pGCL for Isabelle

David Cock

June 11, 2019
Contents

1 Overview 1

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

3 Semantic Structures 17
   3.1 Expectations ................................ 17
      3.1.1 Bounded Functions ....................... 18
      3.1.2 Non-Negative Functions .................. 20
      3.1.3 Sound Expectations ....................... 21
      3.1.4 Unitary expectations ..................... 25
      3.1.5 Standard Expectations .................... 25
      3.1.6 Entailment ................................ 27
      3.1.7 Expectation Conjunction .................. 28
      3.1.8 Rules Involving Conjunction ............... 29
      3.1.9 Rules Involving Entailment and Conjunction Together 31
   3.2 Expectation Transformers ..................... 33
      3.2.1 Comparing Transformers .................. 36
      3.2.2 Healthy Transformers ..................... 39
      3.2.3 Sublinearity ............................ 47
## CONTENTS

3.2.4 Determinism ........................................... 57
3.2.5 Modular Reasoning ................................. 60
3.2.6 Transforming Standard Expectations ............... 61

3.3 Induction .................................................. 62
3.3.1 The Lattice of Expectations ......................... 62
3.3.2 The Lattice of Transformers .......................... 66
3.3.3 Tail Recursion ........................................ 72

4 The pGCL Language ........................................ 75
4.1 A Shallow Embedding of pGCL in HOL ................ 75
4.1.1 Core Primitives and Syntax ......................... 75
4.1.2 Unfolding rules for non-recursive primitives .... 79

4.2 Healthiness ............................................... 81
4.2.1 The Healthiness of the Embedding ................. 81
4.2.2 Healthiness for Loops ............................... 94

4.3 Continuity ............................................... 104
4.3.1 Continuity of Primitives ........................... 105
4.3.2 Continuity of a Single Loop Step .................. 122

4.4 Continuity and Induction for Loops ................. 126
4.4.1 The Limit of Iterates ............................... 134

4.5 Sublinearity .............................................. 144
4.5.1 Nonrecursive Primitives ............................ 144
4.5.2 Sublinearity for Loops ............................... 150

4.6 Determinism .............................................. 154
4.6.1 Additivity ............................................. 154
4.6.2 Maximaly .............................................. 155
4.6.3 Determinism .......................................... 157

4.7 Well-Defined Programs ................................ 158
4.7.1 Strict Implies Liberal ............................... 158
4.7.2 Sub-Distributivity of Conjunction ................. 162
4.7.3 The Well-Defined Predicate ......................... 172

4.8 The Loop Rules .......................................... 175
4.8.1 Liberal and Strict Invariants ....................... 175
4.8.2 Partial Correctness .................................. 176
4.8.3 Total Correctness ................................... 176
4.8.4 Unfolding ............................................ 177

4.9 The Algebra of pGCL ................................... 178
4.9.1 Program Refinement ................................ 178
4.9.2 Simple Identities .................................... 179
4.9.3 Deterministic Programs are Maximal .............. 184
4.9.4 The Algebraic Structure of Refinement ............ 186
4.9.5 Data Refinement ..................................... 189
4.9.6 The Algebra of Data Refinement .................... 193
4.9.7 Structural Rules for Correspondence ............... 200
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 1. OVERVIEW
Chapter 2

Introduction to pGCL

2.1 Language Primitives

theory Primitives imports ../pGCL begin

Programs in pGCL are probabilistic automata. They can do anything a traditional program can, plus, they may make truly probabilistic choices.

2.1.1 The Basics

Imagine flipping a pair of fair coins: $a$ and $b$. Using a record type for the state allows a number of syntactic niceties, which we describe shortly:

```plaintext
datatype coin = Heads | Tails

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

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

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

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

### 2.1.2 Assertion and Annotation

Assertions in pGCL are real-valued functions of the state, which are often interpreted as a probability distribution over possible outcomes. These 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 \( P::'s \Rightarrow \) bool is embedded as \( « P »::'s \Rightarrow \) real, such that \( P s \rightarrow « P » s = 1 \land \neg P s \rightarrow « P » s = 0 \).

An annotation consists of an assertion on the initial state and one on the final state, which for standard expectations may be interpreted as ‘if \( P \) holds in the initial state, then \( Q \) will hold in the final state’. These are in weakest-precondition form: we assert that the precondition implies the weakest precondition: the weakest assertion on the initial state, which implies that the postcondition must hold on the final state. So far, this is identical to the standard approach. Remember, however, that we are working with real-valued assertions. For standard expectations, the logic is nevertheless identical, if the implication \( \forall s. P s \rightarrow Q s \) is substituted with the equivalent expectation entailment \( « P » \vdash « Q » \), \[ \llbracket « ?P » \vdash « ?Q » \rrbracket \Rightarrow ?Q \rrbracket = \Rightarrow ?Q \rrbracket \]. Thus a valid specification of \( \text{Apply} \left( \lambda s. s (a := b s) \right) \) is:

**lemma**

\[ \wedge x. « \lambda s. b s = x » \vdash wp (a := b) « \lambda s. a s = x » \]

**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**

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

**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 \( x \oplus y \) for a probabilistic choice between \( x \) and \( y \). This program behaves as \( x \) with probability \( p \), and as \( y \) with probability \( (1::'a) - p \). The probability may depend on the state, and is therefore of
2.1. LANGUAGE PRIMITIVES

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

**definition**

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

**where**

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

**lemma**

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

**unfolding** flip-a-def

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

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

2.1.4 Nondeterminism

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

**lemma**

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

**unfolding** flip-a-def

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

2.1.5 Properties of Expectations

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

**definition**

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

**where**

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

**lemma**

\[ (\lambda s. 1/4) = wp (\text{flip-a} (1/2) \sqcup \text{flip-b} (1/2)) \langle \lambda s. a = \text{Heads} \wedge b = \text{Tails} \rangle \]

**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.
CHAPTER 2. INTRODUCTION TO PGCL

record dice =
  red :: nat
  blue :: nat

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

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

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

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

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

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

The expectations of independent variables add:

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

end

2.2 Loops

theory LoopExamples imports ../pGCL begin

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

2.2.1 Guaranteed Termination

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

**definition**  
=countdown **::** int prog  

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

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

**definition**  
=inv-count **::** int ⇒ bool  

where  
=inv-count = (λx. 0 ≤ x)

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

**lemma**  
=wp-inv-count:  

wp-inv (λx. 0 < x) (Apply (λs. s − 1)) «inv-count»  

unfolding wp-inv-def inv-count-def wp-eval o-def  

proof(clarify, cases)  

fix x::int  

assume 0 ≤ x  
then show «λx. 0 < x» x * «λx. 0 ≤ x» x ≤ «λx. 0 ≤ x» (x − 1)  
by(simp add:embed-bool-def)

next  

fix x::int  

assume ¬ 0 ≤ x  
then show «λx. 0 < x» x * «λx. 0 ≤ x» x ≤ «λx. 0 ≤ x» (x − 1)  
by(simp add:embed-bool-def)

qed

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

**lemma**  
=term-countdown:  

«inv-count» ⊢ ⊢ wp countdown (λs. 1)  

unfolding countdown-def  

proof(intro loop-term-nat-measure[where m=λx. nat (max x 0)] wp-inv-count)  

let ?p = Apply (λx. x − 1::int)

As usual, well-definedness is trivial.

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

A measure of 0 implies termination.

show ∃x. nat (max x 0) = 0 → ¬ 0 < x  
by(auto)

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

This is the kind of proof that the VCG is good at. It leaves a purely logical goal, which we can solve with auto.
show \( \land n. \lambda x. \text{nat}(\text{max } x \ 0) = \text{Suc } n \) \&\& \( \langle \text{inv-counts} \vdash \) wp \? p \langle \lambda x. \text{nat}(\text{max } x \ 0) = n \rangle \\
unfolding \text{inv-count-def} \\
by(pvcg, \\
\text{auto simp: o-def exp-conj-std-split[symmetric]} \\
\quad \text{intro: implies-entails})
qed

2.2.2 Probabilistic Termination

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

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

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

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

Nonetheless, we can show termination.

lemma wait-for-heads-term: 
\( \lambda s. \frac{1}{2} \vdash \text{wp} \text{wait-for-heads} (\lambda s. 1) \) 
unfolding wait-for-heads-def

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

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

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

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

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

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

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

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

2.3 The Monty Hall Problem

theory Monty imports ../pGCL begin

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

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

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

2.3.1 The State Space

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

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

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

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

We prove explicitly that only valid doors are ever chosen.

**definition** inv-prize :: game ⇒ bool
**where**
inv-prize g ≡ prize g ∈ \{1, 2, 3\}

**definition** inv-clue :: game ⇒ bool
**where**
inv-clue g ≡ clue g ∈ \{1, 2, 3\}

**definition** inv-guess :: game ⇒ bool
**where**
inv-guess g ≡ guess g ∈ \{1, 2, 3\}

### 2.3.2 The Game

Hide the prize behind door \( D \).

**definition** hide-behind :: nat ⇒ game prog
**where**
hide-behind \( D \) ≡ Apply (prize-update (λx. \( D \)))

Choose door \( D \).

**definition** guess-behind :: nat ⇒ game prog
**where**
guess-behind \( D \) ≡ Apply (guess-update (λx. \( D \)))

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

**definition** open-door :: nat ⇒ game prog
**where**
on-open-door \( D \) ≡ Apply (clue-update (λx. \( D \)))

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

**definition** hide-prize :: game prog
**where**
hide-prize ≡ hide-behind 1 \( \cap \) hide-behind 2 \( \cap \) hide-behind 3

Guess uniformly at random.

**definition** make-guess :: game prog
**where**
make-guess ≡ guess-behind 1 (λs. 1/3) ⊕ guess-behind 2 (λs. 1/2) ⊕ guess-behind 3

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

**definition** reveal :: game prog
**where**
reveal ≡ \( \prod \) \( d \in (λs. \{1, 2, 3\} - \{prize s, guess s\}) \). open-door \( d \)

Switch your guess to the other unopened door.

**definition** switch-guess :: game prog
**where**
switch-guess ≡ \( \prod \) \( d \in (λs. \{1, 2, 3\} - \{clue s, guess s\}) \). guess-behind \( d \)

The complete game, either with or without switching guesses.

**definition** monty :: bool ⇒ game prog
2.3. THE MONTY HALL PROBLEM

where

\[ \text{monty switch} \equiv \text{hide-prize} ;; \]
\[ \text{make-guess} ;; \]
\[ \text{reveal} ;; \]
\[ (\text{if switch then switch-guess else Skip}) \]

2.3.3 A Brute Force Solution

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

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

**Lemma eval-loss [simp]:**
\[ p \neq g \implies \langle \text{player-wins} \rangle (s \langle \text{prize} := p, \text{guess} := g, \text{clue} := c \rangle) = 0 \]
by (simp add: embed-bool-def player-wins-def)

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

**Lemma wp-monty-noswitch:**
\[ (\lambda s. \frac{1}{3}) = \text{wp (monty False)} \langle \text{player-wins} \rangle \]
unfolding monty-def hide-prize-def make-guess-def reveal-def
\[ \text{hide-behind-def guess-behind-def open-door-def} \]
\[ \text{switch-guess-def} \]
by (simp add: wp-eval insert-Diff-if o-def)

**Lemma swap-upd:**
\[ s \langle \text{prize} := p, \text{clue} := c, \text{guess} := g \rangle = \]
\[ s \langle \text{prize} := p, \text{guess} := g, \text{clue} := c \rangle \]
by (simp)

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

**Lemma wp-monty-switch-bruteforce:**
\[ (\lambda s. \frac{2}{3}) = \text{wp (monty True)} \langle \text{player-wins} \rangle \]
unfolding monty-def hide-prize-def make-guess-def reveal-def
\[ \text{hide-behind-def guess-behind-def open-door-def} \]
\[ \text{switch-guess-def} \]
— Note that this is getting slow
by (simp add: wp-eval insert-Diff-if swap-upd o-def cong del: INF-cong-simp)

2.3.4 A Modular Approach

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

**Healthiness**

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

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

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

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

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

**lemma** *wd-reveal*:

well-def reveal

**proof** –

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

have $\forall s. \{1, 2, 3\} - \{\text{prize } s, \text{guess } s\} \neq \{\}$
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:**
\[(\lambda s. 1) \vdash \wp \text{hide-prize} \quad \text{«} \text{inv-prize} \text{»} \]

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

*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 \(a \in \{1::\text{n}at,2,3\}\) 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:**
\[\text{«} \lambda g. \text{guess g} \neq \text{prize g} \land \text{inv-prize g} \land \text{inv-guess g} \text{»} \vdash \wp \text{reveal} \text{«} \lambda g. \text{guess g} \neq \text{prize g} \land \text{clue g} \neq \text{prize g} \land \text{clue g} \neq \text{guess g} \land \text{inv-prize g} \land \text{inv-guess g} \land \text{inv-clue g} \text{»} \]

(is \(?X\vdash \wp \text{reveal} \ ?Y\))

*proof* (rule use-premise, rule well-def-wp-healthy[OF wd-reveal], clarify)
*fix* \(s\)
*assume* \(\text{guess s} \neq \text{prize s}\)
*and* \(\text{inv-prize s}\)
*and* \(\text{inv-guess s}\)

*moreover then obtain* \(c\)

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

*unfolding* inv-prize-def inv-guess-def
*by* (force dest:last-one 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.
Given the invariants, switching from the wrong guess gives the right one.

**Lemma** \( \text{wp-switch-guess} \):

\[
\lambda g. \text{guess } g \neq \text{prize } g \land \text{clue } g \neq \text{prize } g \land \text{clue } g \neq \text{guess } g \land
\text{inv-prize } g \land \text{inv-guess } g \land \text{inv-clue } g
\]

\[
\vdash \text{wp switch-guess} \left( \lambda \cdot \text{player-wins} \right)
\]

**Proof**

- **Fix** \( s \)
- **Assume** \( \text{guess } s \neq \text{prize } s \) and \( \text{clue } s \neq \text{prize } s \)
- and \( \text{clue } s \neq \text{guess } s \) and \( \text{inv-prize } s \)
- and \( \text{inv-guess } s \) and \( \text{inv-clue } s \)

**Note** \( \text{state} = \text{this} \)

**Hence** \( 1 \leq \text{Inf} \left( \lambda a. \text{player-wins} \right) \left( s[\text{guess} := a] \right) \)

\[
\left( \{\text{guess }, \text{prize } s, \text{clue } s\} - \{\text{clue } s, \text{guess } s\} \right)
\]

by \( \text{auto simp:insert-Diff-if player-wins-def} \)

**Also from state**

- **Have** \( ... = \text{Inf} \left( \lambda a. \text{player-wins} \right) \left( s[\text{guess} := a] \right) \)

\[
\left( \{1, 2, 3\} - \{\text{clue } s, \text{guess } s\} \right)
\]

by \( \text{simp add:distinct-game[symmetric]} \)

**Also have** \( ... = \text{wp switch-guess \{player-wins\} } s \)

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

**Finally show** \( 1 \leq \text{wp switch-guess \{player-wins\} } s \).

**Qed**

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

**Lemma** \( \text{wp-monty-switch-modalar} \):

\[
\lambda s. \frac{2}{3} \vdash \text{wp (monty True \{player-wins\}}
\]

**Proof**

- **Work in probabilistic Hoare triples**
- **Note** \( \text{wp-validI[OF wp-scale, OF wp-hide-prize, simplified]} \)
  - Here we apply scaling to match our pre-expectation
- **Also note** \( \text{wp-validI[OF wp-make-guess]} \)
- **Also note** \( \text{wp-validI[OF wp-reveal]} \)
- **Also note** \( \text{wp-validI[OF wp-switch-guess]} \)
- **Finally show** \( \lambda s. \frac{2}{3} \vdash \text{monty True \{player-wins\}} \)

**Unfolding** \( \text{monty-def} \)
by(simp add:wd-intros sound-intros monty-healthy)
qed

Using the VCG

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

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

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

end
Chapter 3

Semantic Structures

3.1 Expectations

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

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

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a → 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 \( \Rightarrow \) (\( \lambda a \Rightarrow \) 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]:
\[
[ \forall x. \ P x \leq b ] \Rightarrow \text{bounded-by } b \ P
\]
by (simp add:bounded-by-def)

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

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

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

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

**definition** bounded :: (\( \lambda a \Rightarrow \) 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 :: (\( \lambda a \Rightarrow \) real) \( \Rightarrow \) real
**where** bound-of P \( \equiv \text{Sup } (P : \text{UNIV}) \)

**lemma** bounded-bdd-above[intro]:
**assumes** bP: bounded P
**shows** bdd-above (range P)
**proof**
five x assume x \in range P
3.1. EXPECTATIONS

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

The least upper bound has the usual properties:

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

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

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

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

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

This is occasionally easier to apply:

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

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

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

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

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

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

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

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

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

3.1.2 Non-Negative Functions.

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

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

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

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

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

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

lemma nneg-bdd-below[intro]:
\( \text{nneg } P \Rightarrow \text{bdd-below } (\text{range } P) \)
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 soundness, or deriving a simple consequence (e.g. sound P → 0 ≤ P s) will usually follow by blast, force or simp.

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

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

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

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

lemma bound-of-sound[intro]:
assumes sP: sound P
CHAPTER 3. SEMANTIC STRUCTURES

shows $0 \leq \text{bound-of } P$

using assms by(auto)

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

**Lemma sound-sum[simp,intro]:**
assumes $sP$: sound $P$ and $sQ$: sound $Q$
shows sound $\lambda s. P s + Q s$

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

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

qed

**Lemma mult-sound:**
assumes $sP$: sound $P$ and $sQ$: sound $Q$
shows sound $\lambda s. P s \times Q s$

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

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

qed

**Lemma div-sound:**
assumes $sP$: sound $P$ and $\text{cpos}$: $0 < c$
shows sound $\lambda s. P s / c$

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

qed

**Lemma tminus-sound:**
assumes $sP$: sound $P$ and $\text{nnc}$: $0 \leq c$
3.1. EXPECTATIONS

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

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

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

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

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

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

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

lemma bounded-by-add:
fixes P::'s \Rightarrow real and Q
assumes bP: bounded-by a P
      and bQ: bounded-by b Q
shows bounded-by (a + b) (\lambda s. P s + Q s)
using asms 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 \ast Q s)
proof (rule bounded-byI)
  fix s
  have P s \ast Q s \leq 1 \ast 1
    using asms by (blast dest:bounded-by-mult)
  thus P s \ast Q s \leq 1 by (simp)
qed

lemma sum-sound:
assumes sP: \forall x \in S. sound (P x)
shows sound (\lambda s. \sum x \in S. P x s)
proof (rule soundI2)
  from sP show bounded-by (\sum x \in S. bound-of (P x)) (\lambda s. \sum x \in S. P x s)
    by (auto intro!:sum-mono)
  from sP show nneg (\lambda s. \sum x \in S. P x s)
    by (auto intro!:sum-nonneg)
qed
3.1.4 Unitary expectations

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

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

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

lemma unitaryI2:
  [ nneg P; bounded-by 1 P ] ⇒ unitary P
  by(auto)

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

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

3.1.5 Standard Expectations

definition embed-bool :: (′s ⇒ bool) ⇒ ′s ⇒ real (« - » 1000)
where
  «P» ≡ (λs. if P s then 1 else 0)

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

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

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

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

Standard expectations have a number of convenient properties, which mostly
follow from boolean algebra.
lemma embed-bool-idem:
\[ «P» s * «P» s = «P» s \]
by (simp add:embed-bool-def)

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

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

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

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

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

lemma expect-1-I:
\[ P x \Longrightarrow 1 \leq «P» x \]
by (simp)

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

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

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

definition negate :: ('s => bool) => 's => bool (N)
where negate \(P = (\lambda s. \neg P s)\)

lemma negateI:
\[ \neg P s \Longrightarrow N P s \]
by (simp add:negate-def)

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

lemma negate-embed:
\[ \langle \neg P \rangle s = 1 - \langle P \rangle s \]
by (simp add: embed-bool-def negate-def)

lemma eval-nembed-true[simp]:
\[ P s \implies \langle \neg P \rangle s = 0 \]
by (simp add: embed-bool-def negate-def)

lemma eval-nembed-false[simp]:
\[ \neg P s \implies \langle \neg P \rangle s = 1 \]
by (simp add: embed-bool-def negate-def)

lemma negate-Not[simp]:
\[ \neg \text{Not} = (\lambda x. x) \]
by (simp add: negate-def)

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

lemma embed-bool-cancel:
\[ \langle G \rangle s * \langle \neg G \rangle s = 0 \]
by (cases G s, simp-all)

3.1.6 Entailment

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

abbreviation entails :: \( (s \Rightarrow \text{real}) \Rightarrow (s \Rightarrow \text{real}) \Rightarrow \text{bool} \)
where \( P \vdash Q \equiv P \leq Q \)

lemma entailsI[intro]:
\[ \forall s. P s \leq Q s \implies P \vdash Q \]
by (simp add: le-funI)

lemma entailsD[dest]:
\[ P \vdash Q \implies P s \leq Q s \]
by (simp add: le-funD)

lemma eq-entails[intro]:
\[ P = Q \implies P \vdash Q \]
by (blast)

lemma entails-trans[trans]:
\[ [ P \vdash Q; Q \vdash R ] \implies P \vdash R \]
by (blast intro:order-trans)

For standard expectations, both notions of entailment coincide. This result justifies the above claim that our definition generalises predicate entailment:
**CHAPTER 3. SEMANTIC STRUCTURES**

**Lemma implies-entails:**
\[
\[ \bigwedge s. \ P s \Rightarrow Q s \ \Rightarrow \ \{ P \} \vdash \{ Q \} \]
by (rule entailsI, case-tac P s, simp-all)

**Lemma entails-implies:**
\[
\bigwedge s. \ \{ P \} \vdash \{ Q \}; \ P s \ \Rightarrow \ Q s
\]
by (rule ccontr, drule-tac s = s in entailsD, simp)

### 3.1.7 Expectation Conjunction

**Definition**
\[
\begin{align*}
\text{pconj} & : \text{real} \Rightarrow \text{real} \Rightarrow \text{real} \\
\text{where} & \\
p . \& q & \equiv p + q \ominus 1
\end{align*}
\]

**Definition**
\[
\begin{align*}
\text{exp-conj} & : (\text{'}s \Rightarrow \text{real}) \Rightarrow (\text{'}s \Rightarrow \text{real}) \Rightarrow (\text{'}s \Rightarrow \text{real}) \\
\text{where} & \\
a \&\& b & \equiv \lambda s. (a s .\& b s)
\end{align*}
\]

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

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

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

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

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

**Lemma** pconj-bconj:
\[
\{ a \} s .\& \{ b \} s = \{ \lambda s. a s \& b s \} s
\]
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:
\[
\begin{align*}
\[ 0 \leq a; \ a \leq 1; \ 0 \leq b; \ b \leq 1; \ 0 \leq c; \ c \leq 1 \ \Rightarrow \]
\end{align*}
\]
3.1. EXPECTATIONS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.1.8 Rules Involving Conjunction.

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

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

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

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

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

by\( \text{simpl add: exp-conj-def ac-simps})\)

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

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

fix \( x \)
from \( bP \) have \( P x \leq 1 \) by (blast)
moreover from \( bQ \) have \( Q x \leq 1 \) by (blast)
ultimately have \( P x + Q x \leq 2 \) by (auto)
thus \( P x + Q x \ominus 1 \leq 1 \)
unfolding tminus-def by (simp)

qed

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

**lemma** exp-conj-assoc:
assumes unitary \( P \) and unitary \( Q \) and unitary \( R \)
shows \( P \&\& (Q \&\& R) = (P \&\& Q) \&\& R \)
unfolding 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** exp-conj-top-left [simp]:
sound \( P \implies \<\lambda s. \True> \&\& P = P \)
unfolding exp-conj-def by (force)

**lemma** exp-conj-top-right [simp]:
sound \( P \implies P \&\& \<\lambda s. \True> = P \)
unfolding exp-conj-def by (force)

**lemma** exp-conj-idem [simp]:
\(<\lambda s. 0> \&\& <\lambda s. 0> = <\lambda s. 0>\)
unfolding exp-conj-def
by (rule ext, cases \( P s \), simp-all)

**lemma** exp-conj-nneg [intro, simp]:
\((\lambda s. 0) \leq P \&\& Q \)
unfolding exp-conj-def
by (blast intro: le-funI)
lemma \textit{exp-conj-sound\{intro,simp\}}:

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

lemma \textit{exp-conj-rzero\{simp\}}:

\textit{bounded-by} 1 P \Rightarrow P \& (\lambda \cdot 0) = (\lambda \cdot 0) \\
\textbf{unfolding} \textit{exp-conj-def} \textbf{by}\(\text{force}\) \\

lemma \textit{exp-conj-1-right\{simp\}}:

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

lemma \textit{exp-conj-std-split}:

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

3.1.9 Rules Involving Entailment and Conjunction Together

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

lemma \textit{entails-frame}:

\textbf{assumes} ePR: \(P \vdash R\) \\
\textbf{and} eQS: \(Q \vdash S\)  \\
\textbf{shows} \(P \&\& Q \vdash R \&\& S\) \\
\textbf{proof}(\text{rule le-funI}) \\
\textbf{fix} s
from ePR have \( P \ s \leq R \ s \) by (blast)
moreover from eQS have \( Q \ s \leq S \ s \) by (blast)
ultimately have \( P \ s + Q \ s \leq R \ s + S \ s \) by (rule add-mono)
hence \( P \ s + Q \ s \ominus 1 \leq R \ s + S \ s \ominus 1 \) by (rule tminus-left-mono)
thus \( (P \ & \& Q) \ s \leq (R \ & \& S) \ s \)
unfolding exp-conj-def pconj-def.

qed

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

lemma pentails-cases:
assumes \( \mathit{PQe} : \forall x. P \ x \vdash Q \ x \)
and exhaust: \( \forall s. \exists x. P \ (x \ s) \ s = 1 \)
and framed: \( \forall x. P \ x \ &\& R \vdash Q \ x \ &\& S \)
and \( \mathit{bQ} : \forall x. \text{bounded-by} \ 1 \ (Q \ x) \)
shows \( R \vdash S \)
proof (rule le-funI)
fix \( s \)
from exhaust obtain \( x \) where \( P\text{-xs}: P \ x \ s = 1 \) by (blast)
moreover {
  hence \( 1 = P \ x \ s \) by (simp)
  also from \( \mathit{PQe} \) have \( P \ x \ s \leq Q \ x \ s \) by (blast dest: le-funD)
  finally have \( Q \ x \ s = 1 \)
  using \( \mathit{bQ} \) by (blast intro: antisym)
}
moreover note le-funD[OF framed[where \( x=x \)], where \( x=s \)]
moreover from \( \mathit{sR} \) have \( 0 \leq R \ s \) by (blast)
moreover from \( \mathit{sS} \) have \( 0 \leq S \ s \) by (blast)
ultimately show \( R \ s \leq S \ s \) by (simp add: exp-conj-def)
qed

lemma unitary-bot[iff]:
  unitary \( (\lambda s. 0 :: \text{real}) \)
  by (auto)
lemma unitary-top[iff]:
  unitary \( (\lambda s. 1 :: \text{real}) \)
  by (auto)
lemma unitary-embed[iff]:
  unitary \( \langle P \rangle \)
  by (auto)
lemma unitary-const[iff]:
  \[[ 0 \leq c; c \leq 1 \] \implies unitary \( (\lambda s. c) \)
  by (auto)
lemma unitary-mult:
3.2. EXPECTATION TRANSFORMERS

assumes \( uA \): unitary \( A \) and \( uB \): unitary \( B \)
shows unitary \((\lambda s. A \ s \ast B \ s)\)

proof (intro unitaryI2 nnegI bounded-byI)

fix \( s \)

from asms have \( nnA: 0 \leq A \ s \) and \( nnB: 0 \leq B \ s \) by (auto)
thus \( 0 \leq A \ s \ast B \ s \) by (rule mult-nonneg-nonneg)

from asms have \( A \ s \leq 1 \) and \( B \ s \leq 1 \) by (auto)

with \( nnB \) have \( A \ s \ast B \ s \leq 1 \) by (intro mult-mono, auto)
also have \( ... = 1 \) by (simp)
finally show \( A \ s \ast B \ s \leq 1 \).

qed

lemma exp-conj-unitary:
\[ \begin{array}{l}
\text{unitary } P; \text{ unitary } Q \Rightarrow \text{unitary } (P \&\& Q) \\
\text{by (intro unitaryI2 nnegI2, auto)}
\end{array} \]

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

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

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

end

3.2 Expectation Transformers

theory Transformers imports Expectations begin type-synonym \('s\) trans = \('s\) expect \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: 

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

but this need not be the case.

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

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

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

![A diverging automaton.](image)

Figure 3.3: A diverging automaton.

arguments in Section 4.11.

Having reached \( a \), the automaton will proceed to \( b \) with probability \( 0.5 \ast \left( \frac{1}{0.5 + 0.3} \right) = 0.625 \), 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, \( P_{\text{prior}}(e) = 0.3125 \ast P_{\text{post}}(b) + 0.1875 \ast P_{\text{post}}(c) \). 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
le-trans :: 's trans ⇒ 's trans ⇒ bool
where
le-trans t u ≡ ∀ P. sound P → t P ≤ u P

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

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

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

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

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

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

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

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

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

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

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

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

Definition
l-trans :: 's trans \Rightarrow 's trans \Rightarrow bool
where
l-trans t u \iff le-trans t u \land \neg le-trans u t

Transformer equivalence is induced by comparison:

Definition
equiv-trans :: 's trans \Rightarrow 's trans \Rightarrow bool
where
equiv-trans t u \iff le-trans t u \land le-trans u t

Definition
equiv-utrans :: 's trans \Rightarrow 's trans \Rightarrow bool
where
equiv-utrans t u \iff le-utrans t u \land le-utrans u t

Lemma equiv-transI[intro]:
\[ \forall P. \text{sound } P \implies t P = u P \] \implies equiv-trans t u

Unfolding equiv-trans-def by(force)

Lemma equiv-utransI[intro]:
\[ \forall P. \text{sound } P \implies t P = u P \] \implies equiv-utrans t u

Unfolding equiv-utrans-def by(force)

Lemma equiv-transD[dest]:
\[ \text{equiv-trans } t u; \text{sound } P \] \implies t P = u P

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

Lemma equiv-utransD[dest]:
\[ \text{equiv-utrans } t u; \text{unitary } P \] \implies t P = u P

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

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

By(blast)

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

By(blast)

Lemma le-trans-antisym:
\[ \text{le-trans } x y; \text{le-trans } y x \] \implies equiv-trans x y

Unfolding equiv-trans-def by(simp)

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

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

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

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

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

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

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

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

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

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

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

by(force simp:ac-simps)

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

by(force simp:ac-simps)

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

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

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

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

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

by(simp)

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

by(simp)

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

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

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

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

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

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

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

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

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

3.2.2 Healthy Transformers

Feasibility

definition feasible :: (('a \Rightarrow \text{real}) \Rightarrow ('a \Rightarrow \text{real})) \Rightarrow \text{bool}
where feasible t \iff (\forall P. \text{bounded-by } b P \land \text{nneg } P \implies \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 feasibleI[intro]:**

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

by (force simp: feasible-def)

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

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

by (simp add: feasible-def)

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

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

by (simp add: feasible-def)

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

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

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

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

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

assumes ft: feasible \( t \)

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

proof (rule ext, rule antisym)

fix \( s \)

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

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

thus \( t \left( \lambda \cdot 0 \right) \leq 0 \) by (blast)

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

with ft have \( \text{nneg } \left( t \left( \lambda \cdot 0 \right) \right) \) by (blast)

thus \( 0 \leq t \left( \lambda \cdot 0 \right) \) by (blast)

qed

**lemma feasible-id:**

feasible \( \left( \lambda x. x \right) \)

unfolding feasible-def by (blast)

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

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

by (auto)

**lemma feasible-fixes-top:**

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

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

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

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

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

Monotonicity

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

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

Given another, monotonic, transformer \( u \), we have that \( u Q \vdash u (t R) \), or that the weakest precondition of \( Q \) under \( u \) entails that of \( R \) under the composition \( u \circ t \). If we additionally know that \( P \not\vdash u Q \), then by transitivity we have \( P \vdash u (t R) \). We thus derive a probabilistic form of the standard rule for sequential composition: \([\text{mono-trans } t; P \vdash u Q; Q \vdash t R] \implies P \vdash u (t R)\).

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**

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

where

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

The \textit{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 \textit{unitary} expectations (those bounded by 1): \( t \ P \ s = \text{bound-of} \ P \ast t \ (\lambda s. \ P \ s / \text{bound-of} \ P) \ s \). Feasibility in turn ensures that the lattice of unitary expectations is closed under the action of a healthy transformer. We take advantage of this fact in Section 3.3, in order to define the fixed points of healthy transformers.

**Lemma** \textit{scalingI}\[\text{[intro]}\]:

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

by (\textit{simp add:scaling-def})

**Lemma** \textit{scalingD}\[\text{[dest]}\]:

\[ c \ast t \ P \ x = t \ (\lambda x. \ c \ast P \ x) \ x \]

by (\textit{simp add:scaling-def})

**Lemma** \textit{right-scalingD}:

\textbf{assumes} \( st: \text{scaling} t \)

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

\textbf{and} \( nnc: \ 0 \leq c \)

\textbf{shows} \( t \ P \ s \ast c = t \ (\lambda s. \ P \ s \ast c) \ s \)

**Proof** –

\textbf{have} \( t \ P \ s \ast c = c \ast t \ P \ s \)

\textbf{by (simp add:algebra-simps)}

\textbf{also from} \textbf{assms} \textbf{have} \( ... = t \ (\lambda s. \ c \ast P \ s) \ s \)

\textbf{by (rule scalingD)}

\textbf{also have} \( ... = t \ (\lambda s. \ P \ s \ast c) \ s \)

\textbf{by (simp add:algebra-simps)}

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

**QED**

Healthiness

Healthy transformers are feasible and monotonic, and respect scaling.

**Definition**

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

where

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

**Lemma** \textit{healthyI}\[\text{[intro]}\]:

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

\[ \Rightarrow \text{healthy} t \]

by (\textit{simp add:healthy-def})
lemmas healthy-parts = healthyI[OF feasibleI mono-transI scalingI]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

show mono-trans u
proof
  fix P::'s \Rightarrow real and Q::'s \Rightarrow real
  assume sP: sound P and sQ: sound Q
    and le: P \vdash Q
  from sP and le-t-u have u P \vdash t P by (blast)
  also from sP and sQ and le and healthy have t P \vdash t Q by (blast)
  also from sQ and le-t-u have t Q \vdash u Q by (blast)
  finally show u P \vdash u Q.
3.2. EXPECTATION TRANSFORMERS

qed

show scaling u
proof
  fix P::'s ⇒ real and c::real and x:'s
  assume sound: sound P
  and pos: 0 ≤ c

  hence bounded-by (c * bound-of P) (λx. c * P x)
    by (blast intro:mult-left-mono dest!:less-imp-le)
  hence sc-bounded: bounded (λx. c * P x)
    by (blast)
  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 ...
    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 and x::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 \( \bigwedge s. c \cdot t P s \leq c \cdot b \) by (blast intro:mult-left-mono)
also { 
from \( mnP \) and \( bP \) have \( \emptyset \leq b \) by (auto)
with bc have \( c \cdot b \leq 1 \cdot b \) by (blast intro:mult-right-mono)
hence \( c \cdot b \leq b \) by (simp)
}
finally show bounded-by b \( (\lambda s. c \cdot t P s) \) by (blast)

from \( ht \) \( nnP bP \) have \( \bigwedge s. 0 \leq t P s \) by (blast)
with nc have \( \bigwedge s. 0 \leq c \cdot t P s \) by (rule mult-nonneg-nonneg)
thus \( \text{nneg} (\lambda s. c \cdot t P s) \) by (blast)
qed

show mono-trans \( (\lambda P s. c \cdot t P s) \)
proof
fix \( P :: \then s \Rightarrow \text{real} \) and \( Q \)
assume \( sP: \text{sound P} \) and \( sQ: \text{sound Q} \) and \( \text{le}: P \vdash Q \)
with \( ht \) have \( \bigwedge s. t P s \leq t Q s \) by (auto intro:le-funD)
with nc have \( \bigwedge s. c \cdot t P s \leq c \cdot t Q s \)
by (blast intro:mult-left-mono)
thus \( \lambda s. c \cdot t P s \vdash \lambda s. c \cdot t Q s \) by (blast)
qed

lemmas nearly-healthy = 
\begin{itemize}
\item healthy-top \[iff\]: \( \text{healthy} (\lambda P s. \text{bound-of P}) \) by (auto intro!:healthy-parts)
\item healthy-bot \[iff\]: \( \text{healthy} (\lambda P s. \emptyset) \) by (auto intro!:healthy-parts)
\end{itemize}

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

definition
\[
nearly-healthy :: (('s \Rightarrow \text{real}) \Rightarrow ('s \Rightarrow \text{real})) \Rightarrow \text{bool}
\]
where
\[
nearly-healthy t \longleftrightarrow (\forall P. \text{unitary P} \rightarrow \text{unitary} (t P)) \land
(\forall P Q. \text{unitary P} \rightarrow \text{unitary Q} \rightarrow P \vdash Q \rightarrow t P \vdash t Q)
\]

lemmas nearly-healthy = 
\begin{itemize}
\item healthy-top \[intro\]: \( \bigwedge P. \text{unitary P} \Rightarrow \text{unitary} (t P); \)
\item healthy-bot \[intro\]: \( \bigwedge P Q. \big[ \text{unitary P; unitary Q; P \vdash Q} \big] \Rightarrow t P \vdash t Q \big]\Rightarrow nearly-healthy t
\end{itemize}

lemmas nearly-healthy-monoD = 
\begin{itemize}
\item nearly-healthy-mono \[intro\]: \( \bigwedge P. \text{unitary P} \Rightarrow \text{unitary} (t P);
\item healthy-top \[dest\]: 
\end{itemize}
3.2. EXPECTATION TRANSFORMERS

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

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

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

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

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

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

\text{lemma sublinearD[dest]:}
\[ \forall a b c P Q s. (\text{sublinear } t; \text{sound } P; \text{sound } Q; 0 \leq a; 0 \leq b; 0 \leq c \) \implies
\[ a \ast t P s + b \ast t Q s \ominus c \leq t (\lambda s'. a \ast P s' + b \ast Q s' \ominus c) s \]
\text{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.
CHAPTER 3. SEMANTIC STRUCTURES

Sub-additivity

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

where

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

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 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.

Figure 3.4: A graphical depiction of sub-additivity as convexity.
that respect scaling. Note the form of the definition of convexity:

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

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

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

Can be rewritten, using scaling, to:

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

Which holds everywhere exactly when $v$ is sub-additive i.e.:

$$vR + vS \leq v(R + S)$$

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

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

by(simp add:sub-add-def)

**lemma** sub-addI2:

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

by(auto)

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

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

by(simp add:sub-add-def)

**lemma** equiv-sub-add:

fixes $t::(s \Rightarrow real) \Rightarrow s \Rightarrow real$

assumes $eq: \ equiv-trans t u$

and $sa: \ sub-add t$

shows $\ sub-add u$

**proof**

fix $P::s \Rightarrow real$ and $Q::s \Rightarrow real$ and $s::s$

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

with $eq$ have $u P s + u Q s = t P s + t Q s$

by(simp add:equiv-transD)

also from $sP$ $sQ$ $sa$ have $t P s + t Q s \leq t (\lambda s. P s + Q s) s$

by(auto)

also {

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

Sublinearity and feasibility imply sub-additivity.

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

A few properties following from sub-additivity:

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

lemma sub-add-sum:
fixes t::'s trans and S::'a set
assumes sat: sub-add t
and ht: healthy t
and sP: ∃x. sound (P x)
shows (∀x. ∑y∈S. t (P y) x) ≤ t (∀x. ∑y∈S. P y x)

proof (cases infinite S, simp-all add:ht)
assume fS: finite S
show ?thesis
proof (rule finite-induct[of fS le-funI le-funI], simp-all)
  fix s::'s
  from ht have sound (t (∀s. 0)) by(auto)
  hence t (λs. 0) s by(auto)

  fix F::'a set and x::'a
  assume IH: λa. ∑y∈F. t (P y) a t (∀x. ∑y∈F. P y x)
  hence t (P x) s + (∑y∈F. t (P y) s) ≤
                     t (P x) s + t (∀x. ∑y∈F. P y x) s
    by(auto intro: add-left-mono)
  also from sat sP
  have ... ≤ t (∀x. P x xa + (∑y∈F. P y xa)) s
    by(auto intro!: addD[OF sat] sum-sound)
  finally
  show t (P x) s + (∑y∈F. t (P y) s) ≤
                     t (∀x. P x xa + (∑y∈F. P y xa)) s .
  qed
qed

lemma sub-add-guard-split:
fixes t::'s finite trans and P::'s expect and s::'s
assumes sat: sub-add t
and ht: healthy t
and sP: sound P
shows (∀G. ∑y∈{s. G s}. P y t « λz. z = y » s) +
       (∑y∈{s. ¬G s}. P y t « λz. z = y » s) ≤ t P s

proof (cases {s. G s} ∩ {s. ¬G s} = {∅}, simp)
  have (∀G. ∑y∈{s. G s}. P y t « λz. z = y » s) +
       (∑y∈{s. ¬G s}. P y t « λz. z = y » s) =
       (∑y∈{s. G s} ∪ {s. ¬G s}, P y t « λz. z = y » s)
    by(auto intro: sum.union-disjoint[symmetric])
  also { have {s. G s} ∪ {s. ¬G s} = UNIV by (blast)
            hence (∑y∈{s. G s} ∪ {s. ¬G s}. P y t « λz. z = y » s) =
              (∀x. ∑y∈UNIV. P y t (λx. «λz. z = y» x) s)
        by(simps)
  } also {
from $sP$ have $\forall y. \ 0 \leq P y$ by(auto)
with healthy-scalingD[OF ht]
have $(\forall x. \sum y \in \text{UNIV}. \ P y \ast t (\lambda x. \ «\lambda z. \ z = y\» \ x) ) \ s =$
  $(\forall x. \sum y \in \text{UNIV}. \ t (\lambda x. \ P y \ast «\lambda z. \ z = y\» \ x) ) \ s$
by(simp add:scalingD)
}\} also { from sat ht $sP$
have $(\forall x. \sum y \in \text{UNIV}. \ t (\lambda x. \ P y \ast «\lambda z. \ z = y\» \ x) ) \leq$
  $(\forall x. \sum y \in \text{UNIV}. \ P y \ast «\lambda z. \ z = y\» \ x)$
by(intro sub-add-sum sound-intros, auto)
hence $(\forall x. \sum y \in \text{UNIV}. \ t (\lambda x. \ P y \ast «\lambda z. \ z = y\» \ x) ) \ s \leq$
  $(\forall x. \sum y \in \text{UNIV}. \ P y \ast «\lambda z. \ z = y\» \ x) \ s$ by(auto)
}\} also { have nw1: $(\forall x. \sum y \in \text{UNIV}. \ P y \ast «\lambda z. \ z = y\» \ x) =$
  $(\forall x. \sum y \in \text{UNIV}. \ \text{if} \ y = x \ \text{then} \ P y \ \text{else} \ 0)$
by (rule ext [OF sum.cong]) auto
also from $sP$ have ... $\vdash P$
by(cases finite (UNIV::'s set), auto simp:sum.delta)
finally have leP: $(\forall x. \sum y \in \text{UNIV}. \ P y \ast «\lambda z. \ z = y\» \ x \ \vdash P$.
moreover have sound $(\forall x. \sum y \in \text{UNIV}. \ P y \ast «\lambda z. \ z = y\» \ x)$
proof(intro introI2 bounded-byI nnegI sum-nonneg ballI)
fix $x$
from leP have $(\sum y \in \text{UNIV}. \ P y \ast «\lambda z. \ z = y\» \ x) \leq P x$ by(auto)
also from $sP$ have ... $\leq \text{bound-of} \ P$ by(auto)
finally show $(\sum y \in \text{UNIV}. \ P y \ast «\lambda z. \ z = y\» \ x) \leq \text{bound-of} \ P$.
fix $y$
from $sP$ show $0 \leq P y \ast «\lambda z. \ z = y\» \ x$
by(auto intro:mult-nonneg-nonneg)
qed
ultimately have $t (\forall x. \sum y \in \text{UNIV}. \ P y \ast «\lambda z. \ z = y\» \ x) \ s \leq t \ P \ s$
using $sP$ by(auto intro:le-funD[OF mono-transD, OF healthy-monoD, OF ht])
} finally show $?thesis$.
qed

Sub-distributivity

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

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

lemma sub-distrib12:
\[ \left( \bigwedge P . \text{sound } P \Rightarrow \lambda s . \text{t } P \ s \ominus \ 1 \vdash \ t \ (\lambda s . \text{P } s \ominus \ 1) \right) \Rightarrow \text{sub-distrib } t \]
by(auto)

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

lemma equiv-sub-distrib:
fixes t ::\( \, \forall : \, \forall \to \real \Rightarrow \forall \to \real \)
assumes eq: equiv-trans t u
and sd: sub-distrib t
shows sub-distrib u
proof
fix P::'s \Rightarrow \real
       s::'s
assume sP: sound P
with eq have u P s \ominus \ 1 = t P s \ominus \ 1 by(simp add:equiv-transD)
also from sP eq have ... \leq t (\lambda s . \text{P } s \ominus \ 1) \ s by(auto)
also from sP sd have ...
       = u (\lambda s . \text{P } s \ominus \ 1) \ s
       by(simp add:equiv-transD tminus-sound)
finally show u P s \ominus \ 1 \leq u (\lambda s . \text{P } s \ominus \ 1) \ s .
qed

Sublinearity implies sub-distributivity:

lemma sublinear-sub-distrib:
fixes t ::\( \, \forall : \, \forall \to \real \Rightarrow \forall \to \real \)
assumes slt: sublinear t
and sat: sub-add t
and ht: healthy t
shows sublinear t
proof
fix P::'s \Rightarrow \real
       s::'s
assume sP: sound P
moreover have sound (\lambda s . \text{0}) by(auto)
ultimately show t P s \ominus \ 1 \leq t (\lambda s . \text{P } s \ominus \ 1) \ s
       by(rule sublinearD[OF slt, where a=1 and b=0 and c=1, simplified])
qed

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

lemma sd-sa-sublinear:
fixes t::('s \Rightarrow \real) \Rightarrow 's \Rightarrow \real
assumes sdt: sub-distrib t
and sat: sub-add t
and ht: healthy t
shows sublinear t
proof
fix P::'s \Rightarrow \real
       Q::'s \Rightarrow \real
       s::'s
and a::\real
       b::\real
       c::\real
assume sP: sound P
       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\((auto \ intro\::sc\-sound)\)

**Chapter 3. Semantic Structures**

**CHAPTER 3. SEMANTIC STRUCTURES**

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

from \(ht \ sP \ sQ \ nna \ nmb\)
have \(a \ast t \ P \ s + b \ast t \ Q \ s = t \ (\lambda s. a \ast P \ s) \ast t \ (\lambda s. b \ast Q \ s) \ s\)
by\((simp \ add\::\ scalingD \ healthy\-scalingD)\)
also from \(saP \ sbQ \ sat\)
have \(t \ (\lambda s. a \ast P \ s) \ast s \ast t \ (\lambda s. b \ast Q \ s) \ s \leq \ t \ (\lambda s. a \ast P \ s + b \ast Q \ s) \ s\)
by\((blast)\)

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

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

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

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

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

have \(\forall s. \ (inverse \ c \ast a) \ast P \ s + (inverse \ c \ast b) \ast Q \ s = \ inverse \ c \ast (a \ast P \ s + b \ast Q \ s)\)
by\((simp \ add\::\ divide\-simps)\)
with \(sabPQ\) and \(nni\)

have \(si: \) sound \((\lambda s. (inverse \ c \ast a) \ast P \ s + (inverse \ c \ast b) \ast Q \ s)\)
by\((auto \ intro\::sc\-sound)\)

**hence** \(sim: \) sound \((\lambda s. (inverse \ c \ast a) \ast P \ s + (inverse \ c \ast b) \ast Q \ s \ominus 1)\)
by\((auto \ intro\::tminus\-sound)\)

from \(nz\)

have \(a \ast t \ P \ s + b \ast t \ Q \ s \ominus c = \)
\((c \ast inverse \ c) \ast a \ast t \ P \ s + \)
\((c \ast inverse \ c) \ast b \ast t \ Q \ s \ominus c\)
by\((simp)\)
also

have \(\ldots = c \ast (inverse \ c \ast a \ast t \ P \ s) +\)
\(c \ast (inverse \ c \ast b \ast t \ Q \ s) \ominus c\)
by\((simp \ add\::field\-simps)\)
also from \(nnc\)

have \(\ldots = c \ast (inverse \ c \ast a \ast t \ P \ s + inverse \ c \ast b \ast t \ Q \ s \ominus 1)\)
by\((simp \ add\::distrib\-left \ tminus\-left\-distrib)\)
3.2. EXPECTATION TRANSFORMERS

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

Sub-conjunctivity

definition
sub-conj :: \( (('s \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real}) \Rightarrow \text{bool} \)
where
sub-conj \( t \equiv \forall P. Q. (\text{sound } P \land \text{sound } Q) \rightarrow t \ P \\& \& t \ Q \vdash t (P \\& \& Q) \)
lemma sub-conjI\[\text{[intro]}\]:
\[ \land P \ Q \ \land \text{sound } P ; \ \text{sound } Q \ \implies \ t \ P \ \land \ t \ Q \ \vdash t \ (P \ \land \ Q) \ \implies \ \text{sub-conj } t \]
\text{unfolding } \text{sub-conj-def by(simp)}

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

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

Sublinearity implies sub-conjunctivity:

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

Sublinearity under equivalence

Sublinearity is preserved by equivalence.

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

3.2.4 Determinism

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

Additivity

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

**definition**

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

**where**

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

**lemma** additiveD:

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

by (simp add:additive-def)

**lemma** additiveI[intro]:

```
[ ∀ P Q s. [ [ sound P; sound Q ] ] ⟹ t (λs. P s + Q s) ] s =
additive t
```

unfolding additive-def by (blast)

Additivity is strictly stronger than sub-additivity.

**lemma** additive-sub-add:

```
addeative t ⟹ sub-add t
```

by (simp add:sub-addI additiveD)

The additivity property extends to finite summation.

**lemma** additive-sum:

```
fixes S::'s set
assumes additive: additive t
and healthy: healthy t
and finite: finite S
and sPz: (∀ z. sound (P z))
shows t (λx. ∑ y∈S. P y x) = (λx. ∑ y∈S. t (P y) x)
```

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

fix z::'s and T::'s set
assume finT: finite T
and IH: t (λx. ∑ y∈T. P y x) = (λx. ∑ y∈T. t (P y) x)

from additive sPz

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

by (auto intro!: sum-sound additiveD)

also from IH
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 \( ht: \text{healthy } t \)
  - and \( sP: \text{sound } P \)
- **Shows** \( t \ P \ x = (\sum y\in\text{UNIV}. \ P \ y \ast t \ «\lambda z. \ z = y» \ x) \)
- **Proof** –
  - have \( \bigwedge x. (\sum y\in\text{UNIV}. \ P \ y \ast «\lambda z. \ z = y» \ x) = (\sum y\in\text{UNIV}. \text{if } y = x \text{ then } P \ y \text{ else } 0) \)
    - by (rule sum.cong) auto
  - also have \( \bigwedge x. \ldots x = P \ x \)
    - by (simp add:sum.delta)
- **Finally**
  - have \( t \ P \ x = t (\lambda x. \sum y\in\text{UNIV}. \ P \ y \ast «\lambda z. \ z = y» \ x) \ x \)
    - by (simp)
  - also \{ \)
    - from \( sP \) have \( \bigwedge z. \text{sound } (\lambda a. \ P \ z \ast «\lambda a. \ za = z = a) \)
      - by (auto intro!:mult-sound)
    - hence \( t (\lambda x. \sum y\in\text{UNIV}. \ P \ y \ast «\lambda z. \ z = y» \ x) \ x = (\sum y\in\text{UNIV}. t (\lambda x. \ P \ y \ast «\lambda z. \ z = y» \ x) \ x) \)
      - by (subst additive-sum, simp-all add:assms)
  \}
- also from \( sP \)
  - have \( (\sum y\in\text{UNIV}. t (\lambda x. \ P \ y \ast «\lambda z. \ z = y» \ x) \ x) = (\sum y\in\text{UNIV}. P \ y \ast t «\lambda z. \ z = y» \ x) \)
    - by (subst scalingD)[OF healthy-scalingD, OF ht], auto
- **Finally show** \( t \ P \ x = (\sum y\in\text{UNIV}. P \ y \ast t «\lambda z. \ z = y» \ x) \).
- **Qed**

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

**Lemma** additive-guard-split:

- **Fixes** \( t::('s::finite ⇒ real) ⇒ 's ⇒ real \)
- **Assumes** additive; additive \( t \)
  - and \( ht: \text{healthy } t \)
  - and \( sP: \text{sound } P \)
- **Shows** \( t \ P \ x = (\sum y\in\{s. \ G \ s\}. \ P \ y \ast t «\lambda z. \ z = y» \ x) + (\sum y\in\{s. \neg G \ s\}. \ P \ y \ast t «\lambda z. \ z = y» \ x) \)
- **Proof** –
3.2. EXPECTATION TRANSFORMERS

from assms
have \( t \, P \, x = \left( \sum_y \in \text{UNIV.} \, P \, y \ast t \, \lambda z. \, z = y \circ \right) \, x \)
by (rule additive-delta-split)
also {
have \( \text{UNIV} = \{ s. \, G \, s \} \cup \{ s. \, \neg \, G \, s \} \)
by (auto)

hence \( \left( \sum_y \in \text{UNIV.} \, P \, y \ast t \, \lambda z. \, z = y \circ \right) \, x = \left( \sum_y \in \{ s. \, G \, s \} \cup \{ s. \, \neg \, G \, s \}. \, P \, y \ast t \, \lambda z. \, z = y \circ \right) \, x \)
by (simp)
}
also
have \( \left( \sum_y \in \{ s. \, G \, s \} \cup \{ s. \, \neg \, G \, s \}. \, P \, y \ast t \, \lambda z. \, z = y \circ \right) \, x = \left( \sum_y \in \{ s. \, G \, s \}. \, P \, y \ast t \, \lambda z. \, z = y \circ \right) \, x + \left( \sum_y \in \{ s. \, \neg \, G \, s \}. \, P \, y \ast t \, \lambda z. \, z = y \circ \right) \, x \)
by (auto intro; sum.union-disjoint)
finally show \( \text{thesis} \).
qed

Maximality

definition
maximal :: \((\, \prime \, a \Rightarrow \, \text{real} \, \Rightarrow \, \prime \, a \Rightarrow \, \text{real} \, \Rightarrow \, \text{bool} \, \)
where
maximal \( t \equiv \forall \, c. \, 0 \leq c \, \rightarrow \, t \, (\, \lambda - \, . \, c \, ) = (\, \lambda - \, . \, c \, ) \)

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

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

A transformer that is both additive and maximal is deterministic:

definition determ :: \((\, \prime \, a \Rightarrow \, \text{real} \, \Rightarrow \, \prime \, a \Rightarrow \, \text{real} \, \Rightarrow \, \text{bool} \, \)
where
determ \( t \equiv \text{additive} \, t \, \land \, \text{maximal} \, t \)

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

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

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

**Lemma determ-negate:**

*Assumes* _determ:_ determ _t_

*Shows* _t«P» s + t«N P» s = 1_

*Proof* –

*Have* _t«P» s + t«N P» s = t (√λ s. «P» s + «N P» s) s_

*By:* (_simp add: additiveD determ determ-additiveD_)

*Also* {  
  *Have* _∧ s. «P» s + «N P» s = 1_
  *By:* (_case-tac P s, simp-all_)
  *Hence* _t (√λ s. «P» s + «N P» s) = t (√λ s. 1)_
  *By:* (_simp_)
}

*Also have* _t (√λ s. 1) = (√λ s. 1)_

*By:* (_simp add: maximalD determ determ-maximalD_)

*Finally show* ?thesis .

qed

### 3.2.5 Modular Reasoning

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

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

**Lemma entails-combine:**

*Assumes* _wp1: P ⊢ t R_

  *and* _wp2: Q ⊢ t S_

  *and* _sc: sub-conj t_

  *and* _sR: sound R_

  *and* _sS: sound S_

*Shows* _P && Q ⊢ t (R && S)_

*Proof* –

*From* _wp1 and wp2 have* _P && Q ⊢ t R && t S_

  *By:* (_blast intro: entails-frame_)

*Also from* _sc and sR and sS have* _… ≤ t (R && S)_

  *By:* (_rule sub-conjD_)

*Finally show* ?thesis .

qed

These allow mismatched results to be composed

**Lemma entails-strengthen-post:**

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

*By:* (_blast intro: entails-trans_)


3.2. EXPECTATION TRANSFORMERS

**lemma entails-weaken-pre:**
\[
[ \ Q \vdash t \ R ; \ P \vdash Q ] \implies P \vdash t \ R
\]
by (blast intro: entails-trans)

This rule is unique to pGCL. Use it to scale the post-expectation of a rule to 'fit under' the precondition you need to satisfy.

**lemma entails-scale:**
\[
assumes wp: P \vdash t \ Q \text{ and } h: \text{healthy } t
\]
and \(sQ: \text{sound } Q \text{ and } \text{pos: } 0 \leq c\)
\[\text{shows } (\lambda s. c * P s) \vdash t (\lambda s. c * Q s)\]
\[\text{proof (rule le-funI)}\]
fix \(s\)
from \(\text{pos and wp have } c * P s \leq c * t \ Q s\)
by (auto intro: mult-left-mono)
with \(sQ \text{ pos h show } c * P s \leq t (\lambda s. c * Q s)\)
by (simp add: scalingD healthy-scalingD)
qed

3.2.6 Transforming Standard Expectations

Reasoning with standard expectations, those obtained by embedding a predicate, is often easier, as the analogues of many familiar boolean rules hold in modified form.

One may use a standard pre-expectation as an assumption:

**lemma use-premise:**
\[
assumes h: \text{healthy } t \text{ and } wP: \forall s. \ P s \implies 1 \leq t \ «Q» s
\]
\[\text{shows } «P» \vdash t «Q»\]
\[\text{proof (rule entailsI)}\]
fix \(s\)
assume \(P s\)
\[\text{hence } 1 = «P» s \text{ by (simp)}\]
also from \(wp \text{ have } ... \leq t «Q» s \text{ by (auto)}\)
finally show \(1 \leq t «Q» s\).
qed
Predicate conjunction behaves as expected:

**lemma** `conj-post`:

\[
\begin{align*}
P \vdash t \left( \lambda s . \text{healthy } t \right) & \Rightarrow \text{healthy } t
\end{align*}
\]

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

Similar to \[
\begin{align*}
\text{healthy } ?t ; \wedge s . \ ?P s \Rightarrow 1 \leq \ ?t \to \ ?Q \\
\text{to } \Rightarrow \ ?P \vdash \ ?t \to \ ?Q
\end{align*}
\], but more general.

**lemma** `entails-pconj-assumption`:

- assumes \( f : \text{feasible } t \text{ and } wP : \wedge s . \ P s \Rightarrow Q s \leq t R s \)
- and \( uQ : \text{unitary } Q \text{ and } uR : \text{unitary } R \)
- shows \( \langle P \rangle \& \& Q \vdash t R \)
- unfolding `exp-conj-def`

**proof** `rule entailsI`

- fix \( s \)
- show `\langle P \rangle \& \& Q \leq t R s`

**proof** `rule cases P s`

- case `True`
  - moreover from `uQ` have `0 \leq Q s` by `(auto)`
  - ultimately show `?thesis` by `(simp add:pconj-lone wP)`
  - next
  - case `False`
  - moreover from `uQ` have `Q s \leq 1` by `(auto)`
  - ultimately show `?thesis` using `assms` by `auto`

```
qed
```

end

3.3 Induction

**theory** `Induction`

- **imports** `Expectations Transformers`

