \text{by:}(\text{simp add:}\ \text{wlp-def }\text{DC-split)}
\end{align*}
\]

lemma \textit{wp-DC-split-same}:\[
[ \; P \vdash \text{wp } \text{prog } Q; \; P \vdash \text{wp } \text{prog}' \ Q \; ] \implies P \vdash \text{wp } (\text{prog } [\text{prog}') \ Q \\
\text{unfolding } \text{wp-eval by:}\ (\text{blast intro:}\text{min.boundedI)}
\]

lemma \textit{wlp-DC-split-same}:\[
[ \; P \vdash \text{wlp } \text{prog } Q; \; P \vdash \text{wlp } \text{prog}' \ Q \; ] \implies P \vdash \text{wlp } (\text{prog } [\text{prog}') \ Q \\
\text{unfolding } \text{wp-eval by:}\ (\text{blast intro:}\text{min.boundedI)}
\]

lemma \textit{SetPC-split}:\[
\begin{align*}
& \text{fixes } f::'x \Rightarrow 'y \text{ prog} \\
& \quad \text{and } p::'y \Rightarrow 'x \Rightarrow \text{real}
\end{align*}
\]
4.10. STRUCTURED REASONING

proof (rule le-funI)

fix s

from rec have \( \land x. x \in \text{supp} (p s) \Rightarrow P x \vdash f x ab Q \)

moreover from nnp have \( \land x. \theta \leq p s x \) by (blast)

ultimately have \( \land x. x \in \text{supp} (p s) \Rightarrow p s x \ast P x s \leq p s x \ast f x ab Q s \)

by (blast intro: mult-left-mono)

thus \( (\sum x \in \text{supp} (p s), p s x \ast P x s) \leq (\sum x \in \text{supp} (p s), p s x \ast f x ab Q s) \)

by (rule setsum-mono)

qed

lemma wp-SetPC-split:

\[
[ \land x. x \in \text{supp} (p s) \Rightarrow P x \vdash wp (f x) Q; \land s. \text{nneq} (p s) ] \Rightarrow
(\land s. \sum x \in \text{supp} (p s), p s x \ast P x s) \vdash wp (\text{SetPC} f p) Q
\]

by (simp add: wp-def SetPC-split)

lemma wlp-SetPC-split:

\[
[ \land x. x \in \text{supp} (p s) \Rightarrow P x \vdash wlp (f x) Q; \land s. \text{nneq} (p s) ] \Rightarrow
(\land s. \sum x \in \text{supp} (p s), p s x \ast P x s) \vdash wlp (\text{SetPC} f p) Q
\]

by (simp add: wp-def SetPC-split)

lemma wp-SetDC-split:

\[
[ \land s. x. x \in S s \Rightarrow P \vdash wp (f x) Q; \land s. S s \neq {} ] \Rightarrow
P \vdash wp (\text{SetDC} f S) Q
\]

by (rule le-funI, unfold wp-eval, blast intro!: cInf-greatest)

lemma wlp-SetDC-split:

\[
[ \land s. x. x \in S s \Rightarrow P \vdash wlp (f x) Q; \land s. S s \neq {} ] \Rightarrow
P \vdash wlp (\text{SetDC} f S) Q
\]

by (rule le-funI, unfold wp-eval, blast intro!: cInf-greatest)

lemma wp-SetDC:

assumes wp: \( \land s. x. x \in S s \Rightarrow P x \vdash wp (f x) Q \)

and ne: \( \land s. S s \neq {} \)

and sP: \( \land x. \text{sound} (P x) \)

shows \( (\land s. \text{Inf} ((\land x. P x s) \cdot S s)) \vdash wp (\text{SetDC} f S) Q \)

using assms by (intro le-funI, simp add: wp-eval del: Inf-image-eq, blast intro!: cInf-monono)

lemma wlp-SetDC:

assumes wp: \( \land s. x. x \in S s \Rightarrow P x \vdash wlp (f x) Q \)

and ne: \( \land s. S s \neq {} \)

and sP: \( \land x. \text{sound} (P x) \)

shows \( (\land s. \text{Inf} ((\land x. P x s) \cdot S s)) \vdash wlp (\text{SetDC} f S) Q \)

using assms by (intro le-funI, simp add: wp-eval del: Inf-image-eq, blast intro!: cInf-monono)

lemma wp-Embed:
\[ P \vdash t Q \Rightarrow P \vdash wp (\text{Embed} t) Q \]
by\(\text{simp add:wp-def Embed-def}\)

**Lemma wlp-Embed:**
\[ P \vdash t Q \Rightarrow P \vdash wlp (\text{Embed} t) Q \]
by\(\text{simp add:wlp-def Embed-def}\)

**Lemma wp-Bind:**
\[ [ \forall s. P s \leq wp (a (f s)) Q s ] \Rightarrow P \vdash wp (\text{Bind} f a) Q \]
by\(\text{auto simp:wp-def Bind-def}\)

**Lemma wlp-Bind:**
\[ [ \forall s. P s \leq wlp (a (f s)) Q s ] \Rightarrow P \vdash wlp (\text{Bind} f a) Q \]
by\(\text{auto simp:wlp-def Bind-def}\)

**Lemma wp-repeat:**
\[ [ P \vdash wp a Q; Q \vdash wp (\text{repeat} n a) R; \text{well-def} a; \text{sound} Q; \text{sound} R ] \Rightarrow P \vdash wp (\text{repeat} (\text{Suc} n) a) R \]
by\(\text{auto intro:wp-Seq wd-intros}\)

**Lemma wlp-repeat:**
\[ [ P \vdash wlp a Q; Q \vdash wlp (\text{repeat} n a) R; \text{well-def} a; \text{unitary} Q; \text{unitary} R ] \Rightarrow P \vdash wlp (\text{repeat} (\text{Suc} n) a) R \]
by\(\text{auto intro:wlp-Seq wd-intros}\)

Note that the loop rules presented in section Section 4.8 are of the same form, and would belong here, had they not already been stated.

The following rules are specialisations of those for general transformers, and are easier for the unifier to match.

**Lemmas wp-strengthen-post=**
\[ \text{entails-strengthen-post}[\text{where} t=wp a \text{ for } a] \]

**Lemma wp-strengthen-post:**
\[ P \vdash wlp a Q \Rightarrow \text{nearby-healthy} (wp a) \Rightarrow \text{unitary} R \Rightarrow Q \vdash R \Rightarrow \text{unitary} Q \]
\[ P \vdash wlp a R \]
by\(\text{blast intro:entails-trans}\)

**Lemmas wp-weaken-pre=**
\[ \text{entails-weaken-pre}[\text{where} t=wp a \text{ for } a] \]

**Lemmas wlp-weaken-pre=**
\[ \text{entails-weaken-pre}[\text{where} t=wlp a \text{ for } a] \]

**Lemmas wp-scale=**
\[ \text{entails-scale}[\text{where} t=wp a \text{ for } a, \text{OF - well-def-wp-healthy}] \]
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:**

\[
\begin{align*}
[a \sqsubseteq b; P \vdash wp a Q] & \implies P \vdash wp b Q \\
\text{by/auto intro; entails-trans}
\end{align*}
\]

**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 (\{a\} - \{b\}p)

**Where**

wp-valid $P$ prog $Q$ $\equiv$ $P \vdash wp Q$

**Lemma** wp-validI:

$P \vdash wp Q$ $\implies$ $\{P\}$ prog $\{Q\}$p

**Unfolding** wp-valid-def $\text{by}(assumption)$

**Lemma** wp-validD:

$\{P\}$ prog $\{Q\}$p $\implies$ $P \vdash wp Q$

**Unfolding** wp-valid-def $\text{by}(assumption)$

**Lemma** valid-Seq:

\[
\begin{align*}
[\{P\} a \{Q\}p; \{Q\} b \{R\}p; \text{well-def } a; \text{well-def } b; \text{sound } Q; \text{sound } R ] & \implies \\
\{P\} a :: b \{R\}p
\end{align*}
\]

**Unfolding** wp-valid-def $\text{by}(rule \ wp-Seq)$

We make it available to the computational reasoner:

**Declare** valid-Seq[trans]

end

4.11 Loop Termination

**Theory** Termination imports Embedding StructuredReasoning Loops begin

Termination for loops can be shown by classical means (using a variant, or a measure function), or by probabilistic means: We only need that the loop terminates with probability one.
4.11.1 Trivial Termination

A maximal transformer (program) doesn’t affect termination. This is essentially saying that such a program doesn’t abort (or diverge).

**lemma** maximal-Seq-term:

- **fixes** \( r::'s \text{ prog} \) and \( s::'s \text{ prog} \)
- **assumes** \( mr: \text{ maximal} (\text{ wp } r) \)
  and \( ws: \text{ well-def } s \)
- **shows** \( (\lambda s. 1) \vdash \text{ wp } (r ;; s) (\lambda s. 1) \)

**proof**

- **note** \( hs = \text{ well-def-wp-healthy}[\text{OF } ws] \)
- **have** \( \text{ wp } s (\lambda s. 1) = (\lambda s. 1) \)
- **proof**(rule antisym)
  - **show** \( (\lambda s. 1) \vdash \text{ wp } s (\lambda s. 1) \) by(rule ts)
  - **have** bounded-by \( I \) \( (\text{ wp } s (\lambda s. 1)) \)
    by(auto intro!:healthy-bordered-byD[OF hs])
  - **thus** \( \text{ wp } s (\lambda s. 1) \\
  (\lambda s. 1) \) by(auto intro!:le-funI)
  - **qed**
- **with** \( mr \text{ show } ?\text{thesis} \)
  - by(simp add:wp-eval embed-boold-def maximalD)
  - **qed**

From any state where the guard does not hold, a loop terminates in a single step.

**lemma** term-onestep:

- **assumes** \( wb: \text{ well-def } \text{ body} \)
- **shows** \( «N G» \vdash \text{ wp } \text{ do } G \longrightarrow \text{ od } (\lambda s. 1) \)

**proof**(rule le-funI)

- **note** \( hb = \text{ well-def-wp-healthy}[\text{OF } wb] \)
- **fix** \( s \)
- **show** \( «N G» s \leq \text{ wp } \text{ do } G \longrightarrow \text{ od } (\lambda s. 1) s \)
- **proof**(cases \( G \) \( s \), simp-all add:wp-loop-nguard \( hb \))
  - **from** \( hb \text{ have } \text{ sound } \text{ wp } \text{ do } G \longrightarrow \text{ od } (\lambda s. 1) \)
    by(auto intro!:healthy-boundD[OF healthy-wp-loop])
  - **thus** \( 0 \leq \text{ wp } \text{ do } G \longrightarrow \text{ od } (\lambda s. 1) s \) by(auto)
  - **qed**
  - **qed**

4.11.2 Classical Termination

The first non-trivial termination result is quite standard: If we can provide a natural-number-valued measure, that decreases on every iteration, and implies termination on reaching zero, the loop terminates.

**lemma** loop-term-nat-measure-noinv:

- **fixes** \( m :: 's \Rightarrow \text{ nat} \) and \( \text{ body :: 's \text{ prog} } \)
- **assumes** \( wb: \text{ well-def } \text{ body} \)
  and \( \text{ guard: } \forall s. m s = 0 \longrightarrow G s \)
and variant: \( \forall n. \langle \lambda s. m \cdot s = \text{Suc} \cdot n \rangle \vdash wp \; \langle \lambda s. m \cdot s = n \rangle \)

shows \( \lambda s. \cdot 1 \vdash wp \; \langle \lambda s. m \cdot s = n \rangle \)

**proof**

- **note** \( \text{hb} = \text{well-def-wp-healthy}[OF \; \text{wb}] \)

- **have** \( \forall n. (\forall s. m \cdot s = n \rightarrow 1 \leq wp \; \langle \lambda s. \cdot 1 \rangle \) \)

**proof**(\( \text{induct-tac} \; n \))

**fix** \( n \)

**show** \( \forall s. m \cdot s = 0 \rightarrow 1 \leq wp \; \langle \lambda s. \cdot 1 \rangle \) \)

**proof**(\( \text{clarify} \))

**fix** \( s \)

**assume** \( m \cdot s = 0 \)

**with** \( \text{guard} \) **have** \( \neg \; G \; s \) **by**(\( \text{blast} \))

**with** \( \text{hb} \) **show** \( 1 \leq wp \; \langle \lambda s. \cdot 1 \rangle \) \)

**by**(\( \text{simpl add:wp-loop-nguard} \))

**qed**

**assume** \( IH: \forall s. m \cdot s = n \rightarrow 1 \leq wp \; \langle \lambda s. \cdot 1 \rangle \) \)

**hence** \( IH': \forall s. m \cdot s = n \rightarrow 1 \leq wp \; \langle \lambda s. \text{True} \rangle \) \)

**by**(\( \text{simpl add:embed-bool-def} \))

**have** \( \forall s. m \cdot s = \text{Suc} \cdot n \rightarrow 1 \leq wp \; \langle \lambda s. \text{True} \rangle \) \)