```
begin
```

3.3.1 The Lattice of Expectations

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

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

\[
\text{definition } \text{Inf-exp} :: \ 's \ \text{expect set} \Rightarrow \ 's \ \text{expect}
\text{where } \text{Inf-exp} \ S = (\lambda s. \ \text{Inf} \ \{f \cdot s \mid f \in S\})
\]

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

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

\[
\text{definition } \text{Sup-exp} :: \ 's \ \text{expect set} \Rightarrow \ 's \ \text{expect}
\text{where } \text{Sup-exp} \ S = (if S = \{} then \lambda s. \ 0 \ else \ (\lambda s. \ \text{Sup} \ \{f \cdot s \mid f \in S\}))
\]

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

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

\[
\text{lemma } \text{Sup-exp-sound}:
\text{assumes } sS: \ \forall P. \ P \in S \Rightarrow \ \text{sound} \ P
\text{and } bS: \ \forall P. \ P \in S \Rightarrow \ \text{bounded-by} \ b \ P
\text{shows } \text{sound} \ (\text{Sup-exp} \ S)
\text{proof(cases S=\{}, simp add:Sup-exp-def, blast, intro soundI2 bounded-byI2 nnegI2)}
\text{assume neS: } S \neq \{}
\text{then obtain } P \text{ where } \text{Pin: } P \in S \text{ by(auto)}
\text{with } sS bS \text{ have } nP: \ \text{nneg} \ P \ \text{bounded-by} \ b \ P \text{ by(auto)}
\text{hence } nb: 0 \leq b \text{ by(auto)}
\text{from } bS nb \text{ show } \text{Sup-exp} \ S \vdash \ \lambda s. \ b
\text{by(auto intro:Sup-exp-least)}
\]

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

definition lfp-exp :: 's trans ⇒ 's expect
where lfp-exp t = Inf-exp {P. sound P ∧ t P ≤ P}

lemma lfp-exp-lowerbound:
[ t P ≤ P; sound P ] ⇒ lfp-exp t ≤ P
unfolding lfp-exp-def by (auto intro: Inf-exp-lower)

lemma lfp-exp-greatest:
[ [ P. t P ≤ P; sound P ] ⇒ Q ≤ P; sound Q; t R ⊢ R; sound R ] ⇒ Q ≤ lfp-exp t
unfolding lfp-exp-def by (auto intro: Inf-exp-greatest)

lemma feasible-lfp-exp-sound:
feasible t =⇒ sound (lfp-exp t)
by (intro soundI2 bounded-byI2 nnegI2, auto intro!: lfp-exp-lowerbound lfp-exp-greatest)

lemma lfp-exp-bound:
(∀P. unitary P =⇒ unitary (t P)) =⇒ bounded-by 1 (lfp-exp t)
by (auto intro!: lfp-exp-lowerbound)

lemma lfp-exp-unitary:
(∀P. unitary P =⇒ unitary (t P)) =⇒ unitary (lfp-exp t)
proof (intro unitaryI [OF lfp-exp-bound lfp-exp-bound], simp-all)
  assume IH: ∀P. unitary P =⇒ unitary (t P)
  have unitary (λs. 1) by (auto)
  with IH have unitary (t (λs. 1)) by (auto)
  thus t (λs. 1) =⇒ λs. 1 by (auto)
  show sound (λs. 1) by (auto)
qed

lemma lfp-exp-lemma2:
fixes t :: 's trans
assumes st: ∀P. sound P =⇒ sound (t P)
and mt: mono-trans t
3.3. INDUCTION

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

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

fix \( P :: \)'s expect
assume \( fP: t P \vdash P \) and \( sP: \text{sound} P \)
hence \( \text{lfp-exp} t \leq t \ (\text{lfp-exp} t) \)
by (iprover intro: \text{lfp-exp-lowerbound} lfp-exp-sound lfp-exp-lemma2 assms mono-transD [OF mt])

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

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

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

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

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

lemma \text{gfp-exp-bound}:
(\text{\( \forall P. \ \text{unitary} P \ \Rightarrow \ \text{unitary} (t P) \)) \implies \text{bounded-by} t \ (\text{gfp-exp} t)
unfolding \text{gfp-exp-def}
by (rule \text{bounded-byI2} \\text{OF} \ \text{Sup-exp-least}, \text{auto})

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

proof (intro nnegI2, simp add: gfp-exp-def, cases)
assume empty: \{ P. \ \text{unitary} P \ \land \ P \vdash t P \} = \{
show \lambda s. 0 \vdash \text{Sup-exp} \ \{ P. \ \text{unitary} P \ \land \ P \vdash t P \}
by (simp only: empty Sup-exp-def, auto)

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

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

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

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

lemma gfp-exp-unfold:
(\forall P. unitary P \implies unitary (t P)) \implies (\forall P Q. [ unitary P; unitary Q; P \vdash Q
] \implies t P \vdash t Q) \implies
gfp-exp t = t (gfp-exp t)
by (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 \text{	extlambda}. 0::'c, and the
top element is the greatest allowed by feasibility: $\lambda P \vdash \text{bound-of } P$.

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

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

lemma $\text{Inf-trans-greatest}$:
\[
\begin{align*}
\forall t \in S; \forall u \in S; \forall P. \text{sound } P \rightarrow \text{sound } (u P) & \Rightarrow \text{le-trans } u (\text{Inf-trans } S) \\
\end{align*}
\]
unfolding $\text{Inf-trans-def}$
by (auto intro: le-transI [OF Inf-exp-greatest])

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

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

lemma $\text{Sup-trans-upper2}$:
\[
\begin{align*}
\forall t \in S; \forall u \in S. \forall P. (\text{nneg } P \land \text{bounded-by b } P) \rightarrow (\text{nneg } (u P) \land \text{bounded-by b } (u P)); \land P. \text{nneg P; bounded-by b P } & \Rightarrow t P \vdash \text{Sup-trans } S P \\
\end{align*}
\]
unfolding $\text{Sup-trans-def}$
by (blast intro:Sup-exp-upper)

lemma $\text{Sup-trans-least}$:
\[
\begin{align*}
\forall u \in S. \forall P. \text{unitary } P \Rightarrow \text{unitary } (u P) & \Rightarrow \text{le-utrans } (\text{Sup-trans } S) u \\
\end{align*}
\]
unfolding $\text{Sup-trans-def}$
by (auto intro: sound-nneg [OF unitary-sound] le-utransI [OF Sup-exp-least!])

lemma $\text{Sup-trans-least2}$:
\[
\begin{align*}
\forall t \in S; \forall P. \text{nneg } P \rightarrow \text{bounded-by b } P \rightarrow t P \vdash u P; \land P. \text{nneg P; bounded-by b P } & \Rightarrow \text{nneg } (u P) \\
\Rightarrow \text{Sup-trans } S P \vdash u P \\
\end{align*}
\]
unfolding $\text{Sup-trans-def}$
by (blast intro: Sup-exp-least)

lemma $\text{feasible-Sup-trans}$:
fixes $S :: \text{'}s\ trans\ set$
assumes $\text{fs}:: \forall t \in S. \text{feasible } t$
shows $\text{feasible } (\text{Sup-trans } S)$
proof (cases $S = \{\}$, simp add: Sup-trans-def Sup-exp-def, blast, intro feasibleI)
fix $b :: \text{real}$ and $P :: \text{'}s\ expect$
assume $b P; \text{bounded-by b } P$ and $n P: \text{nneg } P$
and $ne S: S \neq \{\}$
from neS obtain t where tin: t ∈ S by(auto)
with fS have fl: feasible t by(auto)
with bP nP have λs. θ ⊢ t P by(auto)
also { from bP nP have sound P by(auto)
  with tin fS have t P ⊢ Sup-trans S P
  by(auto intro!:Sup-trans-upper2)
}
finally show nneg (Sup-trans S P) by(auto)

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

definition lfp-trans :: (′s trans ⇒ ′s trans ⇒ ′s trans)
where lfp-trans T = Inf-trans {t. (∀P. sound P → sound (t P)) ∧ le-trans (T t) t}
lemma lfp-trans-lowerbound:
[ le-trans (T t) t; ∀P. sound P → sound (t P) ] ⇒ le-trans (lfp-trans T) t
unfolding lfp-trans-def by(auto intro:Inf-trans-lower)
lemma lfp-trans-greatest:
[ ∀t P. le-trans (T t) t; ∀P. sound P → sound (t P) ] ⇒ le-trans u t;
∀P. sound P → sound (v P); le-trans (T v) v ] ⇒
le-trans u (lfp-trans T)
unfolding lfp-trans-def by(rule Inf-trans-greatest, auto)
lemma lfp-trans-sound:
fixes P Q::′s expect
assumes sP: sound P
  and fv: le-trans (T v) v
  and sv: ∀P. sound P → sound (v P)
sowsound (lfp-trans T P)
proof(intro soundI2 bounded-byI2 nnegI2)
from fv sv have le-trans (lfp-trans T) v
by(prover intro:lfp-trans-lowerbound)
with sP have lfp-trans T P ⊢ v P by(auto)
also { from sv sP have sound (v P) by(prover)
  hence v P ⊢ λs. bound-of (v P) by(auto)
}
finally show lfp-trans T P ⊢ λs. bound-of (v P).

have le-trans (λP s. 0) (lfp-trans T)
proof(intro lfp-trans-greatest)
3.3. INDUCTION

fix t ::'s trans
assume \( \forall P. \text{sound } P \iff \text{sound } (t P) \)
hence \( \forall P. \text{sound } P \iff \lambda s. 0 + t P \) 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: \( \forall P. \text{sound } P \iff \text{sound } (v P) \)
and [T]: 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)
show bounded-by 1 (\( \text{lfp-trans } T P \))
proof (rule bounded-byI2)
from [T] have le-trans (\( \text{lfp-trans } T \)) (\( \lambda P s. \text{bound-of } P \)) by (auto intro: lfp-trans-lowerbound)
with uP have \( \text{lfp-trans } T P \vdash \lambda s. \text{bound-of } P \) by (auto)
also from uP have ...
finally show \( \text{lfp-trans } T P \vdash \lambda s. 1 \).
qed
qed

lemma lfp-trans-lemma2:
fixes v ::'s trans
assumes mono: \( \forall t u. [ \text{le-trans } t u; \forall P. \text{sound } P \iff \text{sound } (t P); \forall P. \text{sound } P \iff \text{sound } (u P) ] \Rightarrow \text{le-trans } (T t) (T u) \)
and nT: \( \forall t P. [ \forall Q. \text{sound } Q \iff \text{sound } (t Q); \text{sound } P ] \Rightarrow \text{sound } (T t P) \)
and fv: le-trans (T v) v
and sv: \( \forall P. \text{sound } P \iff \text{sound } (v P) \)
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: le-trans (T t) t and st: \( \forall P. \text{sound } P \iff \text{sound } (t P) \)
hence le-trans (\( \text{lfp-trans } T \)) t by(auto intro!: lfp-trans-lowerbound)
with ft st have le-trans (T (\( \text{lfp-trans } T \))) (T t) by (iprover intro: mono lfp-trans-sound fe sv)
also note ft
finally show le-trans \( (T (\text{lfp-trans } T)) \) \( t \).

\[ \text{qed} \]

**Lemma lfp-trans-lemma3:**

fixes \( v ::'s \) trans
assumes mono: \( \forall t u. \ (\text{le-trans } t u; \ \forall P. \ \text{sound } P \Rightarrow \text{sound } (t P)) \)
and \( sT: \ \forall t P. \ (\forall Q. \ \text{sound } Q \Rightarrow \text{sound } (t Q); \ \text{sound } P) \Rightarrow \text{sound } (T t P) \)
and \( fu: \ \text{le-trans } (T v) v \)
and \( sv: \ \forall P. \ \text{sound } P \Rightarrow \text{sound } (v P) \)
shows le-trans \( (\text{lfp-trans } T) (T (\text{lfp-trans } T)) \)

**Proof** (rule lfp-trans-lowerbound)

fix \( P ::'s \) expect
have \( n1: \forall P. \ \text{sound } P \Rightarrow \text{sound } (\text{lfp-trans } T P) \)
  by (iprover intro: lfp-trans-sound fv sv)
with \( sP \) have \( n2: \ \text{sound } (\text{lfp-trans } T P) \)
  by (iprover intro: lfp-trans-sound fv sv sT)
with \( n1 sP \) show \( n3: \ \text{sound } (T (\text{lfp-trans } T P)) \)
  by (iprover intro: sT)
next
show le-trans \( (T (T (\text{lfp-trans } T))) (T (\text{lfp-trans } T)) \)
  by (rule mono[OF lfp-trans-lemma2, OF mono], (iprover intro:assms lfp-trans-sound)+)

\[ \text{qed} \]

**Lemma lfp-trans-unfold:**

fixes \( P ::'s \) expect
assumes mono: \( \forall t u. \ (\text{le-trans } t u; \ \forall P. \ \text{sound } P \Rightarrow \text{sound } (t P)); \ \forall P. \ \text{sound } P \Rightarrow \text{sound } (u P) \)
and \( sT: \ \forall t P. \ (\forall Q. \ \text{sound } Q \Rightarrow \text{sound } (t Q); \ \text{sound } P) \Rightarrow \text{sound } (T t P) \)
and \( fu: \ \text{le-trans } (T v) v \)
and \( sv: \ \forall P. \ \text{sound } P \Rightarrow \text{sound } (v P) \)
shows equiv-trans \( (\text{lfp-trans } T) (T (\text{lfp-trans } T)) \)

**Proof** (rule le-trans-antisym, rule lfp-trans-lemma2[OF mono], (iprover intro:assms)+, rule lfp-trans-lemma2[OF mono], (iprover intro:assms)+)

\[ \text{qed} \]

**Definition gfp-trans :: \('s trans \Rightarrow 's trans \Rightarrow 's trans**

**Where**

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

**Lemma gfp-trans-upperbound:**

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

**Unfolding** gfp-trans-def by(auto intro:Sup-trans-upper)
3.3. INDUCTION

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

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

lemma gfp-trans-unitary:
\[
\begin{align*}
\text{fixes } & P :: s \\
\text{assumes } & uP: \text{unitary } P \\
\text{shows } & \text{unitary } (\text{gfp-trans } T P) \\
\text{proof } & \text{(intro unitaryI2 nnegI2 bounded-byI2)}
\end{align*}
\]

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

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

fix \( t \)

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

note \( fp \)

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

moreover note \( ht \) gfp-trans-unitary

ultimately have \( \text{le-utrans} \ (T \ t) \ (T \ (\text{gfp-trans} T)) \) by (rule mono)

}

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

qed

lemma gfp-trans-lemma3:

assumes mono: \( \forall t \ u. \ \text{le-utrans} \ t \ u; \forall P. \text{unitary} \ P \implies \text{unitary} \ (t \ P);\:

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

and \( hT: \forall t \ P. \ [ \forall Q. \text{unitary} \ Q \implies \text{unitary} \ (t \ Q); \text{unitary} \ P \implies \text{unitary} \]

\( (T \ t \ P) \)

shows \( \text{le-utrans} \ (T \ (\text{gfp-trans} T)) \ (\text{gfp-trans} T) \)

by (blast intro!: mono gfp-trans-unitary gfp-trans-upperbound gfp-trans-lemma2 mono hT)

lemma gfp-trans-unfold:

assumes mono: \( \forall t \ u. \ \text{le-utrans} \ t \ u; \forall P. \text{unitary} \ P \implies \text{unitary} \ (t \ P);\:

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

and \( hT: \forall t \ P. \ [ \forall Q. \text{unitary} \ Q \implies \text{unitary} \ (t \ Q); \text{unitary} \ P \implies \text{unitary} \]

\( (T \ t \ P) \)

shows \( \text{equiv-utrans} \ (\text{gfp-trans} T) \ (T \ (\text{gfp-trans} T)) \)

using assms by (auto intro!: le-utrans-antisym gfp-trans-lemma2 gfp-trans-lemma3)

3.3.3 Tail Recursion

The least (greatest) fixed point of a tail-recursive expression on transformers

is equivalent (given appropriate side conditions) to the least (greatest) fixed

point on expectations.

lemma gfp-pulldown:

fixes \( P; \) s expect

assumes tailcall: \( \forall u \ P. \text{unitary} \ P \implies T \ u \ P = t \ P \ (u \ P) \)

and \( ft: \forall t \ P. \ [ \forall Q. \text{unitary} \ Q \implies \text{unitary} \ (t \ Q); \text{unitary} \ P \implies \text{unitary} \ (T \ t \ P) \)

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

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

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

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

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

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

proof (rule antisym)
  show \( ?X P \leq ?Y P \)
proof (rule gfp-exp-upperbound)
  from \( \text{mono}\ T \ T \ uP \) have \( (\text{gfp-trans} \ T) \ P \leq (T \ (\text{gfp-trans} \ T)) \ P \)
  by (auto intro!: le-utransD[OF gfp-trans-lemma2])
also from \( uP \) have \( (T \ (\text{gfp-trans} \ T)) \ P = tP \ (\text{gfp-trans} \ T \ P) \) by (rule tailcall)
finally show \( \text{gfp-trans} \ T \ P \vdash tP \ (\text{gfp-trans} \ T \ P) \).
from \( uP \) \( \text{gfp-trans-unitary} \) show \( \text{unitary} \ (\text{gfp-trans} \ T \ P) \) by (auto)
qed
show \( ?Y P \leq ?X P \)
proof (rule le-utransD[OF gfp-trans-upperbound], simp_all add:assms)
  show le-utrans \( (\lambda a. \ \text{gfp-exp} \ (t \ a)) \) \( (T \ (\lambda a. \ \text{gfp-exp} \ (t \ a))) \)
proof (rule le-utransI)
  fix \( Q \).
  assumes tailcall: \( \lambda uP. \ \text{sound} \ P \Rightarrow T \ uP = tP \ (uP) \)
  and st: \( \lambda P Q. \ \text{sound} \ P \Rightarrow \text{sound} \ Q \Rightarrow \text{sound} \ (tP \ Q) \)
  and mt: \( \lambda P. \ \text{sound} \ P \Rightarrow \text{mono-trans} \ (tP) \)
  and \( \text{monoT} \): \( \lambda u. \ [ \ \text{le-trans} \ t \ u ; \ \lambda P. \ \text{sound} \ P \Rightarrow \text{sound} \ (tP) ; \ \lambda P. \ \text{sound} \ P \Rightarrow \text{sound} \ (uP) ] \Rightarrow \text{le-trans} \ (T \ t) \ (T \ u) \)
  and \( \text{nT} \): \( \lambda tP. \ [ \ \lambda Q. \ \text{sound} \ Q \Rightarrow \text{sound} \ (tQ) ; \ \text{sound} \ P ] \Rightarrow \text{sound} \ (T \ t \ P) \)
  and \( \text{fv} \): \( \text{le-trans} \ (T \ v) \ v \)
  and \( \text{sv} \): \( \lambda P. \ \text{sound} \ P \Rightarrow \text{sound} \ (vP) \)
  and \( \text{sp} \): \( \text{sound} \ P \)
shows \( \text{lfp-trans} \ T \ P = \text{lfp-exp} \ (tP) \) \( \text{(is} \ ?X P = ?Y P) \)
proof (rule antisym)
  show \( ?Y P \leq ?X P \)
proof (rule lfp-exp-lowerbound)
  from \( \text{sp} \) have \( tP \ (\text{lfp-trans} \ T \ P) = (T \ (\text{lfp-trans} \ T)) \ P \) by (rule tailcall[antisym])
  also have \( (T \ (\text{lfp-trans} \ T)) \ P \leq (\text{lfp-trans} \ T \ P) \)
  by (rule le-utransD[OF lfp-trans-lemma2[OF monoT]], (iprover intro:assms)+)
finally show \( tP \ (\text{lfp-trans} \ T \ P) \leq \text{lfp-trans} \ T \ P \).
from \( \text{sp} \) show \( \text{sound} \ (\text{lfp-trans} \ T \ P) \)

lemma lfp-pulldown:
  fixes \( P \).
  assumes tailcall: \( \lambda uP. \ \text{sound} \ P \Rightarrow T \ uP = tP \ (uP) \)
  and \( \text{st} \): \( \lambda P Q. \ \text{sound} \ P \Rightarrow \text{sound} \ Q \Rightarrow \text{sound} \ (tP \ Q) \)
  and \( \text{mt} \): \( \lambda P. \ \text{sound} \ P \Rightarrow \text{mono-trans} \ (tP) \)
  and \( \text{monoT} \): \( \lambda u. \ [ \ \text{le-trans} \ t \ u ; \ \lambda P. \ \text{sound} \ P \Rightarrow \text{sound} \ (tP) ; \ \lambda P. \ \text{sound} \ P \Rightarrow \text{sound} \ (uP) ] \Rightarrow \text{le-trans} \ (T \ t) \ (T \ u) \)
  and \( \text{nT} \): \( \lambda tP. \ [ \ \lambda Q. \ \text{sound} \ Q \Rightarrow \text{sound} \ (tQ) ; \ \text{sound} \ P ] \Rightarrow \text{sound} \ (T \ t \ P) \)
  and \( \text{fv} \): \( \text{le-trans} \ (T \ v) \ v \)
  and \( \text{sv} \): \( \lambda P. \ \text{sound} \ P \Rightarrow \text{sound} \ (vP) \)
  and \( \text{sp} \): \( \text{sound} \ P \)
shows \( \text{lfp-trans} \ T \ P = \text{lfp-exp} \ (tP) \) \( \text{(is} \ ?X P = ?Y P) \)
proof (rule antisym)
  show \( ?Y P \leq ?X P \)
proof (rule lfp-exp-lowerbound)
  from \( \text{sp} \) have \( tP \ (\text{lfp-trans} \ T \ P) = (T \ (\text{lfp-trans} \ T)) \ P \) by (rule tailcall[antisym])
  also have \( (T \ (\text{lfp-trans} \ T)) \ P \leq (\text{lfp-trans} \ T \ P) \)
  by (rule le-utransD[OF lfp-trans-lemma2[OF monoT]], (iprover intro:assms)+)
finally show \( tP \ (\text{lfp-trans} \ T \ P) \leq \text{lfp-trans} \ T \ P \).
from \( \text{sp} \) show \( \text{sound} \ (\text{lfp-trans} \ T \ P) \)
by (iprover intro: lfp-trans-sound assms)
qed

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

fix \( P :: \tau \)
assume \( \text{sound } P \)
with \( \text{fvP svP} \) show \( \text{sound } (\text{lfp-exp } (t P)) \) by (simp)
qed

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

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

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

end
Chapter 4

The pGCL Language

4.1 A Shallow Embedding of pGCL in HOL

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

Skip does nothing at all.

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

Apply lifts a state transformer into the space of programs.

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

Seq is sequential composition.

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

PC is probabilistic choice between programs.

definition PC :: 's prog ⇒ ('s ⇒ real) ⇒ 's prog ⇒ 's prog
where PC a P b ≡ λab Q s. P s * a ab Q s + (1 − P s) * b ab Q s
**DC** is demonic choice between programs.

**definition** DC :: 's prog ⇒ 's prog ⇒ 's prog (- ⋃ [-58,57] 57)

**where** DC a b ≡ λab Q s. min (a ab Q s) (b ab Q s)

**AC** is angelic choice between programs.

**definition** AC :: 's prog ⇒ 's prog ⇒ 's prog (- ⋃ [-58,57] 57)

**where** AC a b ≡ λab Q s. max (a ab Q s) (b ab Q s)

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

**definition** Embed :: 's trans ⇒ 's prog

**where** Embed t = (λab. t)

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

**definition** Mu :: ('s prog ⇒ 's prog) ⇒ 's prog

**where** Mu(T) ≡ (λab. if ab then lfp-trans (λt. T (Embed t) ab) else gfp-trans (λt. T (Embed t) ab))

repeat expresses finite repetition

**primrec**

**repeat** :: nat ⇒ 'a prog ⇒ 'a prog

**where**

repeat 0 p = Skip |
repeat (Suc n) p = p ;; repeat n p

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

**definition** SetDC :: ('a ⇒ 's prog) ⇒ ('s ⇒ 'a set) ⇒ 's prog

**where** SetDC f S ≡ λab P s. Inf ((λa. f a ab P s)  ⋃ S s)

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

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

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

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

**definition** SetPC :: ('a ⇒ 's prog) ⇒ ('s ⇒ 'a ⇒ real) ⇒ 's prog

**where**

SetPC f p ≡ λab P s. ∑a∈supp (p s). p s a * f a ab P s

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

**definition**

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

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

This gives us something like let syntax

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

(- is in [55,55,55])

**translations** x is f in a => CONST Bind f (%x. a)

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

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

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

**syntax** -PBind :: pttrn => ('s => real) => 's prog => 's prog

(bind - at - in [55,55,55])

**translations** bind x at p in a => CONST SetPC (%x. a) (CONST flip (%x. p))

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

(bind - from - in [55,55,55])

**translations** bind x from S in a => CONST SetDC (%x. a) S

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

**syntax** -assign :: ident => 'a => 's prog (- := - [1000,900])

**ML**

fun assign-tr - [Const (name,-), ary] =

Const (Embedding.Apply, dummyT) $ Abs (s, dummyT, Syntax.const (suffix Record.updateN name) $ Abs (Name.uu-, dummyT, ary $ Bound 1) $ Bound 0)

| assign-tr - ts = raise TERM (assign-tr, ts)

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

**syntax** -SetPC :: ident => ('s => 'a => real) => 's prog

(choose - at - [66,66])

**ML**

fun set-pc-tr - [Const (f,-), P] =

Const (SetPC, dummyT) $ Abs (v, dummyT, (Const (Embedding.Apply, dummyT) $ Abs (s, dummyT, Syntax.const (suffix Record.updateN f) $ Abs (Name.uu-, dummyT, Bound 2) $ Bound 0))) $ P

| set-pc-tr - ts = raise TERM (set-pc-tr, ts)
parse-translation :([@[syntax-const -SetPC], set-pc-tr]);

syntax
-set-dc :: ident => ('s => 'a set) => 's prog (- :∈ - [66,66])
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.ua-, dummyT, Bound 2) $ Bound 0)))) $ S
  | set-dc-tr - ts = raise TERM (set-dc-tr, ts)
parse-translation :([@[syntax-const -set-dc], set-dc-tr]);

These definitions instantiate the embedding as either weakest precondition (True) or weakest liberal precondition (False).

syntax
-set-dc-UNIV :: ident => 's prog (any - [66])
translations
-set-dc-UNIV x => -set-dc x (%. CONST UNIV)

definition
wp :: 's prog ⇒ 's trans
where
wp pr ≡ pr True

definition
wlp :: 's prog ⇒ 's trans
where
wlp pr ≡ pr False

If-Then-Else as a degenerate probabilistic choice.

abbreviation(input)
if-then-else :: ['s ⇒ bool, 's prog, 's prog] ⇒ 's prog
  (If - Then - Else - 58)
where
If P Then a Else b == a $p a ⊕ b

Syntax for loops

abbreviation
do-while :: ['s ⇒ bool, 's prog] ⇒ 's prog
  (do - →/ (4-) //od)
where
do-while P a ≡ μ x. If P Then a ;; x Else Skip
4.1. UNFOLDING RULES FOR NON-RECURSIVE PRIMITIVES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

theory Healthiness imports Embedding begin

4.2.1 The Healthiness of the Embedding

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

lemma healthy-wp-Abort:
healthy (wp Abort)

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

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

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

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

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

assumes $ht$: nearly-healthy (wlp $t$) and $hu$: nearly-healthy (wlp $u$)
shows nearly-healthy (wlp ($t;$ $u$))
proof(rule nearly-healthyI, simp-all add:wp-eval)

fix $b$ and $P$: $'s \Rightarrow \text{real}$
assume unitary $P$
with $hu$ have unitary (wlp $u$ $P$) by(auto)
with $ht$ show unitary (wlp $t$ (wlp $u$ $P$)) by(auto)

next
fix $P$ and $Q$: $'s \Rightarrow \text{real}$
assume unitary $P$ and unitary $Q$ and $P \vdash Q$
with $hu$ have unitary (wlp $u$ $P$) and unitary (wlp $u$ $Q$)
and wlp $u$ $P \vdash \text{wlp u v}$ by(auto)
with $ht$ show wlp $t$ (wlp $u$ $P$) $\vdash$ wlp $t$ (wlp $u$ $Q$) by(auto)

qed

lemma healthy-wp-PC:
fixes $f$: $'s$ prog
assumes $hf$: healthy (wp $f$) and $hg$: healthy (wp $g$)
and $uP$: unitary $P$
shows healthy (wp ($f$ $P$ $\oplus$ $g$))
proof(intro healthy-parts bounded-byI nnegI le-funI, simp-all add:wp-eval)

fix $b$ and $Q$: $'s \Rightarrow \text{real and s:} 's$
assume $nQ$: nneg $Q$ and $bQ$: bounded-by $b$ $Q$

Non-negative:

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

Bounded:

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

next

Monotonic:

fix \( Q R ::'s \Rightarrow \text{real} \) and \( s \)
assume \( sQ \): sound \( Q \) and \( sR \): sound \( R \) and \( le :: Q \vdash R \)

with \( uP \) have \( P \ s \ast \wp f Q \ s \leq P \ s \ast \wp f R \ s \)
by \((\text{auto intro:mult-left-mono})\)

moreover \{ 
from \( sQ sR le hg \)

have \( \wp g Q \ s \leq \wp g R \ s \)
by \((\text{blast dest:mono-transD})\)
moreover from \( uP \)

have \( 0 \leq 1 - P \ s \)
by \((\text{auto simp:sign-simps})\)

ultimately have \( (1 - P \ s) \ast \wp g Q \ s \leq (1 - P \ s) \ast \wp g R \ s \)
by \((\text{auto intro:mult-left-mono})\)
\}

ultimately show \( P \ s \ast \wp f Q \ s + (1 - P \ s) \ast \wp g Q \ s \leq P \ s \ast \wp f R \ s + (1 - P \ s) \ast \wp g R \ s \)
by \((\text{auto})\)

next

Scaling:

fix \( Q ::'s \Rightarrow \text{real} \) and \( c ::\text{real} \) and \( s ::'s \)
assume \( uQ \): unitary \( Q \)
from \( uQ hf hg \)

have \( utQ ::\text{unitary} (\wp f Q \ s) \) and \( \text{unitary} (\wp g Q \ s) \)
by \((\text{auto simp:sign-simps})\)

ultimately show \( c \ast (P \ s \ast \wp f Q \ s + (1 - P \ s) \ast \wp g Q \ s) = 
P \ s \ast (c \ast \wp f Q \ s) + (1 - P \ s) \ast (c \ast \wp g Q \ s) \)
using \( hf hg sQ pos \)
by \((\text{simp add:scalingD(OF healthy-scalingD)})\)

finally show \( c \ast (P \ s \ast \wp f Q \ s + (1 - P \ s) \ast \wp g Q \ s) = 
P \ s \ast \wp f (\lambda s. c \ast Q \ s) s + (1 - P \ s) \ast \wp g (\lambda s. c \ast Q \ s) s \).

qed

lemma nearly-healthy-wlp-PC:

fixes \( f ::'s \text{ prog} \)
assumes \( hf ::\text{nearly-healthy} (\wp f) \)
and \( hg ::\text{nearly-healthy} (\wp g) \)
and \( uP ::\text{unitary} P \)
shows \( \text{nearly-healthy} (\wp f (f \oplus g)) \)
proof \((\text{intro nearly-healthyI unitaryI2 nnegI bounded-byI le-funI, simp-all add:wp-eval})\)
fix \( Q ::'s \text{ expect} \) and \( s ::'s \)
assume \( uQ ::\text{unitary} Q \)
from \( uQ hf hg \)
have \( utQ ::\text{unitary} (\wp f Q \ s) \) and \( \text{unitary} (\wp g Q \ s) \)
by \((\text{auto})\)
from \( uP \) have \( nnP :: 0 \leq P \ s \leq 1 - P \ s \)
by \((\text{auto simp:sign-simps})\)
moreover from \( utQ \)

have \( 0 \leq \wp f Q \ s \leq \wp g Q \ s \)
by \((\text{auto})\)
ultimately show \( 0 \leq P \ s \ast \wp f Q \ s + (1 - P \ s) \ast \wp g Q \ s \)
4.2. HEALTHINESS

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

from utQ have wlp f Q s ≤ 1 wlp g Q s ≤ 1 by(auto)
with nnP have P s * wlp f Q s + (1 − P s) * wlp g Q s ≤ P s * 1 + (1 − P s) * 1
  by(blast intro:add-mono mult-left-mono)
thus P s * wlp f Q s + (1 − P s) * wlp g Q s ≤ 1 by(simp)

fix R::'s expect
assume uR: unitary R and le: Q ⊢ R
with uQ have wlp f Q s ≤ wlp f R s
by(auto intro:le-funI[OF nearly-healthy-monoD, OF hf])
with nnP have P s * wlp f Q s ≤ P s * wlp f R s
by(auto intro:mult-left-mono)
moreover {
  from uQ uR le have wlp g Q s ≤ wlp g R s
  by(auto intro:le-funI[OF nearly-healthy-monoD, OF hg])
  with nnP have (1 − P s) * wlp g Q s ≤ (1 − P s) * wlp g R s
  by(auto intro:mult-left-mono)
}
ultimately show P s * wlp f Q s + (1 − P s) * wlp g Q s ≤ P s * wlp f R s + (1 − P s) * wlp g R s
  by(auto)
qed

lemma healthy-wp-DC:
fixes f::'s prog
assumes hf: healthy (wp f) and hg: healthy (wp g)
shows healthy (wp (f ⊢ g))
proof(intro healthy-parts bounded-byI nnegI le-funI simp-all only:wp-eval)
  fix b and P::'s ⇒ real and s::'s
  assume nP: nneg P and bP: bounded-by b P
  with hf have bounded-by b (wp f P) by(auto)
hence wp f P s ≤ b by(blast)
thus min (wp f P s) (wp g P s) ≤ b by(auto)
from nP bP assms show 0 ≤ min (wp f P s) (wp g P s) by(auto)
next
  fix P::'s ⇒ real and Q and s::'s
  from assms have mf: mono-trans (wp f) and mg: mono-trans (wp g) by(auto)
  assume sP: sound P and sQ: sound Q and le: P ⊢ Q
  hence wp f P s ≤ wp f Q s and wp g P s ≤ wp g Q s
    by(auto intro:le-funI[OF mono-transD[OF mf]] le-funD[OF mono-transD[OF mg]])
  thus min (wp f P s) (wp g P s) ≤ min (wp f Q s) (wp g Q s) by(auto)
next
  fix P::'s ⇒ real and c::real and s::'s
  assume sP: sound P and pos: 0 ≤ c
from \textbf{assms} have \( sf : \text{scaling} \ (wp \ f) \) and \( sg : \text{scaling} \ (wp \ g) \) \((\text{by}(\text{auto}))\)
from \textbf{pos} have \( c \ast \min \ (wp \ f \ P \ s) \ (wp \ g \ P \ s) = \)
\( \min \ (c \ast \ wp \ f \ P \ s) \ (c \ast \ wp \ g \ P \ s) \)
\((\text{by}(\text{simp add}:\text{min-distrib}))\)
also from \textbf{sP} and \textbf{pos}
have \( \ldots = \min \ (wp \ f \ (\lambda s. \ c \ast \ P \ s) \ s) \ (wp \ g \ (\lambda s. \ c \ast \ P \ s) \ s) \)
\((\text{by}(\text{simp add}:\text{scalingD}[\text{OF} \ sf] \ \text{scalingD}[\text{OF} \ sg]))\)
finally show \( c \ast \min \ (wp \ f \ P \ s) \ (wp \ g \ P \ s) = \)
\( \min \ (wp \ f \ (\lambda s. \ c \ast \ P \ s) \ s) \ (wp \ g \ (\lambda s. \ c \ast \ P \ s) \ s). \)
\((\text{qed})\)

\textbf{lemma} \textit{nearly-healthy-wlp-DC}:
\begin{align*}
\text{fixes} & \ f :: '{\text{'}s \ prog } \\
\text{assumes} & \ hf : \text{nearly-healthy} \ (wp \ f) \\
& \ \text{and} \ hg : \text{nearly-healthy} \ (wp \ g) \\
\text{shows} & \ \text{nearly-healthy} \ (wp \ (f \ \sqcap \ g)) \\
\text{proof} (\text{intro} \ \text{nearly-healthyI} \ \text{bounded-byI} \ \text{nnegI} \ \text{le-funI} \ \text{unitaryI2}, \\
& \ \text{simp-all add}:\text{wp-eval}, \text{safe}) \\
\text{fix} & \ P :: '{\text{'}s \ ⇒ \ \text{real} } \\
& \ \text{and} \ s :: '{\text{'}s \\
\text{assume} & \ uP : \text{unitary} \ P \\
& \ \text{with} \ bP \ \text{have} \ \text{atP} : \text{unitary} \ (wp \ f \ P) \ \text{unitary} \ (wp \ g \ P) \ \text{by}(\text{auto}) \\
\text{thus} & \ 0 \leq \ wp \ f \ P \ s \ 0 \leq \ wp \ g \ P \ s \ \text{by}(\text{auto}) \\
\text{have} & \ \min \ (wp \ f \ P \ s) \ (wp \ g \ P \ s) \leq \ wp \ f \ P \ s \ \text{by}(\text{auto}) \\
\text{also from} \ \text{atP} \ \text{have} \ \ldots \leq \ wp \ f \ P \ s \ \text{by}(\text{auto}) \\
\text{finally show} & \ \min \ (wp \ f \ P \ s) \ (wp \ g \ P \ s) \leq \ 1. \\
\text{fix} & \ Q :: '{\text{'}s \ ⇒ \ \text{real} \ \\
\text{assume} & \ uQ : \text{unitary} \ Q \ \text{and} \ \text{le} : \ P \ ⊆ \ Q \\
\text{have} & \ \min \ (wp \ f \ P \ s) \ (wp \ g \ P \ s) \leq \ wp \ f \ P \ s \ \text{by}(\text{auto}) \\
\text{also from} \ \text{uP} \ \text{uQ} \ \text{le} \ \text{have} \ \ldots \leq \ wp \ f \ Q \ s \\
& \ \text{by}(\text{auto intro}:\text{le-funD}[\text{OF} \ \text{nearly-healthy-monoD},\ \text{OF} \ hf]) \\
\text{finally show} & \ \min \ (wp \ f \ P \ s) \ (wp \ g \ P \ s) \leq \ wp \ f \ Q \ s. \\
\text{fix} & \ Q :: '{\text{'}s \ ⇒ \ \text{real} \ \\
\text{assume} & \ uQ : \text{unitary} \ Q \ \text{and} \ \text{le} : \ P \ ⊆ \ Q \\
\text{have} & \ \min \ (wp \ f \ P \ s) \ (wp \ g \ P \ s) \leq \ wp \ g \ P \ s \ \text{by}(\text{auto}) \\
\text{also from} \ \text{uP} \ \text{uQ} \ \text{le} \ \text{have} \ \ldots \leq \ wp \ g \ Q \ s \\
& \ \text{by}(\text{auto intro}:\text{le-funD}[\text{OF} \ \text{nearly-healthy-monoD},\ \text{OF} \ hg]) \\
\text{finally show} & \ \min \ (wp \ f \ P \ s) \ (wp \ g \ P \ s) \leq \ wp \ g \ Q \ s. \\
\text{qed} \\
\textbf{lemma} \textit{healthy-wp-AC}:
\begin{align*}
\text{fixes} & \ f :: '{\text{'}s \ prog} \\
\text{assumes} & \ hf : \text{healthy} \ (wp \ f) \ \text{and} \ hg : \text{healthy} \ (wp \ g) \\
\text{shows} & \ \text{healthy} \ (wp \ (f \ \sqcap \ g)) \\
\text{proof} (\text{intro} \ \text{healthy-parts} \ \text{bounded-byI} \ \text{nnegI} \ \text{le-funI}, \ \text{simp-all only}:\text{wp-eval}) \\
\text{fix} & \ b \ \text{and} \ P :: '{\text{'}s \ ⇒ \ \text{real} \ \text{and} \ s :: '{\text{'}s \\
\text{assume} & \ nP : \text{nneg} \ P \ \text{and} \ bP : \text{bounded-by} \ b \ P \\
& \ \text{with} \ hf \ \text{have} \ \text{bounded-by} \ b \ (wp \ f \ P) \ \text{by}(\text{auto})
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 \( \theta \leq wp \ f \ P \ s \) by(auto)
thus \( \theta \leq \max (wp \ f \ P \ s) \ (wp \ g \ P \ s) \) by(auto)

next
fix \( P ::'s \Rightarrow \real \) and \( Q \) and \( s ::'s \)
from assms have \( mf ::\ mono\-trans \ (wp \ f) \) 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 \( \max (wp \ f \ P \ s) \ (wp \ g \ P \ s) \leq \max (wp \ f \ Q \ s) \ (wp \ g \ Q \ s) \) by(auto)

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

dqed

**lemma** nearly-healthy-wlp-AC:

fixes \( f ::'s \ prog \)
assumes \( hf :: nearly\-healthy \ (wp \ f) \)
and \( hg :: nearly\-healthy \ (wp \ g) \)
shows 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 \( uP :: unitary \ P \)

with \( hf \) have \( wp \ f \ P \ s \leq 1 \) by(auto)
moreover from \( uP \ hg \) have unitary \( (wp \ g \ P) \) by(auto)
hence \( wp \ g \ P \ s \leq 1 \) by(auto)
ultimately show \( \max (wp \ f \ P \ s) \ (wp \ g \ P \ s) \leq 1 \) by(auto)

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

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

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

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

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

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

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

from nonempty obtain \( x \) where \( \xin: x \in (\lambda a. \wp (\text{prog} a) P s) \cdot S s \) by(blast)
moreover from \( sP \) and \( \text{healthy} \)
have \( \forall x:\in(\lambda a. \wp (\text{prog} a) P s) \cdot S s. 0 \leq x \) by(auto)
moreover
have \( \forall x:\in(\lambda a. \wp (\text{prog} a) Q s) \cdot S s. \exists y:\in(\lambda a. \wp (\text{prog} a) P s) \cdot S s. y \leq x \)
proof(rule ballI, clarify, rule bexI)
fix \( x \) and \( a \) assume \( \xin: a \in S s \)
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 \( \xin \) show \( \wp (\text{prog} a) P s \in (\lambda a. \wp (\text{prog} a) P s) \cdot S s \) by(simp)
qed ultimately

show \( \Inf ((\lambda a. \wp (\text{prog} a) P s) \cdot S s) \leq \Inf ((\lambda a. \wp (\text{prog} a) Q s) \cdot S s) \)
by(intro cInf-mono, blast+)
next
fix \( P::'a \Rightarrow \text{real} \) and \( c::\text{real} \) and \( s::'a \)
assume \( sP: \text{sound} P \) and \( \text{pos}: 0 \leq c \)
from nonempty obtain \( x \) where \( \xin: x \in (\lambda a. \wp (\text{prog} a) P s) \cdot S s \) by(blast)
have \( c \ast \Inf ((\lambda a. \wp (\text{prog} a) P s) \cdot S s) = \Inf ((\ast c \cdot ((\lambda a. \wp (\text{prog} a) P s) \cdot S s)) (\is U = ?V) \)
proof(rule antisym)
show \( \is U \leq ?V \)
proof(rule cInf-greatest)
from nonempty show \( (\ast c \cdot ((\lambda a. \wp (\text{prog} a) P s) \cdot S s) \neq \{} \) by(auto)
fix \( x \) assume \( x \in (\ast c \cdot ((\lambda a. \wp (\text{prog} a) P s) \cdot S s) \)
then obtain \( y \) where \( \xin: y \in (\lambda a. \wp (\text{prog} a) P s) \cdot S s \) and \( \text{rwx}: x = c \)
\ast y by(auto)

have \( \Inf ((\lambda a. \wp (\text{prog} a) P s) \cdot S s) \leq y \)
proof(intro cInf-lower[OF \( \xin \)] bdd-belowI)
fix \( z \) assume \( \zin: z \in (\lambda a. \wp (\text{prog} a) P s) \cdot S s \)
then obtain \( a \) where \( a \in S s \) and \( z = \wp (\text{prog} a) P s \) by(auto)
with \( sP \) show \( 0 \leq z \) by(auto dest:healthy)
qed

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

show \( ?V \leq ?U \)
proof(cases)
proof  
lemma nearly-healthy-wlp-SetDC  
qed

CHAPTER 4. THE PGCL LANGUAGE

(\text{x prog a P s})

also from

also have

qed

shows

nearly-healthy

finally show

proof (rule cINF-greatest)

fix x

assume x \in S s

have bdd-below ((\lambda x. c * wp (prog x) P s) \cdot S s)

proof (rule bdd-belowI [of - 0])

fix z

assume z \in (\lambda x. c * wp (prog x) P s) \cdot S s

then obtain b where b \in S s and rwz: z = c * wp (prog b) P s by auto

with sP have 0 \leq wp (prog b) P s by (auto dest: healthy)

with pos show 0 \leq z by (auto simp: rwz intro: mult-nonneg-nonneg)

qed

then have (INF x \in S s. c * wp (prog x) P s) \leq c * wp (prog x) P s

using (x \in S s) by (rule cINF-lower)

with \{c \neq 0\} show inverse c * (INF x \in S s. c * wp (prog x) P s) \leq wp (prog x) P s

by (simp add: mult-div-mono-left pos)

qed

with \{c \neq 0\} have inverse c * ?V \leq inverse c * ?U

by (simp add: mult.assoc [symmetric] image-comp)

with pos have c * (inverse c * ?V) \leq c * (inverse c * ?U)

by (auto intro: mult-left-mono)

with \{c \neq 0\} show ?thesis by (simp add: mult.assoc [symmetric])

qed

also have \ldots = Inf ((\lambda a. c * wp (prog a) P s) \cdot S s)

by (simp add: image-comp)

also from sP and pos have \ldots = Inf ((\lambda a. wp (prog a) (\lambda s. c * P s) s) \cdot S s)

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

finally show c * Inf ((\lambda a. wp (prog a) P s) \cdot S s) =

Inf ((\lambda a. wp (prog a) (\lambda s. c * P s) s) \cdot S s).

qed

lemma nearly-healthy-wlp-SetDC:

fixes prog; 'b \Rightarrow 'a prog and S; 'a \Rightarrow 'b set

assumes healthy: \forall x. x \in S s \Rightarrow nearly-healthy (wp (prog x))

and nonempty: \exists x. x \in S s

shows nearly-healthy (wp (SetDC prog S)) (is nearly-healthy ?T)

proof (intro nearly-healthyI unitaryI2 bounded-byI negI le-fund, simp-all only: wp-eval)
4.2. HEALTHINESS

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

from nonempty obtain \( x \) where \( \text{xin: } x \in (\lambda a. \text{wlp (prog a) P} s) \cdot S \) \( s \) by(blast)
moreover \{
from \( uP \) healthy
have \( \forall x \in (\lambda a. \text{wlp (prog a) P}) \cdot S \) \( s \) unitary \( x \) by(auto)
hence \( \forall x \in (\lambda a. \text{wlp (prog a) P}) \cdot S \) \( s \) \( 0 \leq x \) \( s \) by(auto)
hence \( \forall y \in (\lambda a. \text{wlp (prog a) P}) \cdot S \) \( s \) \( 0 \leq y \) by(auto)
\}
ultimately have \( \text{Inf } ((\lambda a. \text{wlp (prog a) P} s) \cdot S \) \( s \) \( \leq x \) by(intro cInf-lower
bdd-below1, auto)
also from \( \text{xin healthy uP} \) have \( x \leq 1 \) by(blast)
finally show \( \text{Inf } ((\lambda a. \text{wlp (prog a) P} s) \cdot S \) \( s \) \( \leq 1 \).

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

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

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

lemma healthy-wp-SetPC:

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

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

  show 0 ≤ \( (\sum a \in \text{supp} (p \ s). \ p \ s \ a * \text{wp} (f \ a) P s) \)
  proof(rule sum-nonneg [OF mult-nonneg-nonneg])
  fix x
  from sound show 0 ≤ p \ s \ x by(blast)
  assume x ∈ \text{supp} (p \ s) with sP and healthy
  show 0 ≤ \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: \( P \vdash Q \)
with healthy have \( \forall a. a \in \text{supp} (p \ s) \Longrightarrow \text{wp} (f \ a) P s \leq \text{wp} (f \ a) Q s \)
  by(blast)
  with sound have \( (\sum a \in \text{supp} (p \ s). \ p \ s \ a * \text{wp} (f \ a) P s) \leq (\sum a \in \text{supp} (p \ s). \ p \ s \ a * \text{wp} (f \ a) Q s) \)
    by(blast intro:sum-mono mult-left-mono)
next
fix P::'s ⇒ real and c::real and s::'s
assume sound: sound P and pos: 0 ≤ c
have c * \( (\sum a \in \text{supp} (p \ s). \ p \ s \ a * \text{wp} (f \ a) P s) = (\sum a \in \text{supp} (p \ s). \ p \ s \ a * (c * \text{wp} (f \ a) P s)) \)
  (is \( ?A = ?B \)
    by(simp add:sum-distrib-left ac-simps)
  also from sound and pos and healthy
  have \( \ldots = (\sum a \in \text{supp} (p \ s). \ p \ s \ a * \text{wp} (f \ a) (\lambda s. \ c * P s) s) \)
    by(auto simp:scalingD[OF healthy-scalingD])
  finally show \( ?A = \ldots \).
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 (wp (f a))
and sound: ∏ s, sound (p s)
and sub-dist: ∏ s, (∑ a∈supp (p s). p s a) ≤ 1
shows nearly-healthy (wp (SetPC f p)) (is nearly-healthy ?X)

Proof (intro nearly-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, a ∈ supp (p s) ⇒ unitary (wp (f a) P) by(auto)
hence ∏ a, a ∈ supp (p s) ⇒ wp (f a) P s ≤ 1 by(auto)
with sound have (∑ a∈supp (p s). p s a * wp (f a) P s) ≤ (∑ a∈supp (p s). p s a * 1)
  by(blast intro:sum-mono mult-left-mono)
also have ... = (∑ a∈supp (p s). p s a)
  by(simp add:sum-distrib-right)
also note sub-dist
finally show (∑ a∈supp (p s). p s a * wp (f a) P s) ≤ 1 .
show 0 ≤ (∑ a∈supp (p s). p s a * wp (f a) P s)
proof (rule sum-nonneg [OF mult-nonneg-nonneg])
fix x
from sound show 0 ≤ p s x by(blast)
assume x ∈ supp (p s) with uP healthy
show 0 ≤ wp (f x) P s by(blast)

qed

Next
fix P::'s expect and Q::'s expect and s
assume uP: unitary P and uQ: unitary Q and lc: P ⊢ Q
hence ∏ a, a ∈ supp (p s) ⇒ wp (f a) P s ≤ wp (f a) Q s
  by(blast intro:le-funD[OF nearly-healthy-monoD, OF healthyI])
with sound show (∑ a∈supp (p s). p s a * wp (f a) P s) ≤
  (∑ a∈supp (p s). p s a * wp (f a) Q s)
  by(blast intro:sum-mono mult-left-mono)
qed

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

Lemma nearly-healthy-wlp-Apply:
  nearly-healthy (wp (Apply f))
by(intro nearly-healthyI unitaryI2 nnegI bounded-byI, auto simp:o-def wp-eval)

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

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

4.2.2 Healthiness for Loops

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

**lemma** \( \text{wlp-loop-step-mono} \):

fixes \( t, u : \text{'s trans} \)

assumes \( mb \) : nearly-healthy (\( \text{wp body} \))

and \( \text{ht} \) : \( \forall P. \text{unitary} (t P) \implies \text{unitary} (u P) \)

and \( hu \) : \( \forall P. \text{unitary} (t P) \implies \text{unitary} (u P) \)

shows \( \text{le-utrans (wlp (body ;; Embed t \langle G \rangle \oplus \text{Skip}) \text{ wlp (body ;; Embed u \langle G \rangle \oplus \text{Skip})})} \)

\[
\text{proof (intro le-utransI le-funI, simp add: wp-eval)}
\]

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

assume \( u P, \text{ unitary} P \)

with \( \text{le have} \) \( t P \triangleright u P \) \( \text{by (auto)} \)

moreover from \( u P, \text{ht} \) have \( \text{unitary} (t P) \) \( \triangleright \text{unitary} (u P) \) \( \text{by (auto)} \)

ultimately have \( \text{wp body (t P)} \) \( s \leq \text{wp body (u P)} \) \( s \)

\[
\text{by (rule le-funD[OF nearly-healthy-monoD[OF mb]])}
\]

thus \( \langle G \rangle s * \text{wp body (t P)} \) \( s \leq \langle G \rangle s * \text{wp body (u P)} \)

\[
\text{by (auto intro:mult-left-mono)}
\]

\[\text{qed}\]

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

**lemma** \( \text{lfp-loop-fp} \):

assumes \( hb \) : healthy (\( \text{wp body} \))

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

shows \( \text{\lambda a. \text{bound-of} P} \) \( s + \langle \text{N} \rangle \langle G \rangle s * P s \vdash \text{\lambda a. bound-of} P \)

\[
\text{proof (rule le-funI)}
\]

fix \( s \)

from \( s P \) have \( \text{sound} (\lambda a. \text{bound-of} P) \) \( \text{by (auto)} \)

moreover hence bounded-by (\( \text{bound-of} P \)) (\( \lambda a. \text{bound-of} P \)) \( \text{by (auto)} \)

ultimately have bounded-by (\( \text{bound-of} P \)) (\( \text{wp body (\lambda a. \text{bound-of} P)} \))

using \( hb \) \( \text{by (auto)} \)

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

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

ultimately have \( \langle G \rangle s * \text{wp body (\lambda a. \text{bound-of} P)} \) \( s + (1 - \langle G \rangle s) * P s \leq \langle G \rangle s * \text{bound-of} P + (1 - \langle G \rangle s) * \text{bound-of} P \)

\[
\text{by (blast intro:add-mono mult-left-mono)}
\]

thus \( \langle G \rangle s * \text{wp body (\lambda a. \text{bound-of} P)} \) \( s + \langle \text{N} \rangle \langle G \rangle s * P s \leq \text{bound-of} P \)

\[
\text{by (simp add: algebra-simps negate-embed)}
\]

\[\text{qed}\]

**lemma** \( \text{lfp-loop-greatest} \):

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

**Lemma lfp-loop-sound:**
fixes \( P::\text{id } \text{s expect} \)
assumes \( \text{hb: healthy } (\text{wp body}) \)
and \( \text{sP}: \text{sound } P \)
shows \( \text{sound } (\text{lfp-exp } (\lambda Q s. \text{«}G\text{»} s \ast \text{wp body } Q s \vdash \text{«}N \text{»} G s \ast P s)) \)
using \( \text{assms} \ by(auto \ intro!:\text{lfp-exp-sound lfp-loop-fp}) \)

**Lemma wlp-loop-step-unitary:**
fixes \( t::\text{id } \text{trans} \)
assumes \( \text{hb: nearly-healthy } (\text{wlp body}) \)
and \( \text{ht: } \forall P. \text{unitary } P \implies \text{unitary } (t P) \)
and \( \text{uP: unitary } P \)
shows \( \text{unitary } (\text{wlp body } (\text{body } ; \text{ Embed } t \ « G \oplus \text{Skip} ) P) \)
**proof**(intro unitaryI2 nnegI bounded-byI, simp-all add:wp-eval)
fix \( s::\text{'s} \)
from \( \text{ht } uP \ have \ uP: \text{unitary } (t P) \ by(auto) \)
with \( \text{hb have} \ \text{unitary } (\text{wlp body } (t P)) \ by(auto) \)
hence \( 0 \leq \text{wlp body } (t P) s \ by(auto) \)
with \( uP \ show \ 0 \leq \text{«} G \text{»} s \ast \text{wlp body } (t P) s \vdash (1 - \text{«} G \text{»} s) \ast P s \)
by(auto intro!:add-nonneg-nonneg mult-nonneg-nonneg)
from \( \text{ht } uP \ have \ \text{bounded-by } 1 ) (t P) \ by(auto) \)
with \( uP \ hb \ have \ \text{bounded-by } 1 (\text{wlp body } (t P)) \ by(auto) \)
hence \( \text{wlp body } (t P) s \leq 1 \ by(auto) \)
with \( uP \ have \ \text{«}G\text{»} s \ast \text{wlp body } (t P) s \vdash (1 - \text{«}G\text{»} s) \ast P s \leq \text{«}G\text{»} s \ast 1 + (1 - \text{«}G\text{»} s) \ast 1 \)
by(blast intro!:add-nonneg mult-left-nonneg)
also have \( ... = 1 \ by(simp) \)
finally show \( \text{«}G\text{»} s \ast \text{wlp body } (t P) s \vdash (1 - \text{«}G\text{»} s) \ast P s \leq 1 \).
qed

**Lemma wp-loop-step-sound:**
fixes \( t::\text{id } \text{trans} \)
assumes \( \text{hb: healthy } (\text{wp body}) \)
and \( \text{ht: } \forall P. \text{sound } P \implies \text{sound } (t P) \)
and \( \text{sP}: \text{sound } P \)
shows \( \text{sound } (\text{wp body } (\text{body } ; \text{ Embed } t \ « G \oplus \text{Skip} ) P) \)
**proof**(intro soundI2 nnegI bounded-byI, simp-all add:wp-eval)
fix \( s::\text{'s} \)
from \( \text{ht } sP \ have \ stP: \text{sound } (t P) \ by(auto) \)
with \( \text{hb have} \ 0 \leq \text{wp body } (t P) s \ by(auto) \)
with \( sP \ show \ 0 \leq \text{«} G \text{»} s \ast \text{wp body } (t P) s \vdash (1 - \text{«} G \text{»} s) \ast P s \)
4.2. HEALTHINESS

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

from ht sP have sound (t P) by(auto)
moreover hence bounded-by (bound-of (t P)) (t P) by(auto)
ultimately have wp body (t P) s ≤ bound-of (t P) using hb by(auto)
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(auto)
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 * wp body Q s + «N G» s * P s)
(is ?X = gfp-exp (?Y P))
proof —
  let ?Z u = (body ;; Embed u « G ⊕ Skip)
  show ?thesis
proof(simp only: wp-eval, intro gfp-pulldown assms le-funI)
  fix u P
  show wlp (?Z u) P = ?Y P (u P) by(simp add: wp-eval negate-embed)
next
  fix t:.′s trans and P:.′s expect
  assume at: Q. unitary Q ⇒⇒ unitary (t Q) and uP: unitary P
  thus unitary (wlp (?Z t) P)
    by(rule wlp-loop-step-unitary[OF healthy])
next
  fix P Q:.′s expect
  assume uP: unitary P and uQ: unitary Q
  show unitary (λa. « G » a * wp body Q a + « N G » a * P a)
proof(intro unitaryI2 nnegI bounded-byI)
  fix s:.′s
  from healthy uQ
  have unitary (wp body Q) by(auto)
hence 0 ≤ wp body Q s by(auto)
with \( uP \) show \( 0 \leq \langle G \rangle s * \text{wlp body} Q s + \langle N \rangle G \rangle s * P s \)
by\( (\text{auto intro: add-nonneg-nonneg mult-nonneg-nonneg}) \)

from \( \text{healthy uQ} \) have bounded-by 1 \( \text{wlp body} Q s \) 
by\( (\text{auto}) \)

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

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

hence \( \text{wlp body} Q s \leq \text{wlp body} R s \)
by\( (\text{blast intro: le-funD[OF nearly-healthy-monoD, OF healthy]}) \)
thus \( \langle G \rangle s * \text{wlp body} Q s + \langle N \rangle G \rangle s * P s \leq \langle G \rangle s * \text{wlp body} R s + \langle N \rangle G \rangle s * P s \)
by\( (\text{auto intro: mult-left-mono}) \)
next
fix \( t \) \( u::'s \) trans
assume \( \text{le-trans} t u \)

\( \bigwedge P. \text{unitary} P \Rightarrow \text{unitary} (t P) \)
\( \bigwedge P. \text{unitary} P \Rightarrow \text{unitary} (u P) \)
thus \( \text{le-trans} (\text{wlp} (?Z t)) (\text{wlp} (?Z u)) \)
by\( (\text{blast intro!: wp-loop-step-mono[OF healthy]}) \)
qed
qed

lemma \( \text{wp-loop-sound} \):
assumes \( sP: \text{sound} P \)
and \( \text{hb: healthy (wp body)} \)
shows \( \text{sound} (\text{wp do} G \rightarrow \text{body od} P) \)
proof\( (\text{simp only: wp-eval, intro lfp-trans-sound sP}) \)
let \( ?v = \lambda P s. \langle G \rangle s * \text{wlp body} Q s + \langle N \rangle G \rangle s * P s \)
show \( \text{le-trans} (\text{wp} (\text{body} :: \text{Embed} ?v \ a G \oplus \text{Skip}))) ?v \)
by\( (\text{intro le-transI, simp add: wp-eval lfp-loop-fp[unfolded negate-embed] hb}) \)
show \( \bigwedge P. \text{sound} P \Rightarrow \text{sound} (?v P) \) by\( (\text{auto}) \)
qed

Likewise, we can rewrite strict loops.

lemma \( \text{wp-Loop1} \):
fixes \( \text{body} :: 's \) prog
assumes \( sP: \text{sound} P \)
and \( \text{healthy: healthy (wp body)} \)
shows \( \text{wp (do} G \rightarrow \text{body od} P = \text{lfp-exp} (\lambda Q s. \langle G \rangle s * \text{wp body} Q s + \langle N \rangle G \rangle s * P s) \)
(is \( ?X = \text{lfp-exp} (?Y P) \))
proof –
let ?Z u = (body ;; Embed u « G » ⊕ Skip)
show thesis
proof (simp only: wp-eval, intro lfp-pulldown assms le-funI sP mono-transI)
  fix P
  show wp (?Z u) P = ?Y P (u P) by (simp add: wp-eval negate-embed)
next
fix t:′s trans and P:′s expect
assume at: ∨Q. sound Q ⇒ 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 (?Q. « 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 Q) (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 w:′s trans
assume \( le: \text{le-trans } t \ u \)

and \( st: \forall P. \text{sound } P \implies \text{sound } (t \ P) \)

and \( sw: \forall P. \text{sound } P \implies \text{sound } (u \ P) \)

with \( \text{healthy} \) show \( \text{le-trans} (wp (?Z t)) (wp (?Z u)) \)

by (rule \( \text{wp-loop-step-mono} \))

next

from \( \text{healthy} \) show \( \text{le-trans} (wp (\lambda P. \text{s-bound-of } P)) (\lambda P. \text{s-bound-of } P) \)

by (intro \( \text{le-transI} \), simp add: \( \text{wp-eval lfp-loop-fp} \) [unfolded \( \text{negate-embed} \)])

next

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

assume \( \text{sound } P \)

thus \( \text{sound } (\lambda s. \text{s-bound-of } P) \) by (auto)

qed

qed

lemma \( \text{nearly-healthy-wlp-loop}: \)

fixes \( \text{body}::'s \text{ prog} \)

assumes \( \text{hb}: \text{nearly-healthy} (wlp \ \text{body}) \)

shows \( \text{nearly-healthy} (wlp (\text{do } G \rightarrow \text{body od})) \)

proof (intro \( \text{nearly-healthyI} \) unitaryI2 nnegI2 bounded-byI2, simp-all add: \( \text{wlp-Loop1} \) \( \text{hb} \))

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

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

let \( ?X R = \lambda Q s. « G » s \ast wlp \text{ body } Q s + « N G » s \ast R s \)

show \( \lambda s. 0 \vdash \text{gfp-exp } (?X P) \)

proof (rule \( \text{gfp-exp-upperbound} \))

with \( \text{hb have unitary } (wlp \text{ body } (\lambda s. 0)) \) by (auto)

with \( \text{uP show } \lambda s. 0 \vdash (?X P (\lambda s. 0)) \)

by (blast intro!: le-funI add-nonneg-nonneg mult-nonneg-nonneg)

qed

show \( \text{gfp-exp } (?X P) \vdash \lambda s. 1 \)

proof (rule \( \text{gfp-exp-least} \))