**proof**(\( \text{intro fold-premise healthy-intros hb, rule le-funI} \))

**fix** \( s \)

**show** \( \langle \lambda s. m \cdot s = \text{Suc} \cdot n \rangle \) \)

**proof**(\( \text{cases} \; G \; s \))

**case** \( \text{False} \)

**hence** \( 1 = \langle \lambda s. \text{Suc} \rangle \) **by**(\( \text{auto} \))

**also from** \( \text{wb} \) **have** \( \ldots \leq wp \; \langle \lambda s. \cdot 1 \rangle \) \)

**by**(\( \text{rule le-funD}[OF \; \text{term-onestep}] \))

**finally show** \( \langle \lambda s. \cdot 1 \rangle \) **by**(\( \text{simpl add:embed-bool-def} \))

**next**

**case** \( \text{True} \) **note** \( G = \text{this} \)

**from** \( IH' \) **have** \( \langle \lambda s. m \cdot s = n \rangle \) \)

**by**(\( \text{blast intro:use-premise healthy-intros hb} \))

**with** \( \text{variant wb} \)

**have** \( \langle \lambda s. m \cdot s = \text{Suc} \cdot n \rangle \) \)

**by**(\( \text{blast intro:wp-Seq ad-intros} \))

**hence** \( \langle \lambda s. m \cdot s = \text{Suc} \cdot n \rangle \) \)

**by**(\( \text{auto} \))

**also from** \( \text{hb} \) **have** \( \ldots = wp \; \langle \lambda s. \text{True} \rangle \) \)

**by**(\( \text{simpl add:wp-loop-guard} \))

**finally show** \( \langle \lambda s. \text{True} \rangle \) **by**(\( \text{auto} \))

**qed**

**thus** \( \forall s. m \cdot s = \text{Suc} \; n \rightarrow 1 \leq wp \; \langle \lambda s. \cdot 1 \rangle \) \)

**by**(\( \text{simpl add:embed-bool-def} \))

**qed**

**thus** \( \langle \lambda s. \cdot 1 \rangle \) **by**(\( \text{auto} \))

**qed**

This version allows progress to depend on an invariant. Termination is then
determined by the invariant’s value in the initial state.

**lemma** loop-term-nat-measure:
- **fixes** \( m :: 's \rightarrow \text{nat} \) and \( \text{body} :: 's \rightarrow \text{prog} \)
- **assumes** \( \text{wb} :: \text{well-def body} \)
- and **guard**: \( \forall s. \; m \; s = 0 \rightarrow \neg \; G \; s \)
- and **variant**: \( \forall n. \; \lambda s. \; m \; s = \text{Suc} \; n \) \&\& \( \langle I \rangle \) \&\& \( \text{wp body} \; \langle \lambda s. \; m \; s = n \rangle \)
- and **inv**: \( \text{wp-inv G body} \; \langle I \rangle \)
- **shows** \( \langle I \rangle \; \vdash \; \text{wp do} \; G \rightarrow \text{body od} \; (\langle \lambda s. \; I \rangle) \)

**proof**
- **note** \( \text{hb} = \text{well-def-wp-healthy}[\text{OF} \; \text{wb}] \)
- **note** \( \text{scb} = \text{sublinear-sub-conj}[\text{OF} \; \text{well-def-wp-sublinear}, \; \text{OF} \; \text{wb}] \)
- **have** \( \langle I \rangle \; \vdash \; \text{wp do} \; G \rightarrow \text{body od} \; \langle \lambda s. \; \text{True} \rangle \)

**proof**(rule use-premise, intro healthy-intros \( \text{hb} \))
- **fix** \( s \)
- **have** \( \forall n. \; (\forall s. \; m \; s = n \land I \; s \rightarrow I \leq \; \text{wp do} \; G \rightarrow \text{body od} \; \langle \lambda s. \; \text{True} \rangle \; s) \)

**proof**(induct-tac \( n \))
- **fix** \( n \)
- **show** \( \forall s. \; m \; s = 0 \land I \; s \rightarrow I \leq \; \text{wp do} \; G \rightarrow \text{body od} \; \langle \lambda s. \; \text{True} \rangle \; s \)

**proof**(clarify)
- **fix** \( s \)
- **assume** \( m \; s = 0 \)
- with **guard** **have** \( \neg \; G \; s \) by(blast)
- with **hb** **show** \( I \leq \; \text{wp do} \; G \rightarrow \text{body od} \; \langle \lambda s. \; \text{True} \rangle \; s \)
  - by(simp add:wp-loop-nguard)

**qed**
- **assume** \( \forall s. \; m \; s = n \land I \; s \rightarrow I \leq \; \text{wp do} \; G \rightarrow \text{body od} \; \langle \lambda s. \; \text{True} \rangle \; s \)
- **show** \( \forall s. \; m \; s = \text{Suc} \; n \land I \; s \rightarrow I \leq \; \text{wp do} \; G \rightarrow \text{body od} \; \langle \lambda s. \; \text{True} \rangle \; s \)

**proof**(intro fold-premise healthy-intros \( \text{hb} \) le-fun1)
- **fix** \( s \)
- **show** \( \langle \lambda s. \; m \; s = \text{Suc} \; n \land I \; s \rangle \; s \leq \; \text{wp do} \; G \rightarrow \text{body od} \; \langle \lambda s. \; \text{True} \rangle \; s \)

**proof**(cases \( G \; s \))
- **case** False with **hb** **show** ?thesis
  - by(simp add:wp-loop-nguard)

**next**
- **case** True **note** \( G \; = \; \text{this} \)
- **have** \( \langle \lambda s. \; m \; s = \text{Suc} \; n \rangle \) \&\& \( \langle I \rangle \) \&\& \( G \) = \( \langle \lambda s. \; m \; s = \text{Suc} \; n \rangle \) \&\& \( \langle I \rangle \) \&\& \( G \) \&\& \( G \)
  - by(simp)
- also **have** \( \ldots = \langle \lambda s. \; m \; s = \text{Suc} \; n \rangle \) \&\& \( \langle I \rangle \) \&\& \( G \)
  - by(simp add:exp-conj-assoc exp-conj-unitary del:exp-conj-ideq)
- also **have** \( \ldots = \langle \lambda s. \; m \; s = \text{Suc} \; n \rangle \) \&\& \( \langle I \rangle \) \&\& \( G \) \&\& \( I \)
  - by(simp only:exp-conj-comm)
- also \{ from inv \( \text{hb} \) **have** \( G \) \&\& \( \langle I \rangle \) \&\& \( \text{wp body} \; \langle I \rangle \)
    - by(rule wp-inv-stdD)
  with **variant**
- **have** \( \langle \lambda s. \; m \; s = \text{Suc} \; n \rangle \) \&\& \( \langle I \rangle \) \&\& \( G \) \&\& \( I \)
  - \&\& \( \text{wp body} \; \langle \lambda s. \; m \; s = n \rangle \) \&\& \( \text{wp body} \; \langle I \rangle \)
  - by(rule entails-frame)
4.11. LOOP TERMINATION