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

assume \( \text{unitary } Q \)

thus \( Q \vdash \lambda s. 1 \) by (auto)

qed

fix \( Q::'s \text{ expect} \) and \( uQ: \text{unitary } Q \) and \( le: P \vdash Q \)

show \( \text{gfp-exp } (?X P) \vdash \text{gfp-exp } (?X Q) \)

proof (rule \( \text{gfp-exp-least} \))

fix \( R::'s \text{ expect} \) assume \( uR: \text{unitary } R \)

assume \( fp: R \vdash ?X P R \)

also from \( le \) have \( ... \vdash ?X Q R \)

by (blast intro!: add-mono mult-left-mono le-funI)
4.2. HEALTHINESS

finally show $R \vdash \text{gfp-exp} \; (?X \; Q)$

using $uR$ by(auto intro:gfp-exp-upperbound)

next

show unitary ($\text{gfp-exp} \; (?X \; Q)$)

proof (rule gfp-exp-unitary, intro unitaryI2 nnegI bounded-byI)

fix $R$’s expect and $s$’s assume $uR$: unitary $R$

with $hb$ have $ubP$: unitary ($\text{wlp body} \; R$) by(auto)

with $uQ$ show $0 \leq \langle \; G \rangle \; s \; \text{}$ wp body $R \; s + \langle \; \mathcal{N} \; G \rangle \; s \; \text{}$ Q $s$

by(blast intro:ad-nonneg-nonneg mult-nonneg-nonneg)

from $ubP$ $uQ$ have $\text{wlp body} \; R \; s \leq 1 \; Q \; s \leq 1$ by(auto)

hence $\langle \; G \rangle \; s \; \text{}$ wp body $R \; s + \langle \; \mathcal{N} \; G \rangle \; s \; \text{}$ Q $s \leq 1$

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

thus $\langle \; G \rangle \; s \; \text{}$ wp body $R \; s + \langle \; \mathcal{N} \; G \rangle \; s \; Q \; s \leq 1$

by(simp add:negate-embed)

qed

qed

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

lemma healthy-wp-loop:

fixes body:’s prog

assumes $hb$: healthy ($\text{wp body}$)

shows healthy ($\text{wp (do G --> body od)}$)

proof –

let $?X \; P = (\lambda Q \; s. \; \langle \; G \rangle \; s \; \text{}$ wp body $Q \; s + \langle \; \mathcal{N} \; G \rangle \; s \; P \; s)$

show $?thesis$

proof (intro healthy-parts bounded-byI2 nnegI2, simp-all add:wp-Loop1 $hb$ soundI2 sound-intros)

fix $P$’s expect and $c$’s real and $s$’s

assume $sp$: sound $P$ and $nnc$: $0 \leq c$

show $c \; (\text{lfp-exp} \; (?X \; P)) \; s = \text{lfp-exp} \; (?X \; (\lambda s. \; c \; P \; s)) \; s$

proof (cases)

assume $c = 0$ thus $?thesis$

proof (simp, intro antisym)

from $hb$ have $fp$: $\lambda s. \; \langle \; G \rangle \; s \; \text{}$ wp body $\text{(\lambda - 0)} \; s \vdash \; \lambda s. \; 0$ by(simp)

hence $\text{lfp-exp} \; (\lambda P \; s. \; \langle \; G \rangle \; s \; \text{}$ wp body $P \; s) \vdash \; \lambda s. \; 0$

by(auto intro:lfp-exp-lowerbound)

thus $\text{lfp-exp} \; (\lambda P \; s. \; \langle \; G \rangle \; s \; \text{}$ wp body $P \; s) \; s \leq 0$ by(auto)

have $\lambda s. \; 0 \vdash \; \text{lfp-exp} \; (\lambda P \; s. \; \langle \; G \rangle \; s \; \text{}$ wp body $P \; s)$

by(auto intro:lfp-exp-greatest fp)

thus $0 \leq \text{lfp-exp} \; (\lambda P \; s. \; \langle \; G \rangle \; s \; \text{}$ wp body $P \; s) \; s$ by(auto)

qed

next

have onesided: $\forall P. \; c. \; c \neq 0 \implies 0 \leq c \implies \text{sound} \; P$ \implies 

$\lambda a. \; c \; \text{lfp-exp} \; (\lambda a \; b. \; \langle \; G \rangle \; b \; \text{}$ wp body $a \; b + \langle \; \mathcal{N} \; G \rangle \; b \; P \; b) \; a \equiv$

$\text{lfp-exp} \; (\lambda a \; b. \; \langle \; G \rangle \; b \; \text{}$ wp body $a \; b + \langle \; \mathcal{N} \; G \rangle \; b \; (c \; P \; b)$)
proof

- fix P::'s expect and c::real
assume cnz: c ≠ 0 and nnc; 0 ≤ c and sP: sound P
with nnc have cpos: 0 < c by(auto)
hence nnc: 0 ≤ inverse c by(auto)
show λa. c * lfp-exp (λa b. «G» b * wp body a b + «N G» b * P b) a ⊢ lfp-exp (λa b. «G» b * wp body a b + «N G» b * (c * P b))
proof (rule lfp-exp-greatest)
fix Q::'s expect
assume sQ: sound Q
and fp: λb. «G» b * wp body Q b + «N G» b * (c * P b) ⊢ Q
hence ∃s. «G» s * wp body Q s + «N G» s * (c * P s) ≤ Q s by(auto) with nnic
have ∃s. inverse c * («G» s * wp body Q s + «N G» s * (c * P s)) ≤ inverse c * Q s
by(auto intro:mult-left-mono)
hence ∃s. «G» s * (inverse c * wp body Q s) + (inverse c * c) * «N G» s * P s ≤ inverse c * Q s
by(simp add: algebra-simps)
hence ∃s. «G» s * wp body (λs. inverse c * Q s) s + «N G» s * P s ≤ inverse c * Q s
by(simp add:cnz scalingD[OF healthy-scalingD, OF hb sQ nnic])
hence λs. «G» s * wp body (λs. inverse c * Q s) s + «N G» s * P s ⊢ λs. inverse c * Q s by(rule le-funI)
moreover from nnic sQ have sound (λs. inverse c * Q s)
by(iprover intro:sound-intros)
ultimately have lfp-exp (λa b. «G» b * wp body a b + «N G» b * P b) ⊢ λs. inverse c * Q s
by(rule lfp-exp-lowerbound)
hence ∃s. lfp-exp (λa b. «G» b * wp body a b + «N G» b * P b) s ≤ inverse c * Q s
by(rule le-funD)
with nnc
have ∃s. c * lfp-exp (λa b. «G» b * wp body a b + «N G» b * P b) s ≤ c * (inverse c * Q s)
by(auto intro:mult-left-mono)
also from cnz have ∃s. ... s = Q s by(simp)
finally show λa. c * lfp-exp (λa b. «G» b * wp body a b + «N G» b * P b) a ⊢ Q
by(rule le-funI)
next
from sP have sound (λs. bound-of P) by(auto)
with hb sP have sound (lfp-exp (?X P))
by(blast intro:lfp-exp-sound lfp-loop-fp)
with nnc show sound (λs. c * lfp-exp (?X P) s)
by(auto intro!:sound-intros)
from hb sP nnc
4.2. HEALTHINESS

show \( \lambda s. (G \circ s \ast \text{wp body} (\lambda s. \text{bound-of} (\lambda s. c \ast P s))) \ast + \) 
\( \langle N, G \rangle s \ast (c \ast P s) \vdash \lambda s. \text{bound-of} (\lambda s. c \ast P s) \)
by (lprove intro: lfp-loop-fp sound-intros)

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

\[ \text{assume nzc: } c \neq 0 \]
\[ \text{show } \Theta \text{thesis (is } \theta X P c s = \theta Y P c s) \]
\[ \text{proof (rule fun-cong[where } x=s], \text{rule antisym) } \]
\[ \text{from nnc nnc show } \theta X P c \vdash \theta Y P c \text{ by (rule onesided) } \]

\[ \text{moreover with nnc have nnic: } 0 \leq \text{inverse c by (auto) } \]

\[ \text{moreover from nnc } sP \text{ have } sP: \text{ sound } (\lambda s. c \ast P s) \text{ by (auto intro!: sound-intros) } \]

\[ \text{ultimately have } \theta X (\lambda s. c \ast P s) (\text{inverse c}) \vdash \theta Y (\lambda s. c \ast P s) (\text{inverse c}) \]
by (rule onesided)

\[ \text{with nnc have } \lambda s. c \ast \theta X (\lambda s. c \ast P s) (\text{inverse c}) s \vdash \lambda s. c \ast \theta Y (\lambda s. c \ast P s) (\text{inverse c}) s \]
by (blast intro!: mult-left-mono)

\[ \text{with nnc show } \theta Y P c \vdash \theta X P c \text{ by (simp add: mult.assoc[symmetric]) } \]
qed

fix P::'s expect and b::real
assume bP: bounded-by b P and nP: nneg P
show (lfp-exp \( \lambda Q s. (G \circ s \ast \text{wp body} Q s + \langle N, G \rangle s \ast P s) \vdash \lambda s. b \))
proof (intro lfp-exp-lower-bound le-funI)
fix s::'s
from bP nP hb have bounded-by b (wp body (\( \lambda s. b \))) by (auto)

hence \( \text{wp body } (\lambda s. b) s \leq b \text{ by (auto) } \)

moreover from bP have \( P s \leq b \text{ by (auto) } \)

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

from hb bP nP show \( \lambda s. 0 \vdash \text{lfp-exp } (\lambda Q s. (G \circ s \ast \text{wp body} Q s + \langle N, G \rangle s \ast P s) \)
by (auto dest!: sound-nneg intro!: lfp-loop-greatest)

next
fix P Q::'s expect
assume \( sP \): sound \( P \) and \( sQ \): sound \( Q \) and \( le: P \vdash Q \)

show \( \text{lfp-exp} (\forall X \ P) \vdash \text{lfp-exp} (\forall X \ Q) \)

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

fix \( R \): expect

assume \( sR \): sound \( R \)

and \( fp: \lambda s. \langle G \rangle \ s \ast \text{wp body} \ R \ s + \langle N \ G \rangle \ s \ast Q \ s \vdash \ R \)

from \( le \) have \( \lambda s. \langle G \rangle \ s \ast \text{wp body} \ R \ s + \langle N \ G \rangle \ s \ast P \ s \vdash \lambda s. \langle G \rangle \ s \ast \text{wp body} \ R \ s + \langle N \ G \rangle \ s \ast Q \ s \)

by (auto intro: \text{le-funI} add-left-mono mult-left-mono)

also note \( fp \)

finally show \( \text{lfp-exp} (\lambda R \ s. \langle G \rangle \ s \ast \text{wp body} \ R \ s + \langle N \ G \rangle \ s \ast P \ s) \vdash R \)

using \( sR \) by (auto intro: \text{lfp-exp-lowerbound})

next

from \( hb \ sP \) show sound \( (\text{lfp-exp} (\lambda R \ s. \langle G \rangle \ s \ast \text{wp body} \ R \ s + \langle N \ G \rangle \ s \ast Q \ s) \)

by (rule \text{lfp-loop-sound})

from \( hb \ sQ \) show \( \lambda s. \langle G \rangle \ s \ast \text{wp body} \ (\lambda s. \text{bound-of} \ Q) \ s + \langle N \ G \rangle \ s \ast Q \ s \vdash \lambda s. \text{bound-of} \ Q \)

by (rule \text{lfp-loop-fp})

from \( sQ \) show sound \( (\lambda s. \text{bound-of} \ Q) \) by (auto)

qed

qed

qed

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

lemmas \text{healthy-intros} =

\text{healthy-wp-Abort nearly-healthy-wlp-Abort healthy-wp-Skip nearly-healthy-wlp-Skip}
\text{healthy-wp-Seq nearly-healthy-wlp-Seq healthy-wp-PC nearly-healthy-wlp-PC}
\text{healthy-wp-DC nearly-healthy-wlp-DC healthy-wp-AC nearly-healthy-wlp-AC}
\text{healthy-wp-Embed nearly-healthy-wlp-Embed healthy-wp-Apply nearly-healthy-wlp-Apply}
\text{healthy-wp-SetDC nearly-healthy-wlp-SetDC healthy-wp-SetPC nearly-healthy-wlp-SetPC}
\text{healthy-wp-Bind nearly-healthy-wlp-Bind healthy-wp-repeat nearly-healthy-wlp-repeat}
\text{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**: $bd-cts :: 's trans \Rightarrow bool$

where $bd-cts t = (\forall M. (\forall i. (M i \vdash M \text{ (Suc } i)) \land \text{ sound } (M i)) \rightarrow$

$(\exists b. \forall i. \text{ bounded-by } b (M i)) \rightarrow$

t $(\text{Sup-exp (range } M)) = \text{Sup-exp (range } (t o M))$

**Lemma**: $bd-ctsD$:

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

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

**Lemma**: $bd-ctsI$:

$(\exists b. \forall i. M i \vdash M \text{ (Suc } i)) = \Rightarrow (\forall i. \text{ sound } (M i)) \Rightarrow (\forall i. \text{ bounded-by } b (M i)) \Rightarrow$

t $(\text{Sup-exp (range } M)) = \text{Sup-exp (range } (t o M)) \Rightarrow$ bd-cts t

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

A generalised property for transformers of transformers.

**Definition**: $bd-cts-tr :: ('s trans \Rightarrow 's trans) \Rightarrow bool$

where $bd-cts-tr T = (\forall M. (\forall i. \text{ le-trans } (M i) (M \text{ (Suc } i)) \land \text{ feasible } (M i)) \rightarrow$

equiv-trans $(T (\text{Sup-trans } (M \cdot \text{ 'UNIV}))) (\text{Sup-trans } ((T o M) \cdot \text{ 'UNIV}))$

**Lemma**: $bd-cts-trD$:

\[
[ bd-cts-tr T; \forall i. \text{ le-trans } (M i) (M \text{ (Suc } i)); \forall i. \text{ feasible } (M i) ] \Rightarrow
\text{equiv-trans } (T (\text{Sup-trans } (M \cdot \text{ 'UNIV}))) (\text{Sup-trans } ((T o M) \cdot \text{ 'UNIV}))
\]

by (simp add: bd-cts-tr-def)

**Lemma**: $bd-cts-trI$:

$(\forall M. (\forall i. \text{ le-trans } (M i) (M \text{ (Suc } i))) \Rightarrow (\forall i. \text{ feasible } (M i)) \Rightarrow$

equiv-trans $(T (\text{Sup-trans } (M \cdot \text{ 'UNIV}))) (\text{Sup-trans } ((T o M) \cdot \text{ 'UNIV}))$

\Rightarrow$ bd-cts-tr T

by (simp add: bd-cts-tr-def)

4.3.1 Continuity of Primitives

**Lemma**: $cts-wp-Abort$:

bd-cts (wp (Abort::'s prog))

**Proof** –

have $X$: range $\lambda(i::\text{nat}) (s::'s). 0) = \{\lambda s. 0\}$ by (auto)

show $\neg$thesis by (intro bd-ctsI, simp add: wp-eval o-def Sup-exp-def X)

qed

**Lemma**: $cts-wp-Skip$:

bd-cts (wp Skip)

by (rule bd-ctsI, simp add: wp-def Skip-def o-def)

**Lemma**: $cts-wp-Apply$:

bd-cts (wp (Apply f))

**Proof** –
have $X: \{ M. \{ P \mid P \in \text{range } M \} = \{ P \mid P \in \text{range } (\lambda i. M i (f s)) \} \} \text{ by } (\text{auto})$

show $?\text{thesis}$ by (intro bd-ctsI ext, simp add: wp-eval o-def Sup-exp-def $X$)

qed

lemma cts-wp-Bind:
fixes $a::'a \Rightarrow 's \text{ prog}$
assumes $ca: \bigwedge s. \text{bd-cts } (\text{wp } (a (f s)))$
shows $\text{bd-cts } (\text{wp } (\text{Bind } f a))$

proof (rule bd-ctsI)
fix $M::\text{nat} \Rightarrow 's \text{ expect}$ and $c::\text{real}$
assume $\text{chain}: \bigwedge i. M i \vdash M (\text{Suc } i)$ and $sM: \bigwedge i. \text{sound } (M i)$
and $bM: \bigwedge i. \text{bounded-by } c (M i)$
with $\text{bd-ctsD}(OF ca)$

have $\bigwedge s. \text{wp } (a (f s)) (\text{Sup-exp } (\text{range } M)) = \text{Sup-exp } (\text{range } (wp (a f s))) (\text{o } M)$

by (auto)

moreover have $\bigwedge s. \{ \text{fa } s | \text{fa. fa } \in \text{range } (\lambda x. \text{wp } (a (f s))) (M x)\} = \\{ \text{fa } s | \text{fa. fa } \in \text{range } (\lambda x. \text{wp } (a (f s))) (M x) s\}$

by (auto)

ultimately show $\text{wp } (\text{Bind } f a) (\text{Sup-exp } (\text{range } M)) = \\text{Sup-exp } (\text{range } (wp (\text{Bind } f a))) (\text{o } M)$

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

qed

The first nontrivial proof. We transform the suprema into limits, and appeal to the continuity of the underlying operation (here infimum). This is typical of the remainder of the nonrecursive elements.

lemma cts-wp-DC:
fixes $a b::'a \Rightarrow 's \text{ prog}$
assumes $ca: \text{bd-cts } (\text{wp } a)$
and $cb: \text{bd-cts } (\text{wp } b)$
and $ha: \text{healthy } (\text{wp } a)$
and $hb: \text{healthy } (\text{wp } b)$
shows $\text{bd-cts } (\text{wp } (a \sqcap b))$

proof (rule bd-ctsI, rule antisym)
fix $M::\text{nat} \Rightarrow 's \text{ expect}$ and $c::\text{real}$
assume $\text{chain}: \bigwedge i. M i \vdash M (\text{Suc } i)$ and $sM: \bigwedge i. \text{sound } (M i)$
and $bM: \bigwedge i. \text{bounded-by } c (M i)$

from $\text{ha hb have } \text{hab: healthy } (\text{wp } (a \sqcap b)) \text{ by } (\text{rule healthy-intros})$
from $bM \text{ have } \text{leSup}: \bigwedge i. M i \vdash \text{Sup-exp } (\text{range } M) \text{ by } (\text{auto intro: Sup-exp-upper})$
from $sM bM \text{ have } \text{sSup: sound } (\text{Sup-exp } (\text{range } M)) \text{ by } (\text{auto intro: Sup-exp-sound})$

show $\text{Sup-exp } (\text{range } (wp (a \sqcap b))) \vdash wp (a \sqcap b) (\text{Sup-exp } (\text{range } M))$

proof (rule Sup-exp-least, clarsimp, rule le-funI)

fix $i s$

from $\text{mono-transD}(OF \text{healthy-monoD}[OF \text{hab}]) \text{ leSup } sM \text{ sSup}$

have $\text{wp } (a \sqcap b) (M i) \vdash \text{wp } (a \sqcap b) (\text{Sup-exp } (\text{range } M)) \text{ by } (\text{auto})$
4.3. CONTINUITY

thus \(wp\) (\(a \coprod b\) \(\langle M i \rangle\)) \(s \leq wp\) (\(a \coprod b\) \(\langle Sup-exp (range M) \rangle\)) \(s\) by(auto)

from \(hab\) \(Sup\) have sound (\(wp\) (\(a \coprod b\) \(\langle Sup-exp (range M) \rangle\))) by(auto)
thus \(nveq\) (\(wp\) (\(a \coprod b\) \(\langle Sup-exp (range M) \rangle\))) by(auto)
qed

from \(sM bM\) ha have \(\forall i.\) bounded-by \(c\) (\(wp\) a \(\langle M i \rangle\)) by(auto)
hence baM: \(\forall i\). \(wp\) a (\(M i\)) \(s \leq c\) by(auto)
from \(sM bM\) hb have \(\forall i.\) bounded-by \(c\) (\(wp\) b (\(M i\))) by(auto)
hence bbM: \(\forall i\). \(wp\) b (\(M i\)) \(s \leq c\) by(auto)
show \(wp\) (\(a \coprod b\) \(\langle Sup-exp (range M) \rangle\)) \(\vdash\) \(Sup-exp\) (\(range\) \(wp\) (\(a \coprod b\) \(\circ M\)))
proof(simp add:wp-eval o-def, rule le-funI)
fix \(s\):\(\langle s\rangle\)
from \(bd-ctsD[OF ca, of M, OF chain\ sM bM]\ bd-ctsD[OF cb, of M, OF chain\ sM bM]
have \(\min\) \(\langle wp\ a \langle Sup-exp (range M) \rangle\rangle\) \(s\) \(\langle wp\ b \langle Sup-exp (range M) \rangle\rangle\) \(s\) = \(\min\) \(\langle Sup-exp (\langle wp\ a\ o M\rangle)\rangle\) \(s\) \(\langle Sup-exp (\langle wp\ b\ o M\rangle)\rangle\) \(s\)
by(simp)
also {
  have \(\{f s. f \in range\ \langle \lambda x.\ wp\ a\ (M x)\rangle\}\) = \(\langle range\ \lambda i.\ wp\ a\ (M i)\rangle\) \(s\)
  \(\{f s. f \in range\ \langle \lambda x.\ wp\ b\ (M x)\rangle\}\) = \(\langle range\ \lambda i.\ wp\ b\ (M i)\rangle\) \(s\)
  by(auto)
  hence \(\min\) \(\langle Sup-exp (\langle wp\ a\ o M\rangle)\rangle\) \(s\) \(\langle Sup-exp (\langle wp\ b\ o M\rangle)\rangle\) \(s\) =
    \(\min\) \(\langle Sup\ (\langle \lambda i.\ wp\ a\ (M i)\rangle)\rangle\) \(\langle Sup\ (\langle \lambda i.\ wp\ b\ (M i)\rangle)\rangle\)
by(simp add:Sup-exp-def o-def)
}
also {
  have \(\langle \lambda i.\ wp\ a\ (M i)\rangle\) \(\longrightarrow\) \(Sup\ (\langle range\ \lambda i.\ wp\ a\ (M i)\rangle)\)
proof(rule increasing-LIMSEQ)
    fix \(n\)
    from \(mono-transD[OF healthy-monoD, OF ha]\ sM\ chain\ show\ wp\ a\ (M n)\ s \leq wp\ a\ (M\ (Suc\ n))\ s\ by(auto\ intro;le-funD)
    from baM show wp a (\(M n\)) \(s \leq Sup\ (\langle range\ \lambda i.\ wp\ a\ (M i)\rangle)\)
by(intro cSup-upper bdd-aboveI, auto)
    fix \(c\):real assume pe: \(0 < e\)
    from baM have cSup: \(Sup\ (\langle range\ \lambda i.\ wp\ a\ (M i)\rangle)\) \(\subseteq\) closure \((\langle range\ \lambda i.\ wp\ a\ (M i)\rangle)\)
by(blast intro;closure-contains-Sup)
    with pe obtain y where yin: \(y \in\) \(\langle range\ \lambda i.\ wp\ a\ (M i)\rangle\)
        and dy: dist y (\(\langle Sup\ (\langle range\ \lambda i.\ wp\ a\ (M i)\rangle)\rangle\)) < \(e\)
by(blast dest:iffD1[OF closure-approachable])
    from yin obtain i where y = wp a (\(M i\)) \(s\) by(auto)
    with dy have dist (wp a (\(M i\)) \(s\)) (\(\langle Sup\ (\langle \lambda i.\ wp\ a\ (M i)\rangle)\rangle\)) < \(e\)
    by(simp)
    moreover from baM have wp a (\(M i\)) \(s \leq Sup\ (\langle range\ \lambda i.\ wp\ a\ (M i)\rangle)\)
by(intro cSup-upper bdd-aboveI, auto)
ultimately have \( \text{Sup} \left( \text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s) \right) \leq \text{wp} \ a \ (M \ i) \ s + e \)
by(simp add:dist-real-def)
thus \( \exists i. \ \text{Sup} \left( \text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s) \right) \leq \text{wp} \ a \ (M \ i) \ s + e \) by(auto)
qed

moreover
have \( (\lambda i. \ \text{wp} \ b \ (M \ i) \ s) \longrightarrow \text{Sup} \left( \text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s) \right) \)
proof(rule increasing-LIMSEQ)
fix \( n \)
from mono-transD[OF healthy-monoD, OF \( h b \)] \( s M \) chain
show \( \text{wp} \ b \ (M \ n) \ s \leq \text{wp} \ b \ (M \ (\text{Suc} \ n)) \ s \) by(auto intro:le-funD)
from \( b b M \) show \( \text{wp} \ b \ (M \ n) \ s \leq \text{Sup} \left( \text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s) \right) \)
by(intro cSup-upper bdd-aboveI, auto)

fix \( c :: \text{real} \) assume \( pe : 0 < e \)
from \( b b M \) have \( \text{cSup} : \text{Sup} \left( \text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s) \right) \in \text{closure} \left( \text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s) \right) \)
by(blast intro:closure-contains-Sup)
with \( pe \) obtain \( y \) where \( \text{yin} : y \in \ (\text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s)) \)
and \( dy : \text{dist} \ y \ (\text{Sup} \left( \text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s) \right)) < e \)
by(blast dest:iffD[OF closure-approachable])
from \( \text{yin} \) obtain \( i \) where \( y = \text{wp} \ b \ (M \ i) \ s \) by(auto)
with \( dy \) have \( \text{dist} \ (\text{wp} \ b \ (M \ i) \ s) \ (\text{Sup} \left( \text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s) \right)) < e \)
by(simp)
moreover from \( b b M \) have \( \text{wp} \ b \ (M \ i) \ s \leq \text{Sup} \left( \text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s) \right) \)
by(intro cSup-upper bdd-aboveI, auto)
ultimately have \( \text{Sup} \left( \text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s) \right) \leq \text{wp} \ b \ (M \ i) \ s + e \)
by(simp add:dist-real-def)
thus \( \exists i. \ \text{Sup} \left( \text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s) \right) \leq \text{wp} \ b \ (M \ i) \ s + e \) by(auto)
qed
ultimately have \( (\lambda i. \ \text{min} \ (\text{wp} \ a \ (M \ i) \ s) \ (\text{wp} \ b \ (M \ i) \ s)) \longrightarrow \text{min} \ (\text{Sup} \left( \text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s) \right)) \ (\text{Sup} \left( \text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s) \right)) \)
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 bdd-aboveI, clarsimp)
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 { from \( h a s M b M \) have bounded-by \( c \ (\text{wp} \ a \ (M \ i)) \) 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 \( \text{min} \ (\text{Sup} \left( \text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s) \right)) \ (\text{Sup} \left( \text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s) \right)) \)
\leq \( \text{Sup} \left( \text{range} \ (\lambda i. \ \text{min} \ (\text{wp} \ a \ (M \ i) \ s) \ (\text{wp} \ b \ (M \ i) \ s)) \right) \)
by(blast intro:LIMSEQ-le-const2 cSup-upper min.mono[OF baM bbM])
}
also \{
  \textbf{have} \ \text{range} (λi. \text{min} (wp \ a \ (M \ i) \ s) (wp \ b \ (M \ i) \ s)) =
  \{f\ s \mid f \in \text{range} (λi \ s. \text{min} (wp \ a \ (M \ i) \ s) (wp \ b \ (M \ i) \ s))\}
  \text{by} (auto)
  \textbf{hence} \ \text{Sup} (\text{range} (λi. \text{min} (wp \ a \ (M \ i) \ s) (wp \ b \ (M \ i) \ s))) =
  \text{Sup-exp} (\text{range} (λi \ s. \text{min} (wp \ a \ (M \ i) \ s) (wp \ b \ (M \ i) \ s))) \text{ s}
  \text{by} (simp \ add: \text{-def} \ cong \ del: \text{-cong-simp})
\}

finally \textbf{show} \ \text{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} (λi \ s. \text{min} (wp \ a \ (M \ i) \ s) (wp \ b \ (M \ i) \ s))) \ s .
\text{qed}
\text{qed}

\textbf{lemma} \ text{-wp-Seq:}
\text{fixes} \ a, b :: 's prog
\text{assumes} ca: \ text{-cts} (wp \ a)
  \text{and} \ cb: \ text{-cts} (wp \ b)
  \text{and} \ hb: \ \text{healthy} (wp \ b)
\text{shows} \ \text{-cts} (wp \ (a ;; b))
\text{proof} (\text{rule} \ text{-ctsI}, \text{simp} \ add: \text{-o-def} \ \text{wp-eval})
  \text{fix} M :: \text{nat} \Rightarrow 's expect
  \text{and} \ c :: \text{real}
  \text{assume} chain: \ \bigwedge i. \ M \ i \vdash \ Suc \ i \ \text{and} \ sM: \ \bigwedge i. \ \text{sound} (M \ i)
  \text{and} \ bM: \ \bigwedge i. \ \text{bounded-by} \ c \ (M \ i)
  \text{hence} \ wp \ a \ (wp \ b \ (\text{Sup-exp} (\text{range} \ M))) = wp \ a \ (\text{Sup-exp} (wp \ b \ M))
  \text{by} (\text{subst} \ \text{bd-ctsD}[OF \ cb], \text{auto})
  \text{also} \ \textbf{have} \ \text{Sup-exp} (\text{range} (\text{wp} \ a \ o \ (wp \ b \ M))) =
  \text{Sup-exp} (\text{range} (\text{wp} \ a \ (wp \ b \ M)))
  \text{by} (\text{subst} \ \text{bd-ctsD}[OF \ ca], \text{auto})
\}

also \textbf{have} \ \text{Sup-exp} (\text{range} (\text{wp} \ a \ o \ (wp \ b \ M))) =
  \text{Sup-exp} (\text{range} (\text{wp} \ a \ (wp \ b \ M)))
  \text{by} (simp \ add:o-def)

finally \textbf{show} \ \text{wp} \ a \ (wp \ b \ (\text{Sup-exp} (\text{range} \ M))) =
  \text{Sup-exp} (\text{range} (\text{wp} \ a \ (wp \ b \ M))) .
\text{qed}

\textbf{lemma} \ text{-wp-PC:}
\text{fixes} \ a, b :: 's prog
\text{assumes} ca: \ text{-cts} (wp \ a)
  \text{and} \ cb: \ text{-cts} (wp \ b)
  \text{and} \ ha: \ \text{healthy} (wp \ a)
  \text{and} \ hb: \ \text{healthy} (wp \ b)
and \( \text{wp}: \text{unitary} \ p \)
shows \( \text{bd-cts} \ (\text{wp} \ (\text{PC} \ a \ p \ b)) \)
proof(rule bd-ctsI, rule ext, simp add:o-def wp-eval)
fix \( M :: \text{nat} \Rightarrow \text{'s expect and c::real and s':s} \)
assume chain: \( \bigwedge i. \ M \ i \vdash M \ (\text{Suc} \ i) \) \( \text{and} \ bM: \bigwedge i. \text{bounded-by} \ c \ (M \ i) \)

from \( sM \) have \( \bigwedge i. \text{neg} \ (M \ i) \) by(auto)
with \( bM \) have \( \nc\ 0 \leq c \) by(auto)
from chain \( sM \ bM \) have \( \text{wp} \ a \ (\text{Sup-exp} \ (\text{range} \ M)) = \text{Sup-exp} \ (\text{range} \ (\text{wp} \ a \ (M \ i) \ i)) \)
by(rule bd-ctsD[OF ca])
hence \( \text{wp} \ a \ (\text{Sup-exp} \ (\text{range} \ M)) \ s = \text{Sup-exp} \ (\text{range} \ (\text{wp} \ a \ (M \ i) \ i)) \)
by(simp)
also { have \( \{ f \ s | f. \ f \in \text{range} \ (\lambda x. \ \text{wp} \ a \ (M \ x)) \} = \text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s) \)
by(auto)
hence \( \text{Sup-exp} \ (\text{range} \ (\text{wp} \ a \ (M \ i) \ i)) \ s = \text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ i)) \)
by(simp add:Sup-exp-def o-def)
}
finally have \( p \ s \ast \text{wp} \ a \ (\text{Sup-exp} \ (\text{range} \ M)) \ s = \)
\( p \ s \ast \text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s)) \) by(simp)
also have \( ... = \text{Sup} \ \{ p \ s \ast x | x \in \text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ s) \} \)
proof(rule cSup-mult, blast, clarsimp)
from \( \text{wp} \) show \( \theta \leq p \ s \) by(auto)
fix \( i \)
from \( sM \ bM \) ha have \( \text{bounded-by} \ c \ (\text{wp} \ a \ (M \ i)) \) by(auto)
thus \( \text{wp} \ a \ (M \ i) \ s \leq c \) by(auto)
qed
also { have \( \{ p \ s \ast x | x \in \text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ i) \} = \text{range} \ (\lambda i. \ p \ s \ast \text{wp} \ a \ (M \ i) \ i) \)
by(auto)
hence \( \text{Sup} \ \{ p \ s \ast x | x \in \text{range} \ (\lambda i. \ \text{wp} \ a \ (M \ i) \ i) \} = \)
\( \text{Sup} \ (\text{range} \ (\lambda i. \ p \ s \ast \text{wp} \ a \ (M \ i) \ i)) \) by(simp)
}
finally have \( p \ s \ast \text{wp} \ a \ (\text{Sup-exp} \ (\text{range} \ M)) \ s = \text{Sup} \ (\text{range} \ (\lambda i. \ p \ s \ast \text{wp} \ a \ (M \ i) \ i)) \)

moreover { from chain \( sM \ bM \) have \( \text{wp} \ b \ (\text{Sup-exp} \ (\text{range} \ M)) = \text{Sup-exp} \ (\text{range} \ (\text{wp} \ b \ o \ M)) \)
by(rule bd-ctsD[OF cb])
hence \( \text{wp} \ b \ (\text{Sup-exp} \ (\text{range} \ M)) \ s = \text{Sup-exp} \ (\text{range} \ (\text{wp} \ b \ o \ M)) \)
by(simp)
also { have \( \{ f \ s | f. \ f \in \text{range} \ (\lambda x. \ \text{wp} \ b \ (M \ x)) \} = \text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s) \)
by(auto)
hence \( \text{Sup-exp} \ (\text{range} \ (\text{wp} \ b \ o \ M)) \ s = \text{Sup} \ (\text{range} \ (\lambda i. \ \text{wp} \ b \ (M \ i) \ s)) \)
4.3. CONTINUITY

by(simp add:Sup-exp-def o-def)

finally have \((1 - p\ s) * wp\ b\ \text{(Sup-exp\ (range\ M)})\ s = \)
\((1 - p\ s) * \text{Sup\ (range\ (\lambda i.\ wp\ b\ (M\ i)\ s))}\ \text{by(simp)}\)