} also from scb
have \( wp \ body \ (\lambda s. m s = n) \& \& wp \ body \ (\lambda s. m s = n) \& \& I \) ⊢
\( wp \ body \ (\lambda s. m s = n) \& \& I \)
by (blast)
finally have \( \lambda s. m s = Suc n \) \& \& \( \lambda s. True \) ⊢
\( wp \ body \ (\lambda s. m s = n) \& \& I \)
moreover {
from IH have \( \lambda s. m s = n \& \& I \) ⊢ \( wp \ do \ G \rightarrow body \ od \ \lambda s. True \)
by (blast intro: use-premise healthy-intros hb)
hence \( \lambda s. m s = n \& \& I \) ⊢ \( wp \ do \ G \rightarrow body \ od \ \lambda s. True \)
by (simp add: exp-conj-std-split)
}
ultimately have \( \lambda s. m s = Suc n \) \& \& \( \lambda s. True \) ⊢
\( wp \ do \ G \rightarrow body \ od \ \lambda s. True \)
by (auto simp add: embed-bool-def)
also from hb G have \( \ldots \) ⊢ \( wp \ do \ G \rightarrow body \ od \ \lambda s. True \)
by (simp add: wp-loop-guard)
finally show \( \vdash \)
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: \( \lambda s. p \vdash wp \ body \ N \ G \)  
and nzp: \( 0 < p \)  
  -- The body is maximal i.e. it terminates absolutely.  
and mb: maximal (wp body)  
shows \( \lambda s. 1 \vdash wp \ do \ G \rightarrow body \ od \ (\lambda s. 1) \)
proof –

note \( hh = \text{well-def-wp-healthy} \) [OF \( wb \)]

note \( sh = \text{healthy-scalingD} \) [OF \( hh \)]

note \( sab = \text{sublinear-subadd} \) [OF \( \text{well-def-wp-sublinear} \), \( \text{OF wb} \), \( \text{OF healthy-feasibleD} \), \( OF hh \)]

from \( hh \) have \( hloop: \text{healthy} (wp \ G \rightarrow \text{body od}) \)
  by (rule healthy-intros)

hence \( swp: \text{sound} (wp \ G \rightarrow \text{body od} (\lambda s. 1)) \) by (blast)

\( p \) is no greater than 1, by feasibility.

from onestep have \( \forall s. p \leq wp \ N \ G \ s \) by (auto)
also {
  from \( hh \) have unitary (wp \ N \ G \ s) by (auto)
  hence \( \forall s. wp \ N \ G \ s \leq 1 \) by (auto)
}
finally have \( p1: p \leq 1 \).

This is the crux of the proof: that given a lower bound below 1, we can find another, higher one.

have new-bound: \( \forall k. 0 \leq k \Rightarrow k \leq 1 \Rightarrow (\lambda s. k) \vdash wp \ G \rightarrow \text{body od} (\lambda s. 1) \)
proof (rule le-funI)
  fix \( k \) \( s \)
  assume \( X: \lambda s. k \vdash wp \ G \rightarrow \text{body od} (\lambda s. 1) \)
    and \( k0: 0 \leq k \) and \( k1: k \leq 1 \)
from k1 have nz1k: \( 0 \leq 1 - k \) by (auto)
with p1 have \( p \ast (1 - k) + k \leq 1 \ast (1 - k) + k \)
  by (blast intro: mult-right-mono add-mono)
  hence \( p \ast (1 - k) + k \leq 1 \)
  by (simp)

The new bound is \( p \ast (1 - k) + k \).

hence \( p \ast (1 - k) + k \leq \N G \ s + \G s \ast (p \ast (1 - k) + k) \)
  by (cases \( G \ s, \text{simp-all} \))

By the one-step termination assumption:
also from onestep nz1k
have \( \ldots \leq \N G \ s + \G s \ast (wp \ N \ G \ s \ast (1 - k) + k) \)
  by (auto intro: add-left-mono add-right-mono mult-left-mono mult-right-mono)

By scaling:
also from nz1k
have \( \ldots = \N G \ s + \G s \ast (wp \ (\lambda s. \N G \ s \ast (1 - k)) \ s + k) \)
  by (simp add: right-scalingD [OF sb])

By the maximality (termination) of the loop body:
4.11. LOOP TERMINATION

Lastly, by folding two loop iterations:

```
also from mb k0
have ... = \langle N G \rangle s + \langle G \rangle s \ast (wp body (\lambda s. \langle N G \rangle s \ast (1 - k)) s + wp body (\lambda s. k) s)
by(simp add:maximalD)
```

By sub-additivity of the loop body:

```
also from k0 nz1k
have ... \leq \langle N G \rangle s + \langle G \rangle s \ast (wp body (\lambda s. \langle N G \rangle s \ast (1 - k) + k) s)
by(auto intro!:add-left-mono multi-left-mono sub-addD[OF sub] sound-intros)
also
have ... = \langle N G \rangle s + \langle G \rangle s \ast (wp body (\lambda s. \langle N G \rangle s + \langle G \rangle s \ast k) s)
by(simp add:negate-embed algebra-simps)
```

By monotonicity of the loop body, and that \( k \) is a lower bound:

```
also from k0 bloop le-funD[OF X]
have ... \leq \langle N G \rangle s + \langle G \rangle s \ast (wp body (\lambda s. \langle N G \rangle s + \langle G \rangle s \ast wp do G \rightarrow body od (\lambda s. 1) s)
by(iprover intro!:add-left-mono multi-left-mono le-funI embed-ge-0
le-funD[OF mono-transD, OF healthy-monoD, OF hb]
sound-sum standard-sound sound-intros wp)
```

Unrolling the loop once and simplifying:

```
also {
have \( \lambda s. \langle N G \rangle s + \langle G \rangle s \ast wp body (wp do G \rightarrow body od (\lambda s. 1) s) = \langle N G \rangle s + \langle G \rangle s \ast (\langle N G \rangle s + \langle G \rangle s \ast wp body (wp do G \rightarrow body od (\lambda s. 1) s)) s \)
by(simp only:distrib-left mult.assoc[symmetric] embed-bool-idem embed-bool-cancel)
also have \( \lambda s. ... s = \langle N G \rangle s + \langle G \rangle s \ast wp do G \rightarrow body od (\lambda s. 1) s \)
by(simp add:fun-cong[OF wp-loop-unfold][symmetric, where P=\lambda s. 1, simplified, OF hb]])
finally have X: \( \lambda s. \langle N G \rangle s + \langle G \rangle s \ast wp body (wp do G \rightarrow body od (\lambda s. 1) s) s = \langle N G \rangle s + \langle G \rangle s \ast wp do G \rightarrow body od (\lambda s. 1) s \)
by(simp only:X)
}
```

Lastly, by folding two loop iterations:

```
also
have \( \langle N G \rangle s + \langle G \rangle s \ast (wp body (\lambda s. \langle N G \rangle s + \langle G \rangle s \ast wp body (wp do G \rightarrow body od (\lambda s. 1) s) s) = wp do G \rightarrow body od (\lambda s. 1) s \)
by(simp add:wp-loop-unfold[OF - hb, where P=\lambda s. 1, simplified, symmetric]
fun-cong[OF wp-loop-unfold[OF - hb, where P=\lambda s. 1, simplified, symmetric]]
```
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: } \bigwedge 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.

\begin{align*}
\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)}
\end{align*}

\text{let } \text{?k} = \text{Inf } (\text{range } (\wp \text{ do } G \rightarrow \text{body od } (\lambda s. 1)))

\text{from hloop have Inflb: } \bigwedge s. \text{?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 \text{ 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.

\begin{align*}
\text{finally have } &k1: \text{?k} < 1. \\
\text{hence } &\text{?k} \leq 1 \text{ by(auto)} \\
\text{moreover from } &\text{hloop have } k0: 0 \leq \text{?k} \\
\text{by(intro cInf-greatest, auto)}
\end{align*}

The infimum is, naturally, a lower bound.

\begin{align*}
\text{moreover from } &\text{Inflb have } (\lambda s. \text{?k}) \vdash \wp \text{ do } G \rightarrow \text{body od } (\lambda s. 1) \text{ by(auto)} \\
\text{ultimately}
\end{align*}

We can therefore use the previous result to find a new bound, …

\begin{align*}
\text{have } &\bigwedge s. p \ast (1 - \text{?k}) + \text{?k} \leq \wp \text{ do } G \rightarrow \text{body od } (\lambda s. 1) \ s

\text{by(blast intro:le-funD[OF new-bound])}
\end{align*}

… which is lower than the infimum, by minimality, …

\begin{align*}
\text{hence } &p \ast (1 - \text{?k}) + \text{?k} \leq \text{?k} \\
\text{by(blast intro:cInf-greatest)}
\end{align*}

… yet also strictly greater than it.

\begin{align*}
\text{moreover from } &k0 k1 \text{ have } \text{?k} < p \ast (1 - \text{?k}) + \text{?k} \text{ by(rule inc)}
\end{align*}

We thus have a contradiction.

\begin{align*}
\text{ultimately show } &\text{False by(simp)}
\end{align*}
4.12 Automated Reasoning

theory Automation imports StructuredReasoning
begin

This theory serves as a container for automated reasoning tactics for pGCL, implemented in ML. At present, there is a basic verification condition generator (VCG).

named-theorems wd
  theorems to automatically establish well-definedness

named-theorems pwp-core
  core probabilistic wp rules, for evaluating primitive terms

named-theorems pwp
  user-supplied probabilistic wp rules

named-theorems pwlp
  user-supplied probabilistic wlp rules

ML-file pVCG.ML

method-setup pvcg =
  ⟨⟨ Scan.succeed (fn ctxt => SIMPLE-METHOD' (pVCG.pVCG-tac ctxt)) ⟩⟩
  Probabilistic weakest preexpectation tactic

declare wd-intros[wd]

lemmas core-wp-rules =
  wp-Skip  wlp-Skip
  wp-Abort wlp-Abort
  wp-Apply wlp-Apply
  wp-Seq  wlp-Seq
  wp-DC-split wlp-DC-split
  wp-PC-fixed wlp-PC-fixed
  wp-SetDC  wlp-SetDC
  wp-SetPC-split wlp-SetPC-split

declare core-wp-rules[pwp-core]

end

qed
deqd
end
4.13 Miscellaneous Mathematics

theory Misc imports Real Multivariate-Analysis begin

lemma setsum-UNIV:
  fixes S :: 'a::finite set
  assumes complete: \(\forall x. x \notin S \implies f x = 0\)
  shows setsum f S = setsum f UNIV
proof (clarsimp)
  from complete have setsum f S = setsum f (UNIV - S) + setsum f S
  also have \(\ldots = \text{setsum f UNIV}\)
  finally show ?thesis .
qed

lemma cInf-mono:
  fixes A :: 'a::conditionally-complete-lattice set
  assumes lower: \(\forall b. b \in B \implies \exists a \in A. a \leq b\)
  and bounded: \(\forall a. a \in A \implies c \leq a\)
  and ne: \(B \neq \emptyset\)
  shows Inf A \leq Inf B
proof (rule cInf-greatest[OF ne])
  fix b assume bin: b \in B
  with lower obtain a where ain: a \in A and le: a \leq b
  from ain bounded have Inf A \leq a
  also note le
  finally show Inf A \leq b .
qed

lemma max-distrib:
  fixes c::real
  assumes nn: \(0 \leq c\)
  shows c * max a b = max (c * a) (c * b)
proof (cases a \leq b)
  case True
  moreover with nn have c * a \leq c * b
  ultimately show ?thesis
next
case False then have b \leq a
moreover with nn have c * b \leq c * a
by(auto intro:mult-left-mono)
ultimately show \( \text{thesis by}(\text{simp add:max.absorb1}) \)

qed

\textbf{lemma mult-div-mono-left:}
\begin{quote}
fixes \( c :: \text{real} \)
assumes nnc: \( 0 \leq c \) and nzc: \( c \neq 0 \)
and inv: \( \text{inverse } c \cdot a \leq b \)
shows \( c \cdot a \leq b \)
\end{quote}
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 nzc have \( \ldots = b \) by(simp)
finally show \( c \cdot a \leq b \).
qed