also have \(\ldots = \text{Sup}\ \{ (1 - p\ s) * x | x \in\ \text{range\ (\lambda i.\ wp\ b\ (M\ i)\ s)} \}\)
proof(rule cSup-mul, blast, clarsimp)
  from up show \(0 \leq 1 - p\ s\ \text{by(auto simp:sign-simps)}\)
  fix \(i\)
  from \(sM\ bM\ \text{have bounded-by c (wp\ b\ (M\ i))}\ \text{by(auto)}\)
  thus \(wp\ b\ (M\ i)\ s \leq c\ \text{by(auto)}\)
qed
also \{
  have \(\{(1 - p\ s) * x | x \in\ \text{range\ (\lambda i.\ wp\ b\ (M\ i)\ s)} \}\ = \)
  \(\text{range\ (\lambda i.\ (1 - p\ s) * wp\ b\ (M\ i)\ s)}\ \text{by(auto)}\)
  hence \(\text{Sup}\ \{ (1 - p\ s) * x | x \in\ \text{range\ (\lambda i.\ wp\ b\ (M\ i)\ s)} \}\ = \)
  \(\text{Sup\ (range\ (\lambda i.\ (1 - p\ s) * wp\ b\ (M\ i)\ s))}\ \text{by(simp)}\)
\}
finally have \((1 - p\ s) * wp\ b\ \text{(Sup-exp\ (range\ M))}\ s = \)
\(\text{Sup\ (range\ (\lambda i.\ (1 - p\ s) * wp\ b\ (M\ i)\ s))}\).

ultimately
have \(p\ s * wp\ a\ \text{(Sup-exp\ (range\ M))}\ s + (1 - p\ s) * wp\ b\ \text{(Sup-exp\ (range\ M))}\)
\(s = \)
\(\text{Sup\ (range\ (\lambda i.\ p\ s * wp\ a\ (M\ i)\ s))} + \text{Sup\ (range\ (\lambda i.\ (1 - p\ s) * wp\ b\ (M\ i)\ s))}\)
\(\text{by(simp)}\)
also \{
  from \(bM\ sM\ \text{have } \lambda i.\ \text{bounded-by c (wp\ a\ (M\ i))}\ \text{by(auto)}\)
  hence \(\lambda i.\ wp\ a\ (M\ i)\ s \leq c\ \text{by(auto)}\)
  moreover from up have \(0 \leq p\ s\ \text{by(auto)}\)
  ultimately have \(\lambda i.\ p\ s * wp\ a\ (M\ i)\ s \leq p\ s * c\ \text{by(auto intro:mult-left-mono)}\)
  also from up nc have \(p\ s * c \leq 1 * c\ \text{by(blast intro:mult-right-mono)}\)
  also have \(\ldots = c\ \text{by(simp)}\)
  finally have \(baM: \lambda i.\ p\ s * wp\ a\ (M\ i)\ s \leq c\).

have \(\text{lma: (\lambda i.\ p\ s * wp\ a\ (M\ i)\ s)} \longrightarrow \text{Sup\ (range\ (\lambda i.\ p\ s * wp\ a\ (M\ i)\ s))}\)
proof(rule increasing-LIMSEQ)
  fix \(n\)
  from \(sM\ \text{chain healthy-monoD[OF ha] have wp\ a\ (M\ n) \vdash wp\ a\ (M\ (Suc\ n))}\)
  \(\text{by(auto)}\)
  with up show \(p\ s * wp\ a\ (M\ n)\ s \leq p\ s * wp\ a\ (M\ (Suc\ n))\ s\)
  \(\text{by(blast intro:mult-left-mono)}\)
  from \(baM\ \text{show p\ s * wp\ a\ (M\ n)\ s \leq Sup\ (range\ (\lambda i.\ p\ s * wp\ a\ (M\ i)\ s))}\)
  \(\text{by(intro cSup-upper bdd-aboveI, auto)}\)
next
  fix \(e: real\)
  assume \(pe: \theta < e\)
from $baM$ have $\sup\left(\text{range}\left(\lambda\cdot i.\ p\ s\ \ast\ wp\ a\ (M\ i\ s)\right)\right)\in\text{closure}\left(\text{range}\left(\lambda\cdot i.\ p\ s\ \ast\ wp\ a\ (M\ i\ s)\right)\right)$
  by(blast intro:closure-contains-Sup)

thm closure-approachable

with $pe$ obtain $y$ where $\text{yin}\cdot y\in\text{range}\left(\lambda\cdot i.\ p\ s\ \ast\ wp\ a\ (M\ i\ s)\right)$
  and $dy\cdot \text{dist}\ y\left(\sup\left(\text{range}\left(\lambda\cdot i.\ p\ s\ \ast\ wp\ a\ (M\ i\ s)\right)\right)\right)<e$
  by(blast dest:iffD1[OF closure-approachable])

from $\text{yin}$ obtain $i$ where $y = p\ s\ \ast\ wp\ a\ (M\ i\ s)$
  by(auto)

with $dy$ have $\text{dist}\ (p\ s\ \ast\ wp\ a\ (M\ i\ s))\left(\sup\left(\text{range}\left(\lambda\cdot i.\ p\ s\ \ast\ wp\ a\ (M\ i\ s)\right)\right)\right)<e$
  by(simp)

moreover from $baM$ have $p\ s\ \ast\ wp\ a\ (M\ i\ s)\leq\sup\left(\text{range}\left(\lambda\cdot i.\ p\ s\ \ast\ wp\ a\ (M\ i\ s)\right)\right)$
  by(intro cSup-upper bdd-aboveI, auto)

ultimately have $\sup\left(\text{range}\left(\lambda\cdot i.\ p\ s\ \ast\ wp\ a\ (M\ i\ s)\right)\right)\leq p\ s\ \ast\ wp\ a\ (M\ i\ s) +e$
  by(simp add:dist-real-def)

thus $\exists i.\ \sup\left(\text{range}\left(\lambda\cdot i.\ p\ s\ \ast\ wp\ a\ (M\ i\ s)\right)\right)\leq p\ s\ \ast\ wp\ a\ (M\ i\ s) +e$
  by(auto)

qed

from $bM\ \text{sM}\ \text{hb}$ have $\bigwedge i.\ \text{bounded-by}\ c\ (wp\ b\ (M\ i\ s))$
  by(auto)

hence $\bigwedge i.\ wp\ b\ (M\ i\ s)\leq c$ by(auto)

moreover from $wp$ have $0\leq(1 - p\ s)$ by(auto simp:sign-simps)

ultimately have $\bigwedge i.\ (1 - p\ s)\ \ast\ wp\ b\ (M\ i\ s)\leq(1 - p\ s)\ \ast\ c$ by(auto intro:mult-left-mono)

also {
  from $wp$ have $1 - p\ s\leq1$ by(auto)
  with $nc$ have $(1 - p\ s)\ \ast\ c\leq1\ \ast\ c$ by(blast intro:mult-right-mono)
}

also have $1\ \ast\ c = c$ by(simp)

finally have $bbM:\ \bigwedge i.\ (1 - p\ s)\ \ast\ wp\ b\ (M\ i\ s)\leq c$.

have $\text{linb}\cdot (\lambda\cdot i.\ (1 - p\ s)\ \ast\ wp\ b\ (M\ i\ s))\longrightarrow\sup\left(\text{range}\left(\lambda\cdot i.\ (1 - p\ s)\ \ast\ wp\ b\ (M\ i\ s)\right)\right)$

proof(rule increasing-LIMSEQ)

fix $n$

from $sM\ \text{chain}\ \text{healthy-mono}\ D[\text{OF}\ \text{hb}]$ have $wp\ b\ (M\ n)\ \rightarrow\ wp\ b\ (M\ (\text{Suc}\ n))$
  by(auto)

moreover from $wp$ have $0\leq1 - p\ s$ by(auto simp:sign-simps)

ultimately show $(1 - p\ s)\ \ast\ wp\ b\ (M\ n)\leq(1 - p\ s)\ \ast\ wp\ b\ (M\ (\text{Suc}\ n))$
  by(blast intro:mult-left-mono)

from $bbM$ show $(1 - p\ s)\ \ast\ wp\ b\ (M\ n)\leq\sup\left(\text{range}\left(\lambda\cdot i.\ (1 - p\ s)\ \ast\ wp\ b\ (M\ i\ s)\right)\right)$
  by(intro cSup-upper bdd-aboveI, auto)

next

fix $e::\text{real}$

assume $pe:0 < e$
from $bbM$ have $\text{Sup} (\text{range} (\lambda x_i. (1 - p s) * \text{wp} b (M i) s)) \in$
\text{closure} (\text{range} (\lambda x_i. (1 - p s) * \text{wp} b (M i) s))
by(blast intro:closure-contains-Sup)
with $p e$ obtain $y$ where $yin: y \in \text{range} (\lambda x_i. (1 - p s) * \text{wp} b (M i) s)$
and $dy: \text{dist} y (\text{Sup} (\text{range} (\lambda x_i. (1 - p s) * \text{wp} b (M i) s))) < e$
by(blast dest:iffD1[OF closure-approachable])
from $yin$ obtain $i$ where $y = (1 - p s) * \text{wp} b (M i) s$ by(auto)
with $dy$ have $\text{dist} ((1 - p s) * \text{wp} b (M i) s)$
$(\text{Sup} (\text{range} (\lambda x_i. (1 - p s) * \text{wp} b (M i) s))) < e$
by(simp)

moreover from $bbM$
\text{have} $(1 - p s) * \text{wp} b (M i) s \leq \text{Sup} (\text{range} (\lambda x_i. (1 - p s) * \text{wp} b (M i) s))$
by(intro cSup-upper bdd-aboveI, auto)
ultimately have $\text{Sup} (\text{range} (\lambda x_i. (1 - p s) * \text{wp} b (M i) s)) \leq (1 - p s) * \text{wp} b (M i) s + e$
by(simp add:dist-real-def)
thus $\exists i. \text{Sup} (\text{range} (\lambda x_i. (1 - p s) * \text{wp} b (M i) s)) \leq (1 - p s) * \text{wp} b (M i) s + e$ by(auto)
qed

from $\text{lima limb}$ have $(\lambda i. p s * \text{wp} a (M i) s + (1 - p s) * \text{wp} b (M i) s)$

$\rightarrow$
$\text{Sup} (\text{range} (\lambda i. p s * \text{wp} a (M i) s)) + \text{Sup} (\text{range} (\lambda i. (1 - p s) * \text{wp} b (M i) s))$
by(rule tendsto-add)

moreover from $\text{add-mono}[OF baM bbM]$
\text{have} $\forall i. p s * \text{wp} a (M i) s + (1 - p s) * \text{wp} b (M i) s \leq$
$\text{Sup} (\text{range} (\lambda i. p s * \text{wp} a (M i) s + (1 - p s) * \text{wp} b (M i) s))$
by(auto)
ultimately have $\text{Sup} (\text{range} (\lambda i. p s * \text{wp} a (M i) s)) +$
$\text{Sup} (\text{range} (\lambda i. (1 - p s) * \text{wp} b (M i) s)) \leq$
$\text{Sup} (\text{range} (\lambda i. p s * \text{wp} a (M i) s + (1 - p s) * \text{wp} b (M i) s))$
by(blast intro: LIMSEQ-le-const2)

} also {  
\text{have} $\text{range} (\lambda x_i. p s * \text{wp} a (M i) s + (1 - p s) * \text{wp} b (M i) s) =$
$\{ f s | f \in \text{range} (\lambda x s. p s * \text{wp} a (M x) s + (1 - p s) * \text{wp} b (M x) s) \} $
by(auto)
hence $\text{Sup} (\text{range} (\lambda i. p s * \text{wp} a (M i) s + (1 - p s) * \text{wp} b (M i) s)) =$
$\text{Sup-exp} (\text{range} (\lambda x s. p s * \text{wp} a (M x) s + (1 - p s) * \text{wp} b (M x) s)) s$
by (simp add: Sup-exp-def cong del: SUP-cong-simp)
}

finally have $p s * \text{wp} a (\text{Sup-exp} (\text{range} M)) s + (1 - p s) * \text{wp} b (\text{Sup-exp} (\text{range} M)) s \leq$
$\text{Sup-exp} (\text{range} (\lambda i s. p s * \text{wp} a (M i) s + (1 - p s) * \text{wp} b (M i) s)) s$.
moreover have $\text{Sup-exp} (\text{range} (\lambda i s. p s * \text{wp} a (M i) s + (1 - p s) * \text{wp} b (M i) s)) s \leq$
$p s * \text{wp} a (\text{Sup-exp} (\text{range} M)) s + (1 - p s) * \text{wp} b (\text{Sup-exp} (\text{range} M)) s$
Proofs (rule le-funD[OF Sup-exp-least], clarsimp, rule le-funI)

Fix \( i : \text{nat} \) and \( s : s' \)

From \( h M \) have \( \text{leSup}: M i \vdash \text{Sup-exp} (\text{range } M) \)

By (blast intro: Sup-exp-upper)

Moreover from \( s M b M \) have \( \text{sSup}: \text{sound} (\text{Sup-exp} (\text{range } M)) \)

By (auto intro: Sup-exp-sound)

Moreover note \( \text{healthy-monoD}[OF ha] s M \)

Ultimately have \( \text{wp a} (M i) \vdash \text{wp a} (\text{Sup-exp} (\text{range } M)) \) by (auto)

Hence \( \text{wp a} (M i) s \leq \text{wp a} (\text{Sup-exp} (\text{range } M)) s \) by (auto)

Moreover \{ from \( \text{leSup} \) \( s M \) \( \text{ha} \) \( \text{hb} \) \( \text{sup} \) \}

Ultimately have \( \text{wp b} (M i) \vdash \text{wp b} (\text{Sup-exp} (\text{range } M)) \) by (auto)

Hence \( \text{wp b} (M i) s \leq \text{wp b} (\text{Sup-exp} (\text{range } M)) s \) by (auto)

Moreover from \( \text{up} \) have \( 0 \leq p s \) \( 0 \leq 1 - p s \) by (auto simp: sign-simps)

Ultimately show \( p s \ast \text{wp a} (M i) s + (1 - p s) \ast \text{wp b} (M i) s \leq \)

\( p s \ast \text{wp a} (\text{Sup-exp} (\text{range } M)) s + (1 - p s) \ast \text{wp b} (\text{Sup-exp} (\text{range } M)) s \)

By (blast intro: add-mono mult-left-mono)

From \( \text{sup} \) \( \text{ha} \) \( \text{hb} \) have \( \text{sound} (\text{wp a} (\text{Sup-exp} (\text{range } M))) \)

\( \text{sound} (\text{wp b} (\text{Sup-exp} (\text{range } M))) \)

By (auto)

Hence \( \forall s. 0 \leq \text{wp a} (\text{Sup-exp} (\text{range } M)) s \) \( \forall s. 0 \leq \text{wp b} (\text{Sup-exp} (\text{range } M)) s \)

By (auto)

Moreover from \( \text{up} \) have \( \forall s. 0 \leq p s \) \( \forall s. 0 \leq 1 - p s \) by (auto simp: sign-simps)

Ultimately show \( \text{nneg} (\lambda c. p c \ast \text{wp a} (\text{Sup-exp} (\text{range } M)) c + \)

\( (1 - p c) \ast \text{wp b} (\text{Sup-exp} (\text{range } M)) c) \)

By (blast intro: add-nonneg-nonneg mult-nonneg-nonneg)

Qed

Ultimately show \( p s \ast \text{wp a} (\text{Sup-exp} (\text{range } M)) s + (1 - p s) \ast \text{wp b} (\text{Sup-exp} (\text{range } M)) s \) = \( \text{Sup-exp} (\text{range } (\lambda x. s. p s \ast \text{wp a} (M x) s + (1 - p s) \ast \text{wp b} (M x) s)) s \)

By (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:

\[ \text{SetPC a p} = \text{Bind p} (\lambda x. \text{SetPC a} (\lambda -. p)) \]

By (intro ext, simp add: SetPC-def Bind-def Let-def)

Lemma SetPC-remove:

Assumes \( nz: p x \neq 0 \) and \( n1: p x \neq 1 \)
and \( \text{fsupp: finite (supp } p) \)

shows \( \text{SetPC } a \ (\lambda x \ p \ x) \ (\text{SetPC } a \ (\lambda x \ \text{dist-remove } p \ x)) \)

proof(intro ext, simp add:SetPC-def PC-def)

fix \( ab \ P \ s \)

from \( \text{nz} \) have \( x \in \text{supp } p \) by(simp add:supp-def)

hence \( \text{supp } p = \text{insert } x \ (\text{supp } p \ - \ \{x\}) \)

by(simp)

also from \( \text{fsupp} \)

have \( \ldots = p \ x \ a \ x \ ab \ P \ s + (\sum x \in \text{supp } p \ - \ \{x\}. \ p \ x \ a \ x \ ab \ P \ s) \)

by(blast intro:sum.insert)

also from \( \text{nz} \)

have \( \ldots = p \ x \ a \ x \ ab \ P \ s + ( (1 - p \ x) * (\sum y \in \text{supp } p \ - \ \{x\}. \ p \ y / (1 - p \ x)) * a \ y \ ab \ P \ s)) \)

by(simp add:sum-divide-distrib)

also have \( \ldots = p \ x \ a \ x \ ab \ P \ s + ( (1 - p \ x) * (\sum y \in \text{supp } (\text{dist-remove } p \ x). \ \text{dist-remove } p \ x \ y * a \ y \ ab \ P \ s)) \)

by(simp add:dist-remove-def)

also from \( \text{nz} \)

have \( \ldots = p \ x \ a \ x \ ab \ P \ s + ( (1 - p \ x) * (\sum y \in \text{supp } \ (\text{dist-remove } p \ x). \ \text{dist-remove } p \ x \ y * a \ y \ ab \ P \ s)) \)

by(simp add:supp-dist-remove)

finally show \( (\sum x \in \text{supp } p. \ p \ x \ a \ x \ ab \ P \ s) = p \ x \ a \ x \ ab \ P \ s + (1 - p \ x) * (\sum y \in \text{supp } (\text{dist-remove } p \ x). \ \text{dist-remove } p \ x \ y * a \ y \ ab \ P \ s) \) .

qed

lemma cts-bot:

\( \text{bd-cts } (\lambda(P::'s \ \text{expect}) \ (s::'s). \ 0::real) \)

proof –

have \( X: \ \text{\text{\\text{\text{\text{\{P::'s \ \text{expect}\}}} \ s | P. \ P \in \text{range } (\lambda P \ s. \ 0)}) = \{0\}} \) by(auto)

show \( ?\text{thesis} \) by(intro bd-ctsI, simp add:Sup-exp-def o-def X)

qed

lemma wp-SetPC-nil:

\( \text{wp } (\text{SetPC } a \ (\lambda s \ . \ 0)) = (\lambda P \ s. \ 0) \)

by(intro ext, simp add:wp-eval)

lemma SetPC-sgl:

\( \text{supp } p = \{x\} \implies \text{SetPC } a \ (\lambda x. \ p) = (\lambda ab \ P \ s. \ p \ x \ a \ x \ ab \ P \ s) \)

by(simp add:SetPC-def)

lemma bd-cts-scale:
CHAPTER 4. THE PGCL LANGUAGE

fixes a::s
assumes ca: bd-cts a
and ha: healthy a
and nnc: 0 ≤ c
shows bd-cts (λP s. c * a P s)
proof (intro bd-ctsI ext, simp add:o-def)
fix M::nat ⇒ 's expect and d::real and s::'
assume chain: \( i \). M i ⊢ M (Suc i) and sM: \( i \). sound (M i)
and bM: \( i \). bounded-by d (M i)

from sM have \( \land i. \) nnc (M i) by(auto)
with bM have \( \land i. \) 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 \( \land s. \) \( \{ f s | f ∈ range (\lambda i. M i s) \} \) = range (λi. M i s) by(auto)
hence a (λs. c * Sup-exp (range M) s) s =
  a (λs. c * Sup (range (λi. M i s))) s
by(simp add:Sup-exp-def)
}
also { from bM have \( \land x s. x ∈ range (\lambda i. M i s) \) ⇒ x ≤ d by(auto)
with nnc have a (λs. c * Sup (range (λi. M i s))) s =
  a (λs. Sup \( \{ c * x | x ∈ range (\lambda i. M i s) \} \) ) s
by(subst cSup-mult, blast+)
}
also { have X: \( \land s. \) \( \{ c * x | x ∈ range (\lambda i. M i s) \} \) = range (λi. c * M i s) by(auto)
have a (λs. Sup \( \{ c * x | x ∈ range (\lambda i. M i s) \} \) ) s =
  a (λs. Sup (range (λi. c * M i s))) s by(simp add:X)
}
also { have \( \land s. \) range (λi. c * M i s) = \( \{ f s | f ∈ range (λi s. c * M i s) \} \)
  by(auto)
hence (λs. Sup (range (λi. c * M i s))) = Sup-exp (range (λi s. c * M i s))
  by (simp add: Sup-exp-def cong del: SUP-cong-simp)
hence a (λs. Sup (range (λi. c * M i s))) s =
  a (Sup-exp (range (λi s. c * M i s))) s by(simp)
}
also { from le-funD[OF chain] nnc
have \( \land i. \) (λs. c * M i s) ⊢ (λs. c * M (Suc i) s)
by(auto intro:le-funI[OF mult-left-mono])
moreover from sM nnc
have \( \land i. \) sound (λs. c * M i s)
by(auto intro:sound-intros)
moreover from \( bM nnc \)
have \( \bigwedge i. \text{bounded-by} \ (c \cdot d) \ (\lambda s. \ c \cdot M \ i \ s) \)
by (auto intro: mult-left-mono)
ultimately
have \( a (\text{Sup-exp} \ (\text{range} \ (\lambda i \ s. \ c \cdot M \ i \ s))) = \)
\( \text{Sup-exp} \ (\text{range} \ (a \ o \ (\lambda i \ s. \ c \cdot M \ i \ s))) \)
by (rule bd-ctsD[OF ca])
hence \( a (\text{Sup-exp} \ (\text{range} \ (\lambda i \ s. \ c \cdot M \ i \ s))) \) \( s = \)
\( \text{Sup-exp} \ (\text{range} \ (a \ o \ (\lambda i \ s. \ c \cdot M \ i \ s))) \)
sby (auto)
\}
also have \( \text{Sup-exp} \ (\text{range} \ (a \ o \ (\lambda i \ s. \ c \cdot M \ i \ s))) \) \( s = \)
\( \text{Sup-exp} \ (\text{range} \ (\lambda x. \ a \ (\lambda s. \ c \cdot M \ x \ s))) \) \( s \)
sby (simp add: a-o-def)
also \{ 
from \( nnc sM \)
have \( \bigwedge x. \ a \ (\lambda s. \ c \cdot M \ x \ s) = (\lambda s. \ c \cdot a \ (M \ x) \ s) \)
by (auto intro: scalingD[OF healthy-scalingD, OF ha, symmetric])
hence \( \text{Sup-exp} \ (\text{range} \ (\lambda x. \ a \ (\lambda s. \ c \cdot M \ x \ s))) \) \( s = \)
\( \text{Sup-exp} \ (\text{range} \ (\lambda x. \ c \cdot a \ (M \ x) \ s)) \) \( s \)
sby (simp)
\}
finally show \( c \cdot a \ (\text{Sup-exp} \ (\text{range} \ M)) \) \( s = \text{Sup-exp} \ (\text{range} \ (\lambda x. \ c \cdot a \ (M \ x) \ s)) \) \( s \).
qed

lemma cts-wp-SetPC-const:
fixes \( a :: 'a \Rightarrow 's \ prog \)
assumes ca: \( \bigwedge x. \ x \in \ (\text{supp} \ p) \Rightarrow \text{bd-cts} \ (\text{wp} \ (a \ x)) \)
and ha: \( \bigwedge x. \ x \in \ (\text{supp} \ p) \Rightarrow \text{healthy} \ (\text{wp} \ (a \ x)) \)
and up: unitary \( p \)
and sump: \( \text{sum} \ p \ (\text{supp} \ p) \leq 1 \)
and fsupp: finite \( (\text{supp} \ p) \)
shows \( \text{bd-cts} \ (\text{wp} \ (\text{SetPC} \ a \ (\lambda \cdot \ p))) \)
proof (cases \( \text{supp} \ p = \{\} \), simp add: supp-empty SetPC-def wp-def cts-bot)
assume nesupp: \( \text{supp} \ p \neq \{\} \)
from fsupp have unitary \( p \rightarrow \text{sum} \ p \ (\text{supp} \ p) \leq 1 \rightarrow \)
\( (\forall x \in \text{supp} \ p. \ \text{bd-cts} \ (\text{wp} \ (a \ x))) \rightarrow \)
\( (\forall x \in \text{supp} \ p. \ \text{healthy} \ (\text{wp} \ (a \ x))) \rightarrow \)
\( \text{bd-cts} \ (\text{wp} \ (\text{SetPC} \ a \ (\lambda \cdot \ p))) \)
proof (induct \( \text{supp} \ p \) arbitrary: \( p \), simp add: supp-empty wp-SetPC-nil cts-bot, clarify)
fix \( x :: 'a \) and \( F :: 'a \ set \) and \( p :: 'a \Rightarrow \text{real} \)
assume \( fF: \text{finite} \ F \)
assume \( \text{insert} \ x \ F = \text{supp} \ p \)
hence pstep: \( \text{supp} \ p = \text{insert} \ x \ F \) by (simp)
hence xin: \( x \in \text{supp} \ p \) by (auto)
assume \( \text{up} : \text{unitary} \ p \) and ca: \( \forall x \in \text{supp} \ p. \ \text{bd-cts} \ (\text{wp} \ (a \ x)) \)
and ha: \( \forall x \in \text{supp} \ p. \ \text{healthy} \ (\text{wp} \ (a \ x)) \)
and \( \text{sump}: \text{sum } p \ (\text{supp } p) \leq 1 \)
and \( x_{ni}: x \not\in F \)

assume IH: \( \bigwedge p. \ F = \text{supp } p \implies \)
unitary \( p \implies \text{sum } p \ (\text{supp } p) \leq 1 \implies \)
\( (\forall x \in \text{supp } p. \ \text{bd-cts } (\text{wp } (a \ x))) \implies \)
\( (\forall x \in \text{supp } p. \ \text{healthy } (\text{wp } (a \ x))) \implies \)
\( \text{bd-cts } (\text{wp } (\text{SetPC } a \ (\lambda- \ p))) \)

from \( fF \ pstep \) have \( f\text{supp}: \text{finite } (\text{supp } p) \) by (auto)
from \( xin \) have \( nzp: p \ x \neq 0 \) by (simp add: supp-def)

have \( \text{xy-le-sum}: \)
\( \bigwedge y. \ y \in \text{supp } p \implies y \neq x \implies p \ x + p \ y \leq \text{sum } p \ (\text{supp } p) \)

proof -
fix \( y \) assume \( gin: y \in \text{supp } p \) and \( yne: y \neq x \)
from \( wp \) have \( 0 \leq \text{sum } p \ (\text{supp } p - \{x, y\}) \)
by (auto intro: sum-nonneg)

hence \( p \ x + p \ y \leq p \ x + p \ y + \text{sum } p \ (\text{supp } p - \{x, y\}) \)
by (auto)
also { from \( gin \) \( yne \) \( f\text{supp} \)

have \( p \ y + \text{sum } p \ (\text{supp } p - \{x, y\}) = \text{sum } p \ (\text{supp } p - \{x\}) \)
by (subst sum.insert[symmetric], (blast intro: sum.cong)+)

moreover
from \( xin \) \( f\text{supp} \)

have \( p \ x + \text{sum } p \ (\text{supp } p - \{x\}) = \text{sum } p \ (\text{supp } p) \)
by (subst sum.insert[symmetric], (blast intro: sum.cong)+)

ultimately
have \( p \ x + p \ y + \text{sum } p \ (\text{supp } p - \{x, y\}) = \text{sum } p \ (\text{supp } p) \) by (simp)
}
finally show \( p \ x + p \ y \leq \text{sum } p \ (\text{supp } p) \).

qed

have \( n1p: \bigwedge y. \ y \in \text{supp } p \implies y \neq x \implies p \ x \neq 1 \)

proof (rule ccontr, simp)
assume \( px1: p \ x = 1 \)
fix \( y \) assume \( gin: y \in \text{supp } p \) and \( yne: y \neq x \)
from \( wp \) have \( 0 \leq p \ y \) by (auto)

with \( gin \) have \( 0 < p \ y \) by (auto simp: supp-def)

hence \( 0 + p \ x < p \ y + p \ x \) by (rule add-strict-right-mono)

with \( px1 \) have \( 1 < p \ x + p \ y \) by (simp)

also from \( gin \) \( yne \) have \( p \ x + p \ y \leq \text{sum } p \ (\text{supp } p) \)

by (rule xy-le-sum)
finally show \( \text{False} \) using \( \text{sump} \) by (simp)

qed

show \( \text{bd-cts } (\text{wp } (\text{SetPC } a \ (\lambda- \ p))) \)

proof (cases \( F = \{\} \))
4.3. CONTINUITY

case True with pstep have supp p = \{x\} by(simp)
hence wp (SetPC a (λs. p x * wp (a x) P s)) by(simp add:SetPC-sgl wp-def)
moreover {
  from up ca ha xin have bd-cts (wp (a x)) healthy (wp (a x)) 0 ≤ p x by(auto)
hence bd-cts (λP s. p x * wp (a x) P s) by(rule bd-cts-scale)
}
ultimately show ?thesis by(simp)
next
assume neF: F ≠ {} then obtain y where yinF: y ∈ F by(auto)
with xni have yne: y ≠ x by(auto)
from yinF pstep have yin: y ∈ supp p by(auto)

from supp-dist-remove[of p x, OF nzp n1p, OF yin yne] have supp-sub: supp (dist-remove p x) ≤ supp p by(auto)

from xin ca have cax: bd-cts (wp (a x)) by(auto)
from xin ha have hax: healthy (wp (a x)) by(auto)

from supp-sub ha have hra: \forall x ∈ supp (dist-remove p x). healthy (wp (a x)) by(auto)
from supp-sub ca have cra: \forall x ∈ supp (dist-remove p x). bd-cts (wp (a x)) by(auto)

from supp-dist-remove[of p x, OF nzp n1p, OF yin yne] pstep xni have Fsupp: F = supp (dist-remove p x)
  by(simp)

have udp: unitary (dist-remove p x)
proof(intro unitaryI2 nnegI bounded-byI)
fix y
show 0 ≤ dist-remove p x y proof(cases y=x, simp-all add:dist-remove-def)
  from up have 0 ≤ p y 0 ≤ 1 − p x by(auto simp:sign-simps)
  thus 0 ≤ p y / (1 − p x) by(rule divide-nonneg-nonneg)
qed

show dist-remove p x y ≤ 1 proof(cases y=x, simp-all add:dist-remove-def, cases y∈ supp p, simp-all add:nsupp-zero)
  assume yne: y ≠ x and yin: y ∈ supp p
  hence p x + p y ≤ sum p (supp p) by(auto intro:xy-le-sum)
  also note sump
  finally have p y ≤ 1 − p x by(auto)
  moreover from up have p x ≤ 1 by(auto)
moreover from \( yin \ yne \) have \( p \ x \neq 1 \) by (rule \( n1p \))
ultimately show \( p \ y / (1 - p \ x) \leq 1 \) by (auto)
qed
qed

from \( xin \) have \( pxn0 \): \( p \ x \neq 0 \) by (auto simp: supp-def)
from \( yin \ yne \) have \( pxn1 \): \( p \ x \neq 1 \) by (rule \( n1p \))

from \( pxn0 \ pxn1 \) have \( \sum (\text{dist-remove} \ p \ x) (\text{supp} (\text{dist-remove} \ p \ x)) = \sum (\text{dist-remove} \ p \ x) (\text{supp} \ p - \{x\}) \)
by (simp add: supp-dist-remove)
also have \( \cdots = (\sum y \in \text{supp} \ p - \{x\}. \ p \ y / (1 - p \ x)) \)
by (simp add: dist-remove-def)
also have \( \cdots = (\sum y \in \text{supp} \ p - \{x\}. \ p \ y) / (1 - p \ x) \)
by (simp add: sum-divide-distrib)
also { from \( xin \) have \( \text{insert} \ x \ (\text{supp} \ p) = \text{supp} \ p \) by (auto)
with \( Fsupp \) have \( \text{upx} \): unitary (\( \lambda \cdot. \ p \ x \)) by (auto)
from \( pxn0 \ pxn1 \) \( Fsupp \) \( Hr \) \( Cra \) \( IH \) have \( \text{cts-dr: bd-cts} \ (\text{up} (\text{SetPC} \ a \ (\lambda\cdot. \text{dist-remove} \ p \ x))) \)
by (auto)
from \( up \) have \( \text{upx: unitary} \ (\lambda\cdot. \ p \ x) \) by (auto)

from \( pxn0 \ pxn1 \) \( Fsupp \) \( Hr \) \( Cra \) \( IH \) show \( \text{?thesis} \)
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 \( cts-wp-SetPC \):
fixes \( a::\ 'a \Rightarrow 's \) prog
assumes \( ca: \ \forall x \ s. \ x \in (\text{supp} \ (p \ s)) \Rightarrow \text{bd-cts} \ (\text{wp} \ (a \ x)) \)
4.3. CONTINUITY

and ha: ∀x s. x ∈ (supp (p s)) → healthy (wp (a x))
and wp: ∀s. unitary (p s)
and sump: ∀s. sum (p s) (supp (p s)) ≤ 1
and fsupp: ∀s. finite (supp (p s))
shows bd-cts (wp (SetPC a p))

proof –
from assms have bd-cts (wp (Bind p (λp. SetPC a (λ-. p))))
by (iprover intro: cts-wp-Bind cts-wp-SetPC-const)
thus ?thesis by (simp add: SetPC-Bind [symmetric])
qed

lemma wp-SetDC-Bind:
SetDC a S = Bind S (λS. SetDC a (λ-. S))
by (intro ext, simp add: SetDC-def Bind-def)

lemma SetDC-finite-insert:
assumes fS: finite S
and neS: S ≠ {}
shows SetDC a (λ-. insert x S) = a x ∩ SetDC a (λ-. S)
proof (intro ext, simp add: SetDC-def DC-def cong del: image-cong-simp cong add: INF-cong-simp)
fix ab P s
from fS have A: finite (insert (a x ab P s) ((λx. a x ab P s) ‘ S))
and B: finite (((λx. a x ab P s) ‘ S)) by (auto)
from neS have C: insert (a x ab P s) ((λx. a x ab P s) ‘ S) ≠ {} 
and D: (λx. a x ab P s) ‘ S ≠ {} by (auto)
from A C have Inf (insert (a x ab P s) ((λx. a x ab P s) ‘ S)) =
Min (insert (a x ab P s) ((λx. a x ab P s) ‘ S))
by (auto intro: cInf-eq-Min)
also from B D have ... = min (a x ab P s) (Min ((λx. a x ab P s) ‘ S))
by (auto intro: Min-insert)
also from B D have ... = min (a x ab P s) (Inf ((λx. a x ab P s) ‘ S)))
by (simp add: cInf-eq-Min)
finally show (INF x∈insert x S. a x ab P s) =
min (a x ab P s) (INF x∈S. a x ab P s)
by (simp cong del: INF-cong-simp)
qed

lemma SetDC-singleton:
SetDC a (λ-. {x}) = a x
by (simp add: SetDC-def cong del: INF-cong-simp)

lemma cts-wp-SetDC-const:
fixes a::′a ⇒ 's prog
assumes ca: ∀x. x ∈ S → bd-cts (wp (a x))
and ha: ∀x. x ∈ S → healthy (wp (a x))
and fS: finite S
and neS: S ≠ {}
shows bd-cts (wp (SetDC a (λ-. S))))
4.3.2 Continuity of a Single Loop Step

A single loop iteration is continuous, in the more general sense defined above for transformer transformers.

lemma cts-wp-loopstep:
fixes body :: 's prog
proof (induct n, auto intro: cts-wp-Skip cts-wp-Seq healthy-intros)
4.3. CONTINUITY

assumes \( \text{hb: healthy (wp body)} \)
and \( \text{cb: bd-cts (wp body)} \)
shows \( \text{bd-cts-tr (} \lambda x. \text{ wp (body ; Embed } x \in G \oplus \text{ Skip}) \text{) (is bd-cts-tr } ?F) \)

proof (rule bd-cts-trI, rule le-trans-antisym)
fix \( \text{M :: nat } \Rightarrow \text{'}s \text{ trans and } b :: \text{real} \)
assume chain: \( \land i. \text{ le-trans (} M i \text{) (} M (\text{Suc } i) \text{)} \)
and \( \text{fM: } \land i. \text{ feasible (} M i \text{)} \)

show \( \text{fu: le-trans (Sup-trans (range (} ?F \circ M \text{))) (} ?F \text{ (Sup-trans (range } M \text{)))} \)
proof (rule le-transI[OF Sup-trans-least2], clarsimp)
fix \( P Q :: \text{'}s \text{ expect and } t \)
assume \( \text{nP: sound } P \)
assume \( \text{nQ: nneg } Q \text{ and } bP: \text{ bounded-by (bound-of } P \text{) } Q \)

hence \( \text{nQ: sound } Q \text{ by(auto)} \)

from \( \text{fM have fSup: feasible (Sup-trans (range } M \text{))} \)
by (auto intro:feasible-Sup-trans)

from \( \text{sQ} \text{ fM have } M t Q \vdash \text{Sup-trans (range } M \text{) } Q \)
by (auto intro:Sup-trans-upper2)

moreover from \( \text{sQ fM} \)

have \( \text{sMtP: sound (} M t Q \text{) sound (Sup-trans (range } M \text{) } Q \text{) by(auto)} \)

ultimately have \( \text{wp body (} M t Q \text{) } \vdash \text{wp body (Sup-trans (range } M \text{) } Q \text{)} \)