\textbf{lemma mult-div-mono-right:}
\begin{quote}
fixes \( c :: \text{real} \)
assumes nnc: \( 0 \leq c \) and nzc: \( c \neq 0 \)
and inv: \( \text{inverse } c \leq a \)
shows \( a \leq c \cdot b \)
\end{quote}
proof --
from nzc 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

\textbf{lemma min-distrib:}
\begin{quote}
fixes \( c :: \text{real} \)
assumes nnc: \( 0 \leq c \)
shows \( c \cdot \text{min } a \cdot b = \text{min } (c \cdot a) \cdot (c \cdot b) \)
\end{quote}
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 \( \text{thesis by}(\text{auto}) \)
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 \( \text{thesis by}(\text{simp add:min.absorb2}) \)
qed

\textbf{lemma nonempty-witness:}
\begin{quote}
\( S \neq \{ \} \imp \exists x. x \in S \)
by(blast)
\end{quote}

\textbf{lemma finite-set-least:}
\begin{quote}
fixes \( S ::'a::\text{linorder set} \)
assumes finite: \( \text{finite } S \)
\end{quote}
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 $x\::\: \text{a}$ and $S\::\: \text{a}$ set
  assume IH: $S = \{\} \lor (\exists x \in S. \ \forall y \in S. \ x \leq y)$
  show $(\forall y \in S. \ x \leq y) \lor (\exists x' \in S. \ x' \leq x \Rightarrow (\forall y \in S. \ x' \leq y))$
  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 $\text{zin}: z \in S$ and $\text{zmin}: \ \forall y \in S. \ z \leq y$ by(auto)
    thus $\text{thesis by(cases } z \leq x, \text{auto)}$
  qed
  qed
  with ne show $\text{thesis by(auto)}$
  qed

lemma $c\text{Sup-add}$:
  fixes $c\::\: \text{real}$
  assumes ne: $S \neq \{\}$
    and bs: $\forall x. \ x \in S \Rightarrow x \leq b$
  shows $\text{Sup } S + c = \text{Sup } \{ x + c | x. \ x \in S \}$
proof (rule antisym)
  from ne bs show $\text{Sup } \{ x + c | x. \ x \in S \} \leq \text{Sup } S + c$
    by(auto intro cSup-least add-right-mono cSup-upper bdd-aboveI)
  have $\text{Sup } S \leq \text{Sup } \{ x + c | x. \ x \in S \} - c$
  proof (intro cSup-least ne)
    fix $x$
    assume xin: $x \in S$
    from bs have $\forall x. \ x \in S \Rightarrow x + c \leq b + c$ by(auto intro:add-right-mono)
    hence $\text{bdd-above } \{ x + c | x. \ x \in S \}$ by(intro bdd-aboveI, blast)
    with xin have $x + c \leq \text{Sup } \{ x + c | x. \ x \in S \}$ by(auto intro:cSup-upper)
    thus $x \leq \text{Sup } \{ x + c | x. \ x \in S \} - c$ by(auto)
  qed
  thus $\text{Sup } S + c \leq \text{Sup } \{ x + c | x. \ x \in S \}$ by(auto)
  qed

lemma $c\text{Sup-mult}$:
  fixes $c\::\: \text{real}$
  assumes ne: $S \neq \{\}$
    and bs: $\forall x. \ x \in S \Rightarrow x \leq b$
    and nnc: $0 \leq c$
  shows $c \ast \text{Sup } S = \text{Sup } \{ c \ast x | x. \ x \in S \}$
proof (cases)
  assume $c = 0$
  moreover from ne have $\exists x. \ x \in S$ by(auto)
  ultimately show $\text{thesis by(simp)}$
next
assume cnz: \( c \neq 0 \)
show \( \vartheta \)thesis
proof (rule antisym)
  from \( bS \) have \( baS: \) bdd-above \( S \) by (intro \( \text{bdd-aboveI, auto} \))
  with \( ne \ nnc \) show \( \Sup \{ c \ast x \mid x \in S \} \leq c \ast \Sup S \)
    by (blast intro: \( c\text{Sup-least mult-left-mono\{OF c\text{Sup-upper}\} \))
  have \( \Sup S \leq \) inverse \( c \ast \Sup \{ c \ast x \mid x \in S \} \)
    by (intro \( c\text{Sup-least ne} \))
  fix \( x \) assume \( \text{xin} : x \in S \)
  moreover from \( bS \ nnc \) have \( \forall x \in S. x \leq B \)
    by (auto intro: cInf-lower)
  hence \( \text{lenInf}: \) \( \forall x \in S. -1 \ast \text{Inf} (\ominus S) \leq -x \)
    by (rule mult-left-mono-neg, auto)
  hence \( \text{lenInf}: \) \( \forall x \in S. x \leq -\text{Inf} (\ominus S) \)
    by (simp)
  with \( ne \ bS \) show \( \Sup S \leq -\text{Inf} ?T \)
    by (blast intro: \( c\text{Sup-least} \))

lemma closure-contains-Sup:
  fixes \( S :: \) real set
  assumes \( neS: S \neq \{\} \) and \( bS: \forall x \in S. x \leq B \)
  shows \( \Sup S \in \text{closure} S \)
proof ("")
  let \( ?T = \ominus S \)
  from \( neS \) have \( neT: ?T \neq \{\} \) by (auto)
  from \( bS \) have \( bT: \) \( \forall x \in ?T. -B \leq x \)
    by (auto intro: \( \text{bdd-below} ?T \) by (intro \( \text{bdd-belowI, blast} \))
  have \( \Sup S = -\text{Inf} ?T \)
    by (rule antisym)
  from \( neT \) have \( bbT \)
    have \( \forall x \in S. \) \( \text{Inf} (\ominus S) \leq -x \)
      by (blast intro: \( c\text{Inf-lower} \))
    hence \( \forall x \in S. -1 \ast -x \leq -1 \ast \text{Inf} (\ominus S) \)
      by (rule mult-left-mono-neg, auto)
    hence \( \text{lenInfi}: \) \( \forall x \in S. x \leq -\text{Inf} (\ominus S) \)
      by (simp)
    with \( neS \) have \( \Sup S \leq -\text{Inf} ?T \)
      by (blast intro: \( c\text{Sup-least} \))
have \( \sup S \leq \inf T \)

proof
\[
\begin{aligned}
&\text{fix } x \text{ assume } x \in \uminus ' S \\
&\text{then obtain } y \text{ where } y \in S \text{ and } rwx: x = -y \text{ by(auto)} \\
&\text{from } y \text{ in } S \text{ have } y \leq \sup S \\
&\text{hence } -1 \ast \sup S \leq -1 \ast y \\
&\text{with } rwx \text{ show } - \sup S \leq x \text{ by(simp)} \\
&\text{qed} \\
&\text{hence } -1 \ast \inf T \leq -1 \ast (-\sup S) \\
&\text{by(simp add: mult-left-mono-neg)} \\
&\text{thus } - \inf T \leq \sup S \text{ by(simp)} \\
&\text{qed}
\end{aligned}
\]

also \{ 
\[ \text{thm } bT \from \text{neT } \text{bbT} \text{ have } \inf T \in \text{closure } T \text{ by rule closure-contains-Inf} \]
\[\text{hence } - \inf T \in \uminus ' \text{closure } T \text{ by(auto)} \]
\}

also \{ 
\[ \text{have linear } \uminus \text{ by(auto intro:linearI)} \]
\[\text{hence } \uminus ' \text{closure } T \subseteq \text{closure } (\uminus ' T) \]
\[\text{by(rule closure-linear-image)} \]
\}

also \{ 
\[ \text{have } \uminus ' T \subseteq S \text{ by(auto)} \]
\[\text{hence } \text{closure } (\uminus ' T) \subseteq \text{closure } S \text{ by(rule closure-mono)} \]
\}

finally show \( \sup S \in \text{closure } S \) .

qed

lemma tendsto-min:
\[
\begin{aligned}
&\text{fixes } x, y: \text{real} \\
&\text{assumes } ta: \ a \ -----> x \\
&\text{and } tb: \ b -----> y \\
&\text{shows } (\lambda i. \min (a i) (b i)) -----> \min x y \\
&\text{proof } \text{rule LIMSEQ-I, simp} \\
&\text{fix } e: \text{real assume } \text{pe: } 0 < e \\
\end{aligned}
\]

\[
\begin{aligned}
&\text{from } ta \text{ pe obtain } \text{noa where balla: } \forall n \geq \text{noa. abs } (a n - x) < e \text{ by(auto dest:LIMSEQ-D)} \\
&\text{from } tb \text{ pe obtain } \text{nob where ballb: } \forall n \geq \text{nob. abs } (b n - y) < e \text{ by(auto dest:LIMSEQ-D)} \\
\end{aligned}
\]

\[
\begin{aligned}
&\{ \\
&\text{fix } n \\
&\text{assume gc: } \max \text{noa nob} \leq n \\
&\text{hence gc: } \text{noa} \leq n \text{ and } \text{geb: } \text{nob} \leq n \text{ by(auto)} \\
&\text{have abs } (\min (a n) (b n) - \min x y) < e \\
\end{aligned}
\]
proof cases
  assume le: min (a n) (b n) ≤ min x y
  show ⊨thesis
proof cases
  assume a n ≤ b n
  hence rwmin: min (a n) (b n) = a n by(auto)
  with le have a n ≤ min x y by(simp)
  moreover from gea balla have abs (a n - x) < e by(simp)
  moreover have min x y ≤ x by(auto)
  ultimately have abs (a n - min x y) < e by(auto)
  with rwmin show abs (min (a n) (b n) - min x y) < e by(simp)
next
  assume ¬ a n ≤ b n
  hence b n ≤ a n by(auto)
  hence rwmin: min (a n) (b n) = b n by(auto)
  with le have b n ≤ min x y by(simp)
  moreover from geb ballb have abs (b n - y) < e by(simp)
  moreover have min x y ≤ y by(auto)
  ultimately have abs (b n - min x y) < e by(auto)
  with rwmin show abs (min (a n) (b n) - min x y) < e by(simp)
qed
next
  assume ¬ min (a n) (b n) ≤ min x y
  hence le: min x y ≤ min (a n) (b n) by(auto)
  show ⊨thesis
proof cases
  assume x ≤ y
  hence rwmin: min x y = x by(auto)
  with le have x ≤ min (a n) (b n) by(simp)
  moreover from gea balla have abs (a n - x) < e by(simp)
  moreover have min (a n) (b n) ≤ a n by(auto)
  ultimately have abs (min (a n) (b n) - x) < e by(auto)
  with rwmin show abs (min (a n) (b n) - min x y) < e by(simp)
next
  assume ¬ x ≤ y
  hence y ≤ x by(auto)
  hence rwmin: min x y = y by(auto)
  with le have y ≤ min (a n) (b n) by(simp)
  moreover from geb ballb have abs (b n - y) < e by(simp)
  moreover have min (a n) (b n) ≤ b n by(auto)
  ultimately have abs (min (a n) (b n) - y) < e by(auto)
  with rwmin show abs (min (a n) (b n) - min x y) < e by(simp)
qed
qed

thus ∃ no. ∀ n≥no. |min (a n) (b n) - min x y| < e by(blast)
qed

definition supp :: ('s ⇒ real) ⇒ 's set
where \( \text{supp } f = \{ x. f x \neq 0 \} \)

**Definition dist-remove** :: \( \langle 's \Rightarrow \text{real} \rangle \Rightarrow 's \Rightarrow 's \Rightarrow \text{real} \)
where dist-remove \( p x = (\lambda y. \text{if } y=x \text{ then } 0 \text{ else } p y / (1 - p x)) \)

**Lemma supp-dist-remove:**
\( p x \neq 0 \implies p x \neq 1 \implies \text{supp } \text{dist-remove } p x = \text{supp } p - \{ x \} \)
by(auto simp:dist-remove-def supp-def)

**Lemma supp-empty:**
\( \text{supp } f = \{ \} \implies f x = 0 \)
by(simp add:supp-def)

**Lemma nsupp-zero:**
\( x \notin \text{supp } f \implies f x = 0 \)
by(simp add:supp-def)

**Lemma setsum-supp:**
\( \text{fixes } f::'a::\text{finite } \Rightarrow \text{real} \)
\( \text{shows } \text{setsum } f \text{ (supp } f \text{)} = \text{setsum } f \text{ UNIV} \)
**Proof**
\( \text{have } \text{setsum } f \text{ (UNIV } - \text{supp } f \text{)} = 0 \)
by(simp add:supp-def)
\( \text{hence } \text{setsum } f \text{ (supp } f \text{)} = \text{setsum } f \text{ (UNIV } - \text{supp } f \text{)} + \text{setsum } f \text{ (supp } f \text{)} \)
by(simp)
\( \text{also have } \ldots = \text{setsum } f \text{ UNIV} \)
by(simp add:setsum.subset-diff[symmetric])
finally show \( ?\text{thesis } \).
qed