using \( \text{healthy-monoD[OF hb] by(auto)} \)

hence \( \land s. \text{ wp body (} M t Q \text{) } s \leq \text{ wp body (Sup-trans (range } M \text{) } Q \text{) } s \)
by (rule le-funI)

thus \( ?F ( \text{Sup-trans (range } M \text{)) } Q \)
by (intro le-funI, simp add:wp-eval mult-left-mono)

show \( \text{nneg (wp (body ; Embed (Sup-trans (range } M \text{)) } G \oplus \text{ Skip}) } Q \)
proof (rule nnegI, simp add:wp-eval)
fix \( s :: \text{'}s \)
from \( \text{fSup sQ have sound (Sup-trans (range } M \text{) } Q \text{) by(auto)} \)
with \( \text{hb have sound (wp body (Sup-trans (range } M \text{) } Q \text{)) by(auto)} \)

hence \( 0 \leq \text{ wp body (Sup-trans (range } M \text{) } Q \text{) } s \text{ by(auto)} \)

moreover from \( \text{sQ have } 0 \leq Q s \text{ by(auto)} \)

ultimately show \( 0 \leq \langle G \rangle s \ast \text{ wp body (Sup-trans (range } M \text{) } Q \text{) } s + (1 - \langle G \rangle s) \ast Q s \)
by (auto intro:add-nonneg-nonneg mult-nonneg-nonneg)

qed

next
fix \( P :: \text{'}s \text{ expect assume } sP: \text{ sound } P \)
thus \( \text{nneg } P \text{ bounded-by (bound-of } P \text{) } P \text{ by(auto)} \)

show \( \forall u : \text{range (} (\lambda x. \text{ wp (body ; Embed } x \in G \oplus \text{ Skip}) \circ M) \).
\forall R. \text{ nneg R } \land \text{ bounded-by (bound-of } P \text{) } R \longrightarrow \text{ nneg (} u \text{ R) } \land \text{ bounded-by (bound-of } P \text{) } (u R) \)
proof (clarsimp, intro conjI nnegI bounded-byI, simp add:wp-eval)
fix \( u :: \text{'}s \text{ expect and } s :: \text{'}s \)
assume \( \text{nR: nneg } R \text{ and } bR: \text{ bounded-by (bound-of } P \text{) } R \)

hence \( sR: \text{ sound } R \text{ by(auto)} \)
with \( fM \) have \( sMuR: \text{sound} (M u R) \) by(auto)
with \( hh \) have \( \text{sound} (\text{wp body} (M u R)) \) by(auto)
hence \( 0 \leq \text{wp body} (M u R) \) s by(auto)
moreover from \( nR \) have \( 0 \leq R \) s by(auto)
ultimately show \( 0 \leq \langle G \rangle s * \text{wp body} (M u R) s + (1 - \langle G \rangle s) * R s \) by(auto intro: add-nonneg-nonneg mult-nonneg-nonneg)

from \( sR bR fM \) have \( \text{bounded-by} (\text{bound-of} P) (M u R) \) by(auto)
with \( sMuR hh \) have \( \text{bounded-by} (\text{bound-of} P) (\text{wp body} (M u R)) \) by(auto)
hence \( \text{wp body} (M u R) s \leq \text{bound-of} P \) by(auto)
moreover from \( bR \) have \( \tilde{R} s \leq \text{bound-of} P \) by(auto)
ultimately have \( \langle G \rangle s * \text{wp body} (M u R) s + (1 - \langle G \rangle s) * R s \leq \langle G \rangle s * \text{bound-of} P + (1 - \langle G \rangle s) * \text{bound-of} P \) by(auto intro: add-nonneg-nonneg mult-left-mono)
also have \( \ldots = \text{bound-of} P \) by(simp add: algebra-simps)
finally show \( \langle G \rangle s * \text{wp body} (M u R) s + (1 - \langle G \rangle s) * R s \leq \text{bound-of} P \).
  qed
qed

show le-trans (?F (Sup-trans (range M))) (Sup-trans (range (?F o M)))
proof (rule le-transI, rule le-funI, simp add: wp-eval cong del: image-cong-simp)
fix \( P \), s expect and \( s' \),
assume \( sP: \text{sound} P \)
have \( \{ t P \mid t. t \in \text{range} M \} = \text{range} (\lambda i. M i P) \) by(blast)
hence \( \text{wp body} (\text{Sup-trans} (\text{range} M) P) s = \text{wp body} (\text{Sup-exp} (\text{range} (\lambda i. M i P))) s \) by(simp add: Sup-trans-def)
also { from \( sP fM \) have \( \forall i. \text{sound} (M i P) \) by(auto)
moreover from \( sP \) chain have \( \forall i. M i P \vdash M (\text{Suc} i) P \) by(auto)
moreover { from \( sP \) have \( \text{bounded-by} (\text{bound-of} P) P \) by(auto)
with \( sP fM \) have \( \forall i. \text{bounded-by} (\text{bound-of} P) (M i P) \) by(auto)
}
ultimately have \( \text{wp body} (\text{Sup-exp} (\text{range} (\lambda i. \text{wp body} (M i P)))) s = \text{Sup-exp} (\text{range} (\lambda i. \text{wp body} (M i P))) s \) by(subst bd-ctsD[OF cb], auto simp:o-def)
}
also have \( \text{Sup-exp} (\text{range} (\lambda i. \text{wp body} (M i P))) s = \text{Sup} \{ f s \mid f. f \in \text{range} (\lambda i. \text{wp body} (M i P)) \} \) by(simp add: Sup-exp-def)
finally have \( \langle G \rangle s * \text{wp body} (\text{Sup-trans} (\text{range} M) P) s + (1 - \langle G \rangle s) * P s = \langle G \rangle s * \text{Sup} \{ f s \mid f. f \in \text{range} (\lambda i. \text{wp body} (M i P)) \} + (1 - \langle G \rangle s) * P s \) by(simp)
also {
4.3. CONTINUITY

from $sp fm$ have $\bigwedge i. \text{sound} \ (M i P) \ \text{by} (\text{auto})$
moreover from $sp fm$ have $\bigwedge i. \text{bounded-by} \ (\text{bound-of} \ P) \ (M i P) \ \text{by} (\text{auto})$
ultimately have $\bigwedge i. \text{bounded-by} \ (\text{bound-of} \ P) \ (wp \ \text{body} \ (M i P))$ using $hb$
by (auto)
hence bound \ $\forall i. \ \text{wp body} \ (M i P) \ s \leq \ \text{bound-of} \ P$ by (auto)
moreover
have $\{ s G s \ast x \ x \in \{ f s \ | f \in \text{range} \ (\lambda i. \ \text{wp body} \ (M i P))\} =$
$\{ s G s \ast f s \ | f \in \text{range} \ (\lambda i. \ \text{wp body} \ (M i P))\}$
by (blast)
ultimately
have $\forall i. \ s G s \ast \text{wp body} \ (M i P) \ s \leq \ \text{bound-of} \ P$
by (cases $G s$, auto)
moreover
have $\text{Sup} \ \{ s G s \ast f s \ | f \in \text{range} \ (\lambda i. \ \text{wp body} \ (M i P))\} + (1 - s G s)$
$P s =$
$\text{Sup} \ \{ s G s \ast f s + (1 - s G s) \ast P s \ | f \in \text{range} \ (\lambda i. \ \text{wp body} \ (M i P))\}$
by (subst $cSup$-add, auto)
moreover
ultimately
have $\forall i. \ s G s \ast \text{wp body} \ (M i P) \ s + (1 - s G s)$
$P s =$
$\text{Sup} \ \{ s G s \ast f s + (1 - s G s) \ast P s \ | f \in \text{range} \ (\lambda i. \ \text{wp body} \ (M i P))\}$
by (simp)

also
have $\bigwedge i. \ s G s \ast \text{wp body} \ (M i P) \ s + (1 - s G s) \ast P s =$
$\lambda x. \ \text{wp body} \ (\text{Embed} x \ G s \oplus \text{Skip}) \ o M \ i P s$
by (simp add: wp-eval)
also have $\bigwedge i. \ ... \ i \ \leq$
$\text{Sup} \ \{ f s \ | f \in \{ t P \ | t. \ t \in \text{range} \ ((\lambda x. \ \text{wp body} \ (\text{Embed} x \ G s \oplus \text{Skip})) \ o M)\} \}$
proof (intro $cSup$-upper bdd-aboveI, blast, clarsimp simp: wp-eval)
fix $i$
from $sp fm$ have $hp: \ \text{bounded-by} \ (\text{bound-of} \ P) \ P$ by (auto)
with $sp fm$ have $\text{sound} \ (M i P)$ bounded-by (bound-of $P$) (M i P) by (auto)
with $hp$ have bounded-by (bound-of $P$) (wp body (M i P)) by (auto)
with $hp$ have bounded-of $P$ s $\leq$ bound-of $P$ s $\leq$ bound-of $P$ by (auto)
hence $s G s \ast \text{wp body} \ (M i P) \ s + (1 - s G s) \ast P s \leq$

CHAPTER 4. THE PGCL LANGUAGE

«G» s * (bound-of P) + (1 - «G» s) * (bound-of P)
by(auto intro:add-mono mult-left-mono)
also have ... = bound-of P by(simp add:algebra-simps)
finally show «G» s * wp body (M i P) s + (1 - «G» s) * P s ≤ bound-of P.
qed
finally have Sup {«G» s * f s + (1 - «G» s) * P s | f ∈ range (λi. wp body (M i P))} ≤ Sup {f s | f ∈ {t P | t ∈ range ((λx. wp (body ;; Embed x « G » ⊕ Skip)) ◦ M))} by(blast intro:Sup-least)
also have Sup {f s | f ∈ {t P | t ∈ range ((λx. wp (body ;; Embed x « G » ⊕ Skip)) ◦ M))} = Sup-trans (range ((λx. wp (body ;; Embed x « G » ⊕ Skip)) ◦ M)) P s by(simp add:Sup-trans-def Sup-exp-def)
finally show «G» s * wp body (Sup-trans (range M) P) s + (1 - «G» s) * P s ≤ Sup-trans (range ((λx. wp (body ;; Embed x « G » ⊕ Skip)) ◦ M)) P s .
qed
qed
end

4.4 Continuity and Induction for Loops

theory LoopInduction imports Healthiness Continuity begin

Showing continuity for loops requires a stronger induction principle than we have used so far, which in turn relies on the continuity of loops (inductively). Thus, the proofs are intertwined, and broken off from the main set of continuity proofs. This result is also essential in showing the sublinearity of loops.

A loop step is monotonic.

lemma wp-loop-step-mono-trans:
fixes body::'s prog
assumes sP: sound P
and hb: healthy (wp body)
shows mono-trans (λQ s. « G » s * wp body Q s + « N' G » s * P s)
proof(intro mono-transI le-funI, simp)
fix Q R::'s expect and s::'s
assume sQ: sound Q and sR: sound R and le: Q ⊢ R
hence wp body Q ⊢ wp body R
by(rule mono-transD[OF healthy-monoD, OF hb])
thus «G» s * wp body Q s ≤ «G» s * wp body R s
4.4. CONTINUITY AND INDUCTION FOR LOOPS

by(auto dest:le-funD intro:mult-left-mono)

qed

We can therefore apply the standard fixed-point lemmas to unfold it:

lemma lfp-wp-loop-unfold:
  fixes body::'s prog
  assumes hb: healthy (wp body)
      and sP: sound P
  shows lfp-exp (λQ s. «G» s * wp body Q s + «N G» s * P s) =
    (λs. «G» s * wp body (lfp-exp (λQ s. «G» s * wp body Q s + «N G» s * P s)) s +
        «N G» s * P s)

proof (rule lfp-exp-unfold)
  from assms show mono-trans (λQ s. «G» s * wp body Q s + «N G» s * P s)
    by(blast intro:wp-loop-step-mono-trans)
  from assms show λs. «G» s * wp body (λs. bound-of P) s + «N G» s * P s ⊢
    λs. bound-of P
    by(blast intro:lfp-loop-fp)
  from sP show sound (λs. «G» s * wp body Q s + «N G» s * P s)
    by(auto)
  fix Q::'s expect
  assume sound Q
  with assms show sound (λs. «G» s * wp body Q s + «N G» s * P s)
    by(intro wp-loop-step-sound[unfolded wp-eval, simplified, folded negate-embed],
        auto)

qed

lemma wp-loop-step-unitary:
  fixes body::'s prog
  assumes hb: healthy (wp body)
      and uP: unitary P and uQ: unitary Q
  shows unitary (λs. «G» s * wp body Q s + «N G» s * P s)

proof (intro unitaryI2 nnegI bounded-byI)
  fix s::'
  from uQ hb have uuQ: unitary (wp body Q) by(auto)
  with uP have 0 ≤ wp body Q s 0 ≤ P s by(auto)
  thus 0 ≤ «G» s * wp body Q s + «N G» s * P s
    by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)

  from uP uuQ have wp body Q s ≤ 1 P s ≤ 1 by(auto)
  hence «G» s * wp body Q s + «N G» s * P s ≤ «G» s * 1 + «N G» s * 1
    by(blast intro:add-mono mult-left-mono)
  also have ... = 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 \((lfp-exp (\lambda Q s. \ll G s \rr s \ast \wp body Q s + \ll N G s \rr s \ast P s))\)
using assms by (blast intro: lfp-exp-unitary wp-loop-step-unitary)

From the lattice structure on transformers, we establish a transfinite induction principle for loops. We use this to show a number of properties, particularly subdistributivity, for loops. This proof follows the pattern of lemma \( \text{lfp}_\text{OrdinalInduct} \) in HOL/Inductive.

**lemma** \( \text{loop-induct} \):

fixes body :: 's prog
assumes hwp: healthy \((\wp \text{body})\)
and hwlp: nearly-healthy \((\wlp \text{body})\)

— The body must be healthy, both in strict and liberal semantics.

and Limit: \( \forall S. \left[ \forall x \in S. P (\text{fst} x) (\text{snd} x) ; \forall x \in S. \text{feasible} (\text{fst} x) ; \forall x \in S. \forall Q. \text{unitary} Q \longrightarrow \text{unitary} (\text{snd} x Q) \right] \implies \twoheadrightarrow P (\text{Sup-trans (fst ' S)}) (\text{Inf-trans (snd ' S)}) \)

— The property holds at limit points.

and IH: \( \forall t u. \left[ \twoheadrightarrow P t u ; \text{feasible } t ; \forall Q. \text{unitary } Q = \implies \text{unitary} (u Q) \right] \implies \twoheadrightarrow P t' u' \)

— The inductive step. The property is preserved by a single loop iteration.

and \( \text{P-equiv} \): \( \forall t t' u u'. \left[ \twoheadrightarrow P t u ; \equiv-trans t t' ; \equiv-utrans u u' \right] \implies \twoheadrightarrow P t' u' \)

— The property must be preserved by equivalence

shows \( \twoheadrightarrow P (\wp (\text{do } G \longrightarrow \text{body od})) (\wlp (\text{do } G \longrightarrow \text{body od})) \)

— The property can refer to both interpretations simultaneously. The unifier will happily apply the rule to just one or the other, however.

**proof** \((\text{simp add: wp-eval})\)

let \(?X t = \wp (\text{body } ;; \text{Embed } t \ll G \rr G s \oplus \text{Skip})\)
let \(?Y t = \wlp (\text{body } ;; \text{Embed } t \ll G \rr G s \oplus \text{Skip})\)

let \(?M = \{ x . P (\text{fst} x) (\text{snd} x) \land \text{feasible} (\text{fst} x) \land (\forall Q. \text{unitary } Q \longrightarrow \text{unitary} (\text{snd} x Q)) \land \text{le-trans} (\text{fst} x) (lfp-trans ?X) \land \text{le-utrans} (gfp-trans ?Y) (\text{snd} x)\}\)

have \( \text{fSup: feasible} (\text{Sup-trans (fst ' ?M)}) \)
**proof** \((\text{intro feasibleI bounded-byI2 nnegI2})\)

fix \( Q::'s \text{ expect and } b::'s \text{ real} \)
assume \( nQ: \text{nneg } Q \text{ and } bQ: \text{bounded-by } b \text{ } Q \)
show \( \text{Sup-trans (fst ' ?M) } Q \vdash \lambda s. b \)

unfolding \( \text{Sup-trans-def} \)
using \( nQ bQ \text{ by (auto intro!: Sup-exp-least)} \)
show \( \lambda s. 0 \vdash \text{Sup-trans (fst ' ?M) } Q \)

**proof** \((\text{cases})\)
assume \( \text{empty: } ?M = \{\} \)
show \( \text{thesis by (simp add: Sup-trans-def Sup-exp-def empty)} \)
next
assume ne: ?M \neq \{\}
then obtain x where \(x \in ?M\) by auto
hence \(\text{ff: feasible (fst x) by(simp)}\)
with \(nQ \ bQ\) have \(\lambda s. 0 \vdash \lambda s. 0\) by(auto)
also from \(x \in ?M\) have \(\text{fst x \vdash Sup-trans (fst ' ?M) Q}\)
  apply(intro Sup-trans-upper2[OF imageI - nQ bQ], assumption)
  apply(clarsimp, blast intro: sound-nneg[OF feasible-sound feasible-boundedD])
done
finally show \(\lambda s. 0 \vdash \lambda s. 0\) done
qed
qed

have \(uInf: \bigwedge P. \text{unitary P \implies unitary (Inf-utrans (snd ' ?M) P)}\)
proof(cases ?M = \{\})
fix P
assume empty: ?M = \{\}
show \(\text{thesis P by(simp only:empty, simp add:Inf-utrans-def)}\)
next
fix P::'s expect
assume uP: \text{unitary P}
  and ne: ?M \neq \{\}
show \(\text{thesis P}\)
proof(intro unitaryI2 nnegI2 bounded-byI2)
  from \(x \in ?M\) by auto
hence \(\text{sxin: snd x} \in \text{snd ' ?M by(simp)}\)
  by(intro Inf-utrans-lower, auto)
with \(uP\)
have \(\text{Inf-utrans (snd ' ?M) P} \vdash \text{snd x P by(auto)}\)
also {\ }
  from \(x \in ?M\) have \(\text{unitary (snd x P) by(simp)}\)
  hence \(\text{snd x P \vdash \lambda s. 1 by(auto)}\)
}
finally show \(\text{Inf-utrans (snd ' ?M) P} \vdash \lambda s. 1\) .

have \(\lambda s. 0 \vdash \text{Inf-trans (snd ' ?M) P}\)
unfolding Inf-trans-def
proof(rule Inf-exp-greatest)
  from \(sxin\) 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 \(t::'s\) trans
  assume \(\forall Q. \text{unitary Q \implies unitary (t Q)}\)
  with \(uP\) have \(\text{unitary (t P) by(auto)}\)
  thus \(\lambda s. 0 \vdash t P\) by(auto)
qed
qed
also {\ }
  from \(\text{ne have X: (snd ' ?M = \{\}) = False by(simp)}\)
have Inf-trans (snd ' ?M) P = Inf-utrans (snd ' ?M) P

unfolding Inf-utrans-def by (subst X, simp)

} finally show λs. 0 ⊢ Inf-utrans (snd ' ?M) P .

qed

have wp-loop-mono: \( \forall t \; u. \; [\text{le-trans } t \; u; \; \forall P. \; \text{sound } P \implies \text{sound } (t \; P); \; \forall P. \; \text{sound } P \implies \text{sound } (u \; P) ] \implies \text{le-trans } (\forall X \; t) (\forall X \; u) \)

proof (intro le-transI le-funI, simp add: wp-eval)

fix t u::s trans and P::s expect and s::s

assume le: le-trans t u

and st: \( \forall P. \; \text{sound } P \implies \text{sound } (t \; P) \)

and su: \( \forall P. \; \text{sound } P \implies \text{sound } (u \; P) \)

and sP: sound P

hence sound (t \; P) sound (u \; P) by (auto)

with healthy-monoD[OF hwp] le sP have wp body (t \; P) \vdash wp body (u \; P) by (auto)

hence wp body (t \; P) s \leq wp body (u \; P) s by (auto)

thus «G» s * wp body (t \; P) s \leq «G» s * wp body (u \; P) s by (auto intro: mult-left-mono)

qed

have plimit: P (Sup-trans (fst ' ?M)) (Inf-utrans (snd ' ?M))

by (auto intro: Limit)
4.4. CONTINUITY AND INDUCTION FOR LOOPS

have feasible-lfp-loop:
  feasible (lfp-trans ?X)

proof
  intro feasibleI bounded-byI2 nnegI2,
  simp-all add:wp-Loop1 [simplified wp-eval] soundI2 hpw
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-lowerbound 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 hpw 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: lfp-exp-upperbound le-funI)
  thus «G» s * wp body (λs. b) s + «N' G» s * P s ≤ «G» s * b + «N' G» s * b
  by(auto intro: add-nonneg-nonneg mult-nonneg-nonneg)
  from nnb show sound (λs. b) by(auto)
qed

from hpw sP show λs. 0 ⊢ lfp-exp (λQ s. « G » s * wp body Q s + « N' G » s * P s) s * P s)
  by(blast intro!: lfp-exp-greatest lfp-loop-fp)
qed

have unitary-gfp:
  ∀P. unitary P ⇒ unitary (gfp-trans ?Y P)
proof
  intro unitaryI2 nnegI2 bounded-byI2,
  simp-all add: wp-Loop1 [simplified wp-eval] hpw
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 hpw nP have 0 ≤ wp body (λs. 0) s 0 ≤ P s by(auto dest!: unitary-sound)
  thus 0 ≤ «G» s * wp body (λs. 0) s + «N' G» s * P s
  by(auto intro: add-nonneg-nonneg mult-nonneg-nonneg)
  show unitary (λs. 0) by(auto)
qed

show gfp-exp (λQ s. « G » s * wp body Q s + « N' G » s * P s) ⊢ λs. 1
  by(auto intro: gfp-exp-least)
qed

have fX:
  ∀t. 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 \( \text{nng} (t \ Q) \) bounded-by \( b (t \ Q) \) by(auto)
moreover hence \( \text{stQ} \) sound \( (t \ Q) \) by(auto)
ultimately have wp body \( (t \ Q) \) \( s \leq b \) using hwlp by(auto)
moreover from \( bQ \) have \( Q s \leq b \) by(auto)
ultimately have \( \langle G \rangle s \ast \) wp body \( (t \ Q) \) \( s + (1 - \langle G \rangle s) \ast Q s \leq \langle G \rangle s \ast b + (1 - \langle G \rangle s) \ast b \)
by(auto intro:add-mono mult-left-mono)
thus \( \langle G \rangle s \ast \) wp body \( (t \ Q) \) \( s + (1 - \langle G \rangle s) \ast Q s \leq b \)
by(simp add:algebra-simps)

from \( nQ \) \( \text{stQ} \) hwlp have \( 0 \leq \) wp body \( (t \ Q) \) \( s \) \( 0 \leq Q s \) by(auto)
thus \( 0 \leq \langle G \rangle s \ast \) wp body \( (t \ Q) \) \( s + (1 - \langle G \rangle s) \ast Q s \)
by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)
qed

have \( uY : \)
\( \forall t \ P. (\forall P. \text{unary} P \implies \text{unitary} (t \ P)) \implies \text{unitary} P \implies \text{unitary} (?Y t \ P) \)
proof(intro unitaryI2 nnegI bounded-byI, simp-all add:wp-eval)
fix t::'s trans and P::'s expect and s::'s
assume at: \( \forall P. \text{unary} P \implies \text{unitary} (t \ P) \)
and uP: \( \text{unitary} P \)
hence utP: \( \text{unitary} (t \ P) \) by(auto)
with hwlp have ubtP: \( \text{unitary} (\text{wp body} (t \ P)) \) by(auto)
with uP have \( 0 \leq P s \) \( 0 \leq \text{wp body} (t \ P) \) s by(auto)
thus \( 0 \leq \langle G \rangle s \ast \text{wp body} (t \ P) \) \( s + (1 - \langle G \rangle s) \ast P s \)
by(auto intro:add-nonneg-nonneg mult-nonneg-nonneg)

from uP ubtP have \( P s \leq 1 \) wp body \( (t \ P) \) \( s \leq 1 \) by(auto)
hence \( \langle G \rangle s \ast \text{wp body} (t \ P) \) \( s + (1 - \langle G \rangle s) \ast P s \leq \langle G \rangle s \ast 1 + (1 - \langle G \rangle s) \ast 1 \)
by(blast intro:add-mono mult-left-mono)
also have ... = 1 by(simp add:algebra-simps)
finally show \( \langle G \rangle s \ast \text{wp body} (t \ P) \) \( s + (1 - \langle G \rangle s) \ast P s \leq 1 \)
qed

have fu-lfp: 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 le-trans \( (?X (\text{Sup-trans} (\text{fst} \ ?M))) \) \( (?X (\text{lfp-trans} ?X)) \)
by(auto intro:wp-loop-mono feasible-sound[OF fSup]
    feasible-sound[OF feasible-lfp-loop])
also have equiv-trans ... \( (\text{lfp-trans} ?X) \)
proof(rule iffD1[OF equiv-trans-comm, OF lf-p-trans-unfold], iprover intro:wp-loop-mono)
fix t::'s trans and P::'s expect
assume st: \( \forall Q. \text{sound} Q \implies \text{sound} (t \ Q) \)
and sP: \( \text{sound} P \)
show sound \( (?X t \ P) \)
proof(intro soundI2 bounded-byI nnegI, simp-all add:wp-eval)
fix s::'s
from sP st hwp have \(0 \leq P s 0 \leq \text{wp body} \ (t P) s\) by(auto)
thus \(0 \leq \langle G \rangle s \ast \text{wp body} \ (t P) s + (1 - \langle G \rangle s) \ast P s\)
by(blast intro:add-nonneg-nonneg mult-nonneg-nonneg)
from sP st have bounded-by (bound-of (t P)) (t P) by(auto)
with sP st hwp have bounded-by (bound-of (t P)) (wp body (t P)) by(auto)
hence wp body (t P) s \leq bound-of (t P) by(auto)
moreover from sP st hwp have \(P s \leq \text{bound-of} \ P\) by(auto)
moreover have \(\langle G \rangle s \leq 1 1 - \langle G \rangle s \leq 1\) by(auto)
moreover from sP st hwp have \(0 \leq \text{wp body} \ (t P) s 0 \leq P s\) by(auto)
moreover have \((\text{?fp :: real}) \leq 1\) by(simp)
ultimately show \(\langle G \rangle s \ast \text{wp body} \ (t P) s + (1 - \langle G \rangle s) \ast P s \leq 1 \ast \text{bound-of} \ (t P) + 1 \ast \text{bound-of} \ P\)
by(blast intro:add-mono mult-mono)
qed
next
let \(\text{?fp} = \lambda R \text{ s. bound-of} \ R\)
show le-trans \((\text{?X} \text{ ?fp}) \ ?fp\) by(auto intro:healthy-intros hwp)
fix \(P::\)’s expect assume sound \(P\)
thus sound \((\text{?fp} \ P)\) by(auto)
qed
finally have le-lfp: le-trans \((\text{?X} \ (\text{Sup-trans} \ (\text{fst} \ ?M)))\) \((\text{lfp-trans} \ ?X)\).

have fu-gfp: le-utrans \((\text{gfp-trans } ?Y)\) \((\text{Inf-utrans} \ (\text{snd} \ ?M))\)
by(auto intro:Inf-utrans-greatest unitary-gfp)

have equic-utrans \((\text{gfp-trans } ?Y)\) \((?Y \ (\text{gfp-trans } ?Y))\)
by(auto intro:gfp-trans-unfold wlp-loop-mono uY)
also from fu-gfp have le-utrans \((\text{?Y} \ (\text{gfp-trans } ?Y))\) \((?Y \ (\text{Inf-utrans} \ (\text{snd} \ ?M)))\)
by(auto intro:wlp-loop-mono uInf unitary-gfp)
finally have ge-gfp: le-utrans \((\text{gfp-trans } ?Y)\) \((?Y \ (\text{Inf-utrans} \ (\text{snd} \ ?M)))\).
from PLimit \([X \ uY] \text{Sup uInf have} \ P\ ((\text{Sup-trans} \ (\text{fst} \ ?M)) \ ((?Y \ (\text{Inf-utrans} \ (\text{snd} \ ?M)))\))
by(iprover intro:IH)
moreover have \(\text{feasible} \ ((\text{Sup-trans} \ (\text{fst} \ ?M)))\) by(rule [X]
moreover have \(\Lambda P. \ \text{unitary} \ P \implies \text{unitary} \ ((?Y \ (\text{Inf-utrans} \ (\text{snd} \ ?M))) \ P)\)
by(auto intro:uY uInf)
moreover note le-lfp ge-gfp
ultimately have pair-in: \((\text{?X} \ (\text{Sup-trans} \ (\text{fst} \ ?M)), \ ?Y \ (\text{Inf-utrans} \ (\text{snd} \ ?M)))\) \(?M\)
by(simp)

have \(\text{?X} \ (\text{Sup-trans} \ (\text{fst} \ ?M)) \in \text{fst} \ ?M\)
by(rule imagef[\ OF pair-in, of fst, simplified])
hence le-trans \((\text{?X} \ (\text{Sup-trans} \ (\text{fst} \ ?M)))\) \((\text{Sup-trans} \ (\text{fst} \ ?M))\)
proof(rule le-transf[\ OF \ Sup-trans-upper2\ where \ t=?X \ (\text{Sup-trans} \ (\text{fst} \ ?M)) \and \ S=\text{fst} \ ?M]])
fix \(P::\)’s expect
assume sP: sound \(P\)
thus \( \text{nng } P \) 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{nng } Q \land \text{bounded-by } (\text{bound-of } \( P \)) \( Q \). \rightarrow \text{nng } (u \ Q) \land \text{bounded-by } (\text{bound-of } \( P \)) \( (u \ Q) \)

by (auto)

qed

hence le-trans (lfp-trans \( ?X \)) (Sup-trans (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 \( ?Y \) (Inf-utrans (snd ' ?M)) \( \in \) snd ' ?M

by (rule imageI [OF pair-in, of snd, simplified])

hence le-utrans (Inf-utrans (snd ' ?M)) (\( ?Y \) (Inf-utrans (snd ' ?M)))

by (intro Inf-utrans-lower, auto)

hence le-utrans (Inf-utrans (snd ' ?M)) (gfp-trans \( ?Y \))

by (blast intro: gfp-trans-upperbound uInf)

with fu-gfp have equ: equiv-utrans (Inf-utrans (snd ' ?M)) (gfp-trans \( ?Y \))

by (auto intro: le-utrans-antisym)

from PLimit eqt equ show \( P \) (lfp-trans \( ?X \)) (gfp-trans \( ?Y \)) by (rule P-equiv)

qed

4.4.1 The Limit of Iterates

The iterates of a loop are its sequence of finite unrollings. We show shortly that this converges on the least fixed point. This is enormously useful, as we can appeal to various properties of the finite iterates (which will follow by finite induction), which we can then transfer to the limit.

definition iterates :: 's prog => ('s => bool) => nat => 's trans

where iterates body G i = ((\( \lambda x. \) wp (body ;; Embed x « \( G_s \oplus \) Skip)) ^^ i) \( (\lambda P \ s. \ 0) \)

lemma iterates-0[simp]:

iterates body G 0 = (\( \lambda P \ s. \ 0) \)

by (simp add: iterates-def)

lemma iterates-Suc[simp]:

iterates body G (Suc i) = wp (body ;; Embed (iterates body G i) « \( G_s \oplus \) Skip)

by (simp add: iterates-def)

All iterates are healthy.

lemma iterates-healthy:

healthy (wp body) \( \rightarrow \) 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 $(\text{wp body})$
shows le-trans $(\text{iterates body } G \ i) \ (\text{iterates body } G \ (\text{Suc } i))$
proof(induct $i$)
show le-trans $(\text{iterates body } G \ 0) \ (\text{iterates body } G \ (\text{Suc } 0))$
proof(simp add: iterates-def, rule le-transI)
  fix $P$::'s expect
  assume sound $P$
  with $hb$ have sound $(\text{wp body ;; Embed } (\lambda P s. 0) \ « G \oplus \text{Skip} ) \ P)$
  by(auto intro: wp-loop-step-sound)
thus $\lambda s. 0 \vdash \vdash \text{wp body ;; Embed } (\lambda P s. 0) \ « G \oplus \text{Skip} ) \ P$
by(auto)
qed
fix $i$
assume IH: le-trans $(\text{iterates body } G \ i) \ (\text{iterates body } G \ (\text{Suc } i))$
have equiv-trans $(\text{iterates body } G \ (\text{Suc } i)) \ (\text{wp body ;; Embed } (\text{iterates body } G \ i) \ « G \oplus \text{Skip} ) )$
by(simp)
also from iterates-healthy[OF $hb$]
have le-trans ... $(\text{wp body ;; Embed } (\text{iterates body } G \ (\text{Suc } i)) \ « G \oplus \text{Skip} ) )$
by(blast intro: wp-loop-step-mono[OF $hb$ IH])
also have equiv-trans ... $(\text{iterates body } G \ (\text{Suc } (\text{Suc } i)))$
by(simp)
finally show le-trans $(\text{iterates body } G \ (\text{Suc } i)) \ (\text{iterates body } G \ (\text{Suc } (\text{Suc } i)))$.
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 $(\text{wp body})$
  shows bounded-by $b$ $(\text{wp body ;; Embed } t \ « G \oplus \text{Skip} ) \ Q )$
proof(rule bounded-byI, simp add: wp-eval)
  fix $s$::'s
  from $nQ$ $bQ$ have $sQ$: sound $Q$ by(auto)
  with $bQ$ $ht$ have sound $(t \ Q)$ bounded-by $b$ $(t \ Q)$ by(auto)
  with $hb$ have bounded-by $b$ $(\text{wp body } (t \ Q))$ by(auto)
  with $bQ$ have wp body $(t \ Q) \ s \leq b \ Q \ s \leq b$ by(auto)
  hence $(G > s * \text{wp body } (t \ Q) \ s + (1-\text{«}G\text{»} \ s) * Q \ s \leq (G > s + b + (1-\text{«}G\text{»} \ s) * b)$
  by(auto intro:add-mono mult-left-mono)
  also have ... $= b$ by(simp add: algebra-simps)
  finally show $(G > s * \text{wp body } (t \ Q) \ s + (1-\text{«}G\text{»} \ s) * Q \ s \leq b$.
qed

This is the key result: The loop is equivalent to the supremum of its iterates.
This proof follows the pattern of lemma continuous_lfp in HOL/Library/Continuity.

lemma lfp-iterates:
CHAPTER 4. THE PGCL LANGUAGE

136

fixes body::’s prog
assumes hb: healthy (wp body)
and cb: bd-cts (wp body)
shows equiv-trans (wp (do G → body od)) (Sup-trans (range (iterates body G))
(is equiv-trans ?X ?Y)
proof (rule le-trans-antisym)
let \( ?F = \lambda x.\ wp (body ;; Embed x « G » ⊕ Skip) \)
let \( ?bot = \lambda (P::’s ⇒ real) s::’s. 0::real \)
have \( HF: \bigwedge i.\ healthy ((?F ^^ i) ?bot) \)
proof
  fix \( i \) from hb show ((thesis \( i \)) ?bot)
  by (induct \( i \), simp-all add:healthy-intros)
qed

from iterates-healthy[OF hb]
have \( \bigwedge i.\ feasible (iterates body G i) \)
by (auto)
hence (Sup: feasible (Sup-trans (range (iterates body G))))
by (auto intro:feasible-Sup-trans)

\{  
  fix \( i \)
  have le-trans ((?F ^^ i) ?bot) \( ?X \)
  proof (induct \( i \))
    show le-trans ((?F ^^ 0) ?bot) \( ?X \)
    proof (simp, intro le-transI)
      fix P::’s expect
      assume sound P
      with hb healthy-wp-loop
      have sound (wp (\( \mu x. \) body ;; x « G » ⊕ Skip) P)
      by (auto)
      thus \( \lambda s. 0 ⊢ wp (\( \mu x. \) body ;; x « G » ⊕ Skip) P \)
      by (auto)
    qed
  qed
  fix \( i \)
  assume IH: le-trans ((?F ^^ i) ?bot) \( ?X \)
  have equiv-trans ((?F ^^ Suc \( i \)) ?bot) ((?F ((?F ^^ i) ?bot)) \( \sum \))
  by (simp)
  also have le-trans ... (?F \( ?X \))
  proof (rule wp-loop-step-mono[OF hb IH])
    fix P::’s expect
    assume sP: sound P
    with hb healthy-wp-loop
    show sound (wp (\( \mu x. \) body ;; x « G » ⊕ Skip) P)
    by (auto)
    from sP show sound ((?F ^^ i) ?bot P)
    by (rule healthy-sound[OF OF HF])
  qed
  also {
    from hb have X: le-trans (wp (body ;; Embed (\( \lambda P s.\ bound-of P \) « G » ⊕)}
4.4. CONTINUITY AND INDUCTION FOR LOOPS

\[\lambda P \, s. \, \text{bound-of } P\]

\[
\text{by (intro leI, simp add: wp-eval, auto intro: lfp-loop-fp unfolded neg-embed)}
\]

\[\text{have equiv-trans (} ?F \, ?X \, ?X\]

\[\text{apply (simp only: wp-eval)}\]

\[\text{by (intro iffD1 [OF equiv-trans-comm, OF lfp-trans-unfold]}
\]

\[\text{wp-loop-step-mono [OF hb] wp-loop-step-sound [OF hb], (blast [rule X+])}
\]

\[\text{finally show le-trans ((} ?F \, ^{\to} (\text{Suc } i) \, ?bot) \, ?X .}
\]

\[\text{qed}
\]

\[\text{hence } \forall i. \, \text{le-trans (iterates body } G \, i) \, (\text{wp do } G \, \to \, \text{body od)}
\]

\[\text{by (simp add: itera-def)}\]

\[\text{thus le-trans } ?Y \, ?X\]

\[\text{by (auto intro !: le-transI [OF Sup-trans-least2 sound-nneg healthy-sound [OF itera-healthy hb] healthy-bounded-byD [OF itera-healthy hb] healthy-sound [OF healthy-wp-loop hb]}
\]

\[\text{show le-trans } ?X \, ?Y\]

\[\text{proof (simp only: wp-eval, rule lfp-trans-lowerbound)}\]

\[\text{from hb cb have bd-cts-tr } ?F \text{ by (rule cts-wp-loopstep)}\]

\[\text{with itera-increasing [OF hb] itera-healthy [OF hb]}\]

\[\text{have equiv-trans (} ?F \, ?Y \, (\text{Sup-trans (range (} ?F \, o \, (\text{iterates body } G))))\]

\[\text{by (auto intro: healthy-feasibleD bd-cts-trD cong del: image-cong-simp)}\]

\[\text{also have le-trans (Sup-trans (range (} ?F \, o \, (\text{iterates body } G))) ?Y}\]

\[\text{proof (rule le-transI)}\]

\[\text{fix } P::'s \, \text{expect and } i\]

\[\text{assume } sP: \, \text{sound } P\]

\[\text{show (Sup-trans (range (} ?F \, o \, (\text{iterates body } G))) P \, \to \, ?Y \, P}\]

\[\text{proof (rule Sup-trans-least2, clarsimp)}\]

\[\text{show } \forall u \in \text{range (} (\lambda x. \, \text{wp (body } :: \, \text{Embed } x \, « \, G \, ⊕ \, \text{Skip}))) \, \circ \, \text{iterates body } G\).
\]

\[\forall R. \, \text{nneg } R \land \text{bounded-by (bound-of } P) \, R \to \text{nneg (} u \, R) \land \text{bounded-by (bound-of } P) \, (u \, R)\]

\[\text{proof (clarsimp, intro conjf)}\]

\[\text{fix } Q::'s \, \text{expect and } i\]

\[\text{assume } nQ: \, \text{nneg } Q \, \text{and } bQ: \, \text{bounded-by (bound-of } P) \, Q\]

\[\text{hence sound } Q \text{ by (auto)}\]

\[\text{moreover from itera-healthy [OF hb]}\]

\[\text{have } \forall P. \, \text{sound } P \to \text{sound (iterates body } G \, i \, P) \text{ by (auto)}\]

\[\text{moreover note } hb\]

\[\text{ultimately have sound (} \text{wp (body } :: \, \text{Embed } (\text{iterates body } G \, i) \, « \, G \, ⊕ \, \text{Skip}) \, Q)\]

\[\text{by (iprover intro: wp-loop-step-sound)}\]

\[\text{thus nneg (} \text{wp (body } :: \, \text{Embed } (\text{iterates body } G \, i) \, « \, G \, ⊕ \, \text{Skip}) \, Q)\]

\[\text{by (auto)}\]
from nQ bQ iterates-healthy[OF hb] hb
    show bounded-by (bound-of P) (wp (body ;; Embed (iterates body G i)) « G » Skip) Q
      by (rule wp-loop-step-bounded)
    qed

from sP show nneg P bounded-by (bound-of P) P by (auto)

next
    fix Q: "s expect
    assume nQ: nneg Q and bQ: bounded-by (bound-of P) Q
    hence sound Q by (auto)
    with fSup have sound (Sup-trans (range (iterates body G)) Q) by (auto)
    thus nneg (Sup-trans (range (iterates body G)) Q) by (auto)

    fix i
    show wp (body ;; Embed (iterates body G i) « G » Skip) Q ⊢
      Sup-trans (range (iterates body G)) Q

    proof (rule Sup-trans-upper2 [OF - - nQ bQ])
        from iterates-healthy[OF hb]
    show ∀ u ∈ range (iterates body G).
        ∀ R. nneg R ∧ bounded-by (bound-of P) R →
        nneg (u R) ∧ bounded-by (bound-of P) (u R)
      by (auto)

    have wp (body ;; Embed (iterates body G i) « G » Skip) = iterates body G (Suc i)
      by (simp)
    also have ... ∈ range (iterates body G)
      by (blast)
    finally show wp (body ;; Embed (iterates body G i) « G » Skip) ∈
      range (iterates body G).
    qed
    qed
    qed

finally show le-trans (?F ?Y) ?Y.

fix P: "s expect
    assume sound P
    with fSup show sound (?Y P) by (auto)
  qed
  qed

Therefore, evaluated at a given point (state), the sequence of iterates gives a sequence of real values that converges on that of the loop itself.

corollary loop-iterates:
  fixes body: "s prog
  assumes hb: healthy (wp body)
  and chc: bd-cts (wp body)
  and sP: sound P
  shows (λi. iterates body G i P s) −−−−→ wp (do G → body od) P s
  proof –
let $\mathcal{X} = \{ s \mid f \in \{ t P \mid t \in \text{range (iterates body } G) \} \}$

have closure-Sup: $\text{Sup } \mathcal{X} \in \text{closure } \mathcal{X}$
proof (rule closure-contains-Sup, simp, clarsimp)
  fix $i$
  from $sP$ have bounded-by (bound-of $P$) $P$ by(auto)
  with iterates-healthy[OF $hb$] $sP$ have $\forall j. \text{bounded-by (bound-of } P) \text{ (iterates body } G j P)$
  by(auto)
  thus iterates body $G i P s \leq \text{bound-of } P$ by(auto)
qed

have $(\lambda i. \text{iterates body } G i P s) \longrightarrow \text{Sup } \{ f s \mid f \in \{ t P \mid t \in \text{range (iterates body } G) \} \}$
proof (rule LIMSEQ-I)
  fix $r :: \text{real}$
  assume posr: $0 < r$
  with closure-Sup obtain $y$ where $y \in \mathcal{X} \text{ and } \forall y. \text{dist } y (\text{Sup } \mathcal{X}) < r$
  by (simp only: closure-approachable, blast)
  from $y$ obtain $i$ where $y_i: y = \text{iterates body } G i P s$ by (auto)
  {
    fix $j$
    have $i \leq j \longrightarrow \text{le-trans (iterates body } G i) (\text{iterates body } G j)$
    proof (induct $j$, simp, clarify)
      fix $k$
      assume IH: $i \leq k \longrightarrow \text{le-trans (iterates body } G i) (\text{iterates body } G k)$
      and le: $i \leq \text{Suc } k$
      show $\text{le-trans (iterates body } G i) (\text{iterates body } G (\text{Suc } k))$
      proof (cases $i = \text{Suc } k$, simp)
        assume $i \neq \text{Suc } k$
        with le have $i \leq k$ by (auto)
        with IH have $\text{le-trans (iterates body } G i) (\text{iterates body } G k)$ by (auto)
        also note iterates-increasing[OF $hb$]
        finally show $\text{le-trans (iterates body } G i) (\text{iterates body } G (\text{Suc } k))$ .
      qed
    qed
  }
  with $sP$ have $\forall j \geq i. \text{iterates body } G i P s \leq \text{iterates body } G j P s$
  by (auto)
  moreover {
    from $sP$ have bounded-by (bound-of $P$) $P$ by (auto)
    with iterates-healthy[OF $hb$] $sP$ have $\forall j. \text{bounded-by (bound-of } P) \text{ (iterates body } G j P)$
    by (auto)
    hence $\forall j. \text{iterates body } G j P s \leq \text{bound-of } P$ by (auto)
    hence $\forall j. \text{iterates body } G j P s \leq \text{Sup } \mathcal{X}$
    by (intro cSup-upper bdd-aboveI, auto)
  }
  ultimately have $\forall j. i \leq j \Rightarrow$
    norm (iterates body $G j P s - \text{Sup } \mathcal{X}$) $\leq$
    norm (iterates body $G i P s - \text{Sup } \mathcal{X}$)
by(auto)
also from ey yit have norm (iterates body G i P s − Sup ?X) < r
by(simp add:dist-real-def)
finally show ∃ no. ∀ n≥ no. norm (iterates body G n P s −

Sup {f s | f ∈ {t P | t ∈ range (iterates body G)}})
< r
by(auto)
qed
moreover from hb cb sP have wp do G −→ body od P s = Sup-trans (range (iterates body G))
by(simp add: Sup-trans-def)
made have ... = Sup {f s | f ∈ {t P | t ∈ range (iterates body G)}}
by(simp add: Sup-trans-def)
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 (λ(n::nat) (s::′s). 0::real) = {λs. 0::real}
by(auto)
thus bd-cts (λP (s::′s). 0)
by(intro bd-ctsI, simp add:o-def Sup-exp-def)
next
fix i
assume IH: bd-cts (iterates body G i)
thus bd-cts (wp (body ;; Embed (iterates body G i) ⊕ 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 −→ body od)
proof(rule bd-ctsI)
fix M::nat ⇒ ′s expect and b::real
assume chain: ∨ i. M i ⊢ M (Suc i)
and sM: ∨ i. sound (M i)
and bM: ∨ i. bounded-by b (M i)
from sM bM iterates-healthy[OF hb]
4.4. CONTINUITY AND INDUCTION FOR LOOPS

have \( \forall j. i. \) bounded-by \( b \) (iterates body \( G \) \( i \) \( (M \ j) \)) by(blast)

hence \( iB: \) \( \forall j i s. \) iterates body \( G \) \( i \) \( (M \ j) \) \( s \leq b \) by(auto)

from \( sM \ bM \) have \( sSup: \) sound \((\text{Sup-exp \ (range \( M \))})\)

by(auto intro:Sup-exp-sound)

with \( \text{ifp-iterates}(\text{OF \( hb \ cb \)}) \)

have \( \wp do \ G \rightarrow body od \ (\text{Sup-exp \ (range \( M \))}) = \)

\( \text{Sup-trans} \ (\text{range \ (iterates body \( G \))}) \ (\text{Sup-exp \ (range \( M \))}) \)

by(simp add:equtrans-transD)

also \( \{ \)

from \( \text{chain \( sM \ bM \)} \)

have \( \forall i. \) iterates body \( G \) \( i \) \( (\text{Sup-exp \ (range \( M \))}) = \text{Sup-exp \ (range \ (iterates body \( G \) \( i \) \( o \) \( M \))}) \)

by(blast intro:bd-ctsD cts-iterates)

hence \( \{ t (\text{Sup-exp \ (range \( M \))} \ | \ t. \ t \in \text{range \ (iterates body \( G \))}) = \)

\( \{\text{Sup-exp \ (range \ (t \ o \ M))} \ | \ t. \ t \in \text{range \ (iterates body \( G \))}\} \)

by(auto intro:sym)

hence \( \text{Sup-trans} \ (\text{range \ (iterates body \( G \))}) \ (\text{Sup-exp \ (range \( M \))}) = \)

\( \text{Sup-exp} \ (\text{Sup-exp \ (range \ (t \ o \ M))} \ | \ t. \ t \in \text{range \ (iterates body \( G \))}) \)

by(simp add:Sup-trans-def)

also \( \{ \)

have \( \forall \lambda s. \) \( \{\lambda f. \exists t. \ f = (\lambda s. \text{Sup} \ \{\lambda s f f. \ f \in \text{range \ (t \ o \ M)}\}) \wedge \)

\( t \in \text{range \ (iterates body \( G \))} = \)

\( \text{range} \ (\lambda i. \text{Sup} \ (\lambda i. \text{iterates body \( G \) \( i \) \( (M \ j) \ s)}) \)

(is \( \forall s. \) \( ?X s = ?Y s \))

proof(intro antisym subsetI)

fix \( s x \)

assume \( x \in ?X s \)

then obtain \( t \) where \( \text{rwx:} \) \( x = \text{Sup} \ \{\lambda s f f. \ f \in \text{range \ (t \ o \ M)}\} \)

\( t \in \text{range \ (iterates body \( G \))} \)

by(auto)

then obtain \( \lambda i \) where \( \lambda t = \text{iterates body \( G \) \( i \) \( by(auto) \)

with \( \text{rwx have} \) \( x = \text{Sup} \ \{\lambda s f f. \ f \in \text{range} \ (\lambda i. \text{iterates body \( G \) \( i \) \( (M \ j) \))}\} \)

\( t \in \text{range \ (iterates body \( G \))} \)

by(auto)

moreover have \( \{\lambda s f f. \ f \in \text{range} \ (\lambda j. \text{iterates body \( G \) \( i \) \( (M \ j) \})) = \)

\( \text{range} \ (\lambda j. \text{iterates body \( G \) \( i \) \( (M \ j) \ s)}) \)

by(auto)

ultimately have \( x = \text{Sup} \ (\lambda i. \text{iterates body \( G \) \( i \) \( (M \ j) \))} \)

by(simp)

thus \( x \in \text{range} \ (\lambda i. \text{Sup} \ (\lambda j. \text{iterates body \( G \) \( i \) \( (M \ j) \ s)}) \)

by(auto)

next

fix \( s x \)

assume \( x \in ?Y s \)

then obtain \( \lambda i \) where \( \lambda A : x = \text{Sup} \ (\lambda i. \text{iterates body \( G \) \( i \) \( (M \ j) \ s)}) \)

by(auto)

have \( \forall \lambda s. \) \( \{\lambda f f. \ f \in \text{range} \ (\lambda j. \text{iterates body \( G \) \( i \) \( (M \ j) \))} = \)

\( \text{range} \ (\lambda j. \text{iterates body \( G \) \( i \) \( (M \ j) s)}) \)

by(auto)

hence \( B: \) \( \forall \lambda s. \text{Sup} \ (\lambda j. \text{iterates body \( G \) \( i \) \( (M \ j) \ s))) \)


(\textfamilydefault{\textsf{(\lambda s. \textsf{Sup}\ \{f \ s \mid f. f \in \textbf{range} (\textbf{iterates body} G \ i \ o M)\})})}
\textbf{by}(\textbf{simp add:o-def})

\textbf{have} C: \textbf{iterates body} G \ i \in \textbf{range} (\textbf{iterates body} G) \ \textbf{by}(\textbf{auto})

\textbf{have} \exists f. \ x = \ f \ s \ \wedge 
(\exists t. \ f = (\textsf{\lambda s. \textbf{Sup}\ \{f \ s \mid f. f \in \textbf{range} (t \ o M)\}) \ \wedge 
 t \in \textbf{range} (\textbf{iterates body} G))
\textbf{by}(\textbf{iprover intro:A B C})
\textbf{thus} \ x \in \ ?X \ s \ \textbf{by}(\textbf{simp})
\textbf{qed}

\textbf{hence} \textbf{Sup-exp}\ \{(\textbf{Sup-exp}\ \{\textbf{range} (t \ o M)\}) \ \mid t. t \in \textbf{range} (\textbf{iterates body} G)\} =
(\textsf{\lambda s. \textbf{Sup}\ \{\textbf{range} (\textsf{\lambda i. \textbf{Sup}\ \{\textbf{range} (\textsf{\lambda j. \textbf{iterates body} G \ i \ (M \ j) \ s)\})\})\})\)
\textbf{by}(\textbf{simp add:Sup-exp-def})

\texttt{)}
\textbf{also have} \textsf{(\lambda s. \textbf{Sup}\ \{\textbf{range} (\textsf{\lambda i. \textbf{Sup}\ \{\textbf{range} (\textsf{\lambda j. \textbf{iterates body} G \ i \ (M \ j) \ s)\})\})\}) =
\textsf{(\lambda s. \textbf{Sup}\ \{\textbf{range} (\textsf{\lambda(i,j). \textbf{iterates body} G \ i \ (M \ j) \ s)\})\})
(\texttt{is} ?X = ?Y)
\textbf{proof}(\textbf{rule ext, rule antisym})
\textbf{fix} \ s\::\ s\in \texttt{nat}
\textbf{show} \ ?Y \ s \leq \ ?X \ s
\textbf{proof}(\textbf{rule cSup-least, blast, clarify})
\textbf{fix} \ i \ j\::\ \texttt{nat}
\textbf{from} \ iB \ \textbf{have} \textbf{iterates body} G \ i \ (M \ j) \ s \leq \textbf{Sup} \ \{\textbf{range} (\textsf{\lambda j. \textbf{iterates body} G \ i \ (M \ j) \ s)\})
\textbf{by}(\textbf{intro cSup-upper bdd-aboveI, auto})
\textbf{also from} \ iB \ \textbf{have} ... \leq \textbf{Sup} \ \{\textbf{range} (\textsf{\lambda j. \textbf{iterates body} G \ i \ (M \ j) \ s)\})\)
\textbf{by}(\textbf{intro cSup-upper cSup-least bdd-aboveI, (blast intro:cSup-least)+})
\textbf{finally show} \textbf{iterates body} G \ i \ (M \ j) \ s \leq 
\textbf{Sup} \ \{\textbf{range} (\textsf{\lambda j. \textbf{iterates body} G \ i \ (M \ j) \ s)\})\} .
\textbf{qed}

\textbf{have} \ \forall i \ j. \textbf{iterates body} G \ i \ (M \ j) \ s \leq 
\textbf{Sup} \ \{\textbf{range} (\textsf{\lambda(i,j). \textbf{iterates body} G \ i \ (M \ j) \ s)\})
\textbf{by}(\textbf{rule cSup-upper, auto intro:iB})
\textbf{thus} \ ?X \ s \leq \ ?Y \ s
\textbf{by}(\textbf{intro cSup-least, blast, clarify, simp, blast intro:cSup-least})
\textbf{qed}

\textbf{also have} \ ... = \ (\textsf{\lambda s. \textbf{Sup}\ \{\textbf{range} (\textsf{\lambda j. \textbf{Sup}\ \{\textbf{range} (\textsf{\lambda i. \textbf{iterates body} G \ i \ (M \ j) \ s)\})\})\})\)
(\texttt{is} ?X = ?Y)
\textbf{proof}(\textbf{rule ext, rule antisym})
\textbf{fix} \ s\::\ s\in \texttt{nat}
\textbf{have} \ \forall i \ j. \textbf{iterates body} G \ i \ (M \ j) \ s \leq 
\textbf{Sup} \ \{\textbf{range} (\textsf{\lambda(i,j). \textbf{iterates body} G \ i \ (M \ j) \ s)\})
\textbf{by}(\textbf{rule cSup-upper, auto intro:iB})
\textbf{thus} \ ?Y \ s \leq \ ?X \ s
\textbf{by}(\textbf{intro cSup-least, blast, clarify, simp, blast intro:cSup-least})
\textbf{show} \ ?X \ s \leq ?Y \ s
\textbf{proof}(\textbf{rule cSup-least, blast, clarify)}
4.4. CONTINUITY AND INDUCTION FOR LOOPS

\[
\begin{align*}
\text{fix } i &:: \text{nat} \\
\text{from } iB &\text{ have iterates body } G \ i \ (M \ j)\ s \leq \ \text{Sup} \ (\text{range} \ (\lambda \ i. \ \text{iterates body} \ G \ i \ (M \ j) \ s)) \\
&\quad \text{by}(\text{intro } cSup-upper \ \text{bdd-aboveI}, \ \text{auto}) \\
&\quad \text{also from } iB \ \text{have } \ldots \leq \ \text{Sup} \ (\text{range} \ (\lambda \ j. \ \text{Sup} \ (\text{range} \ (\lambda \ i. \ \text{iterates body} \ G \ i \ (M \ j) \ s)))) \\
&\quad \quad \text{by}(\text{intro } cSup-upper \ cSup-least \ \text{bdd-aboveI}, \ \text{blast}, \ \text{blast intro;cSup-least}) \\
&\quad \quad \quad \text{finally show iterates body } G \ i \ (M \ j)\ s \leq \\
&\quad \quad \quad \quad \text{Sup} \ (\text{range} \ (\lambda \ j. \ \text{Sup} \ (\text{range} \ (\lambda \ i. \ \text{iterates body} \ G \ i \ (M \ j) \ s)))) \\
&\quad \quad \quad \text{by}(\text{auto}) \\
&\quad \quad \quad \text{qed} \\
&\text{ ultimately show } x = \ \text{Sup} \ (\text{range} \ (\lambda \ i. \ \text{iterates body} \ G \ i \ (M \ j) \ s)) \\
&\quad \quad \quad \text{by}(\text{auto}) \\
&\text{next} \\
&\quad \text{fix } s \ x \\
&\quad \text{assume } x \in ?X s \\
&\quad \text{then obtain } j \ \text{where } \text{rux: } x = \ \text{Sup} \ (\text{range} \ (\lambda \ i. \ \text{iterates body} \ G \ i \ (M \ j) \ s)) \\
&\quad \quad \text{by}(\text{auto}) \\
&\text{also} \\
&\quad \text{have } \exists s. \ \text{range} \ (\lambda \ j. \ \text{Sup} \ (\text{range} \ (\lambda \ i. \ \text{iterates body} \ G \ i \ (M \ j) \ s))) = \\
&\quad \quad \{ f s \ | \ f \in \text{range} \ ((\lambda \ P \ s. \ \text{Sup} \ \{ f s \ | \ \exists t. \ f = t \ (M \ j) \ \land \ t \in \text{range} \ (\text{iterates body} \ G)\}) \circ M\} \ (\text{is } \exists s. \ ?X s = ?Y s) \\
&\quad \text{proof}(\text{intro } \text{antisym } \text{subsetI}) \\
&\quad \quad \text{fix } s \ x \\
&\quad \quad \text{assume } x \in ?X s \\
&\quad \quad \text{then obtain } P \ \text{where } \text{rux: } x = P s \\
&\quad \quad \quad \text{and } \text{Pin: } P \in \text{range} \ ((\lambda \ P \ s. \ \text{Sup} \ \{ f s \ | \ \exists t. \ f = t \ (M \ j) \ \land \ t \in \text{range} \ (\text{iterates body} \ G)\}) \circ M) \\
&\quad \quad \quad \text{by}(\text{auto}) \\
&\quad \quad \quad \text{then obtain } j \ \text{where } P = (\lambda s. \ \text{Sup} \ \{ f s \ | \ \exists t. \ f = t \ (M \ j) \ \land \\
&\quad \quad \quad \quad t \in \text{range} \ (\text{iterates body} \ G)\}) \\
&\quad \quad \quad \text{by}(\text{auto}) \\
&\quad \text{also} \\
&\quad \quad \text{have } \exists s. \ \{ f s \ | \ \exists t. \ f = t \ (M \ j) \ \land \ t \in \text{range} \ (\text{iterates body} \ G)\} = \\
&\quad \quad \quad \text{range} \ (\lambda \ i. \ \text{iterates body} \ G \ i \ (M \ j) \ s) \ \text{by}(\text{auto}) \\
&\quad \quad \quad \text{hence } (\lambda s. \ \text{Sup} \ \{ f s \ | \ \exists t. \ f = t \ (M \ j) \ \land \ t \in \text{range} \ (\text{iterates body} \ G)\}) = \\
&\quad \quad \quad (\lambda s. \ \text{Sup} \ (\text{range} \ (\lambda \ i. \ \text{iterates body} \ G \ i \ (M \ j) \ s))) \\
&\quad \quad \quad \text{by}(\text{simp}) \\
&\quad \text{finally have } x = \ \text{Sup} \ (\text{range} \ (\lambda \ i. \ \text{iterates body} \ G \ i \ (M \ j) \ s)) \\
&\quad \quad \quad \text{by}(\text{simp add: rux})
\end{align*}
\]
thus \( x \in \mathcal{X} \) by (simp)

qed

hence \( (\lambda s. \text{Sup} ((\lambda j. \text{Sup} (\text{range} (\lambda i. \text{iterates body } G) i (M j) s))) o M)) = \text{Sup-exp} (\text{range} (\text{Sup-trans} (\text{iterates body } G) o M)) \)

by (simp add: Sup-exp-def Sup-trans-def cong del: SUP-cong-simp)

\)

also have \( \text{Sup-exp} (\text{range} (\text{Sup-trans} (\text{iterates body } G) o M)) = \text{Sup-exp} (\text{range} (\text{wp do } G \rightarrow body o M)) \)

by (simp add: o-def equiv-transD [OF lfp-iterates, OF hb cb, OF sM])

finally show \( \text{wp do } G \rightarrow body o M = \text{Sup-exp} (\text{range} (\text{wp do } G \rightarrow body o M)) \).

qed

lemmas cts-intros =

cts-wp-Abort cts-wp-Skip
cts-wp-Seq cts-wp-PC
cts-wp-DC cts-wp-Embed
cts-wp-Apply cts-wp-SetDC
cts-wp-SetPC cts-wp-Bind
cts-wp-repeat

end

4.5 Sublinearity

theory Sublinearity imports Embedding Healthiness LoopInduction begin

4.5.1 Nonrecursive Primitives

Sublinearity of non-recursive programs is generally straightforward, and follows from the algebraic properties of the underlying operations, together with healthiness.

lemma sublinear-wp-Skip:

sublinear (wp Skip)

by (auto simp: wp-eval)

lemma sublinear-wp-Abort:

sublinear (wp Abort)

by (auto simp: wp-eval)

lemma sublinear-wp-Apply:

sublinear (wp (Apply f))

by (auto simp: wp-eval)

lemma sublinear-wp-Seq:

fixes x::'s prog

assumes slx: sublinear (wp x) and sly: sublinear (wp y)

and hx: healthy (wp x) and hy: healthy (wp y)
4.5. SUBLINEARITY

shows sublinear \((wp \ (x :: y))\)
proof\(\text{rule sublinearI, simp add:wp-eval}\)
fix \(P::'s \Rightarrow \text{real}\) and \(Q::'s \Rightarrow \text{real}\) and \(s::'s\)
and \(a::real\) and \(b::real\) and \(c::real\)
assume \(sP: \text{sound } P\) and \(sQ: \text{sound } Q\)
and \(\text{nna: } 0 \leq a\) and \(\text{nnb: } 0 \leq b\) and \(\text{nnc: } 0 \leq c\)

with \(\text{slx} \; \text{hy} \; \text{have} \; a \ast \text{wp } x \; (\text{wp } y \; P) \; s + b \ast \text{wp } x \; (\text{wp } y \; Q) \; s \odot c \leq \text{wp } x \; (\lambda s. \; a \ast \text{wp } y \; P \; s + b \ast \text{wp } y \; Q \; s \odot c) \; s\)
by\((\text{blast intro:sublinearD})\)
also \{\}
from \(sP \; sQ \; \text{nna} \; \text{nnb} \; \text{nnc} \; \text{sly}\)
have \(\text{\forall s. } a \ast \text{wp } y \; P \; s + b \ast \text{wp } y \; Q \; s \odot c \leq \text{wp } y \; (\lambda s. \; a \ast \text{wp } y \; P \; s + b \ast \text{wp } y \; Q \; s \odot c) \; s\)
by\((\text{blast intro:sublinearD})\)
moreover from \(sP \; sQ \; \text{by}\)
have \(\text{sound } (\text{wp } y \; P)\) and \(\text{sound } (\text{wp } y \; Q)\) by\((\text{auto})\)
moreover with \(\text{nna} \; \text{nnb} \; \text{nnc}\)
have \(\text{sound } (\lambda s. \; a \ast \text{wp } y \; P \; s + b \ast \text{wp } y \; Q \; s \odot c)\)
by\((\text{auto intro!:sound-intros tminus-sound})\)
moreover from \(sP \; sQ \; \text{nna} \; \text{nnb} \; \text{nnc}\)
have \(\text{sound } (\lambda s. \; a \ast \text{wp } y \; P \; s + b \ast \text{wp } y \; Q \; s \odot c)\)
by\((\text{auto intro!:sound-intros tminus-sound})\)
moreover with \(\text{by have sound } (\text{wp } y \; (\lambda s. \; a \ast \text{wp } y \; P \; s + b \ast \text{wp } y \; Q \; s \odot c))\)
by\((\text{blast})\)
ultimately
have \(\text{wp } x \; (\lambda s. \; a \ast \text{wp } y \; P \; s + b \ast \text{wp } y \; Q \; s \odot c) \; s \leq \text{wp } x \; (\text{wp } y \; (\lambda s. \; a \ast \text{wp } y \; P \; s + b \ast \text{wp } y \; Q \; s \odot c)) \; s\)
by\((\text{blast intro!:le-funD[OF mono-transD[OF healthy-monoD[OF \text{OF hy}]])})\)
\}
finally show \(a \ast \text{wp } x \; (\text{wp } y \; P) \; s + b \ast \text{wp } x \; (\text{wp } y \; Q) \; s \odot c \leq \text{wp } x \; (\text{wp } y \; (\lambda s. \; a \ast \text{wp } y \; P \; s + b \ast \text{wp } y \; Q \; s \odot c)) \; s\).
qed

lemma sublinear-wp-PC:
fixes \(x::'s \; \text{prog}\)
assumes slx: \(\text{sublinear } (wp \; x)\) and \(\text{sly: } \text{sublinear } (wp \; y)\)
and \(uP: \text{unitary } P\)
shows \(\text{sublinear } (wp \; (x \; p\oplus \; y))\)
proof\(\text{rule sublinearI, simp add:wp-eval}\)
fix \(R::'s \Rightarrow \text{real}\) and \(Q::'s \Rightarrow \text{real}\) and \(s::'s\)
and \(a::\text{real}\) and \(b::\text{real}\) and \(c::\text{real}\)
assume slr: \(\text{sound } R\) and \(sQ: \text{sound } Q\)
and \(\text{nna: } 0 \leq a\) and \(\text{nnb: } 0 \leq b\) and \(\text{nnc: } 0 \leq c\)

have \(a \ast (P \; s \ast \text{wp } x \; Q \; s + (1 - P) \ast \text{wp } y \; Q \; s) + \)
\((P \; s \ast a \ast \text{wp } x \; R \; s + (1 - P) \ast \text{wp } y \; R \; s) \odot c = \)
\((P \; s \ast b \ast \text{wp } x \; Q \; s + (1 - P) \ast a \ast \text{wp } y \; Q \; s) + \)
\((P \; s \ast b \ast \text{wp } x \; R \; s + (1 - P) \ast b \ast \text{wp } y \; R \; s) \odot c\)
by\((\text{simp add:field-simps})\)
also
have ... = \((P \ast a \ast wp \, x \, Q \, s + P \ast b \ast wp \, x \, R \, s) + \)
\((1 - P \ast s) \ast a \ast wp \, y \, Q \, s + (1 - P \ast s) \ast b \ast wp \, y \, R \, s) \ominus c\)
by\((\text{simp add:ac-simps})\)
also
have ... = \(P \ast (a \ast wp \, x \, Q \, s + b \ast wp \, x \, R \, s) + \)
\((1 - P \ast s) \ast (a \ast wp \, y \, Q \, s + b \ast wp \, y \, R \, s) \ominus \)
\((P \ast s \ast c + (1 - P \ast s) \ast c)\)
by\((\text{simp add:field-simps})\)
also
have ... \leq \((P \ast (a \ast wp \, x \, Q \, s + b \ast wp \, x \, R \, s) \ominus P \ast s \ast c) + \)
\((1 - P \ast s) \ast (a \ast wp \, y \, Q \, s + b \ast wp \, y \, R \, s) \ominus (1 - P \ast s) \ast c)\)
by\((\text{rule tminus-add-mono})\)
also \{ 
from \(uP\) have \(0 \leq P \ast s\) and \(0 \leq 1 - P \ast s\)
by\((\text{auto simp:sign-simps})\)
hence \((P \ast (a \ast wp \, x \, Q \, s + b \ast wp \, x \, R \, s) \ominus P \ast s \ast c) + \)
\(((1 - P \ast s) \ast (a \ast wp \, y \, Q \, s + b \ast wp \, y \, R \, s) \ominus (1 - P \ast s) \ast c) = \)
\(P \ast s \ast (a \ast wp \, x \, Q \, s + b \ast wp \, x \, R \, s \ominus c) + \)
\((1 - P \ast s) \ast (a \ast wp \, y \, Q \, s + b \ast wp \, y \, R \, s \ominus c)\)
by\((\text{simp add:tminus-left-distrib})\)
\}
also \{