### 4.13.1 Truncated Subtraction

**Definition**
\( \text{tminus } : \text{real } \Rightarrow \text{real } \Rightarrow \text{real} \ (\text{infix } \ominus \ 60) \)
where \( x \ominus y = \max (x - y) \ 0 \)

**Lemma minus-le-tminus**[intro,simp]:
\( a - b \leq a \ominus b \)
unfolding tminus-def by(auto)

**Lemma tminus-cancel-1:**
\( 0 \leq a \implies a + 1 \ominus 1 = a \)
unfolding tminus-def by(simp)

**Lemma tminus-zero-imp-le:**
\( x \ominus y \leq 0 \implies x \leq y \)
by(simp add:tminus-def)
lemma tminus-zero[simp]:
  \[ 0 \leq x \Rightarrow x \ominus 0 = x \]
  by(simp add:tminus-def)

lemma tminus-left-mono:
  \[ a \leq b \Rightarrow a \ominus c \leq b \ominus c \]
  unfolding tminus-def
  by(case-tac a \leq c, simp-all)

lemma tminus-less:
  \[ (0 \leq a; 0 \leq b) \Rightarrow a \ominus b \leq a \]
  unfolding tminus-def by(force)

lemma tminus-left-distrib:
  assumes nna: \[ 0 \leq a \]
  shows \[ a \ast (b \ominus c) = a \ast b \ominus a \ast c \]
  proof(cases b \leq c)
  case True
  note le = this
  hence \[ a \ast \max (b - c) 0 = 0 \] by(simp add:max.absorb2)
  also { 
    from nna le have \[ a \ast b \leq a \ast c \] by(blast intro:mult-left-mono)
    hence \[ 0 = \max (a \ast b - a \ast c) 0 \] by(simp add:max.absorb1)
  }
  finally show ?thesis by(simp add:tminus-def)

next
  case False
  hence le: \[ c \leq b \] by(auto)
  hence \[ a \ast \max (b - c) 0 = a \ast (b - c) \] by(simp only:max.absorb1)
  also { 
    from nna le have \[ a \ast c \leq a \ast b \] by(blast intro:mult-left-mono)
    hence \[ a \ast (b - c) = \max (a \ast b - a \ast c) 0 \] by(simp add:max.absorb1)
    field-simps
  }
  finally show ?thesis by(simp add:tminus-def)
qed

lemma tminus-le[simp]:
  \[ b \leq a \Rightarrow a \ominus b = a - b \]
  unfolding tminus-def by(simp)

lemma tminus-le-alt[simp]:
  \[ a \leq b \Rightarrow a \ominus b = 0 \]
  by(simp add:tminus-def)

lemma tminus-nle[simp]:
  \[ \neg b \leq a \Rightarrow a \ominus b = 0 \]
  unfolding tminus-def by(simp)

lemma tminus-add-mono:
  \[(a+b) \ominus (c+d) \leq (a\ominus c) + (b\ominus d)\]
proof\((\textrm{cases } 0 \leq a - c)\)

\textbf{case True note pac = this}

show \(\?\)thesis

\textbf{proof(\textrm{cases } 0 \leq b - d)\)

\textbf{case True note pbd = this}

from pac and pbd have \((c + d) \leq (a + b)\) by(simp)

\textbf{with pac and pbd show \(\?\)thesis by(simp)}

next

\textbf{case False with pac show \(\?\)thesis}

\textbf{by(cases \(c + d \leq a + b\), auto)}

next

\textbf{case False note nac = this}

show \(\?\)thesis

\textbf{proof(\textrm{cases } 0 \leq b - d)\)

\textbf{case True with nac show \(\?\)thesis}

\textbf{by(cases \(c + d \leq a + b\), auto)}

next

\textbf{case False note nbd = this}

\textbf{with nac have \(\neg(c + d) \leq (a + b)\) by(simp)}

\textbf{with nac and nbd show \(\?\)thesis by(simp)}

qed

\textbf{lemma tminus-setsum-mono:}

\textbf{assumes \(\ell S\): finite \(S\) }

\textbf{shows \(\setsum f S \ominus \setsum g S \leq \setsum (\lambda x. f x \ominus g x) S\) (is \(\?X S\))}

\textbf{proof(\textrm{rule finite-induct)\)

\textbf{from \(\ell S\) show finite \(S\) . \)

\textbf{\textbf{show \(\?X \{\}\) by(simp)}

\textbf{fix \(x\) and \(F\)

\textbf{assume \(\ell F\): finite \(F\) and \(\text{xniF: } x \notin F\)

\textbf{\text{and } IH: \(\?X F\)

\textbf{have \(f x + \setsum f F \ominus g x + \setsum g F \leq (f x \ominus g x) + (\setsum f F \ominus \setsum g F)\) by(rule tminus-add-mono)

\textbf{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)

\textbf{finally show \(\?X (\text{insert } x F)\)

\textbf{by(simp add:setsum.insert[OF \(\ell F\) \text{xniF}])}

\textbf{qed}\)

\textbf{lemma tminus-nneg[simp,intro]:}

\(0 \leq a \ominus b\)

\textbf{by(cases \(b \leq a\), auto)}
lemma \textit{tminus-right-antimono}:
\begin{itemize}
\item[assumes] \textit{clb}: \(c \leq b\)
\item[shows] \(a \ominus b \leq a \ominus c\)
\end{itemize}
\textbf{proof} (\textit{cases }\(b \leq a\))
\begin{itemize}
\item[case] \textit{True}
\begin{itemize}
\item [moreover] with \textit{clb} have \(c \leq a\) by (\textit{auto})
\item [moreover] note \textit{clb}
\item [ultimately] show ?thesis by (\textit{simp})
\end{itemize}
\end{itemize}
\textbf{next}
\begin{itemize}
\item[case] \textit{False} then show ?thesis by (\textit{simp})
\end{itemize}
\textbf{qed}

lemma \textit{min-tminus-distrib}:
\begin{itemize}
\item[\textit{min tminus-distirb}]:
\item[\textit{min} \(a \ominus b \ominus c = \text{min } (a \ominus c) (b \ominus c)\)]
\item [unfolding \textit{tminus-def} by (\textit{auto})]
\end{itemize}
\textbf{end}
Bibliography


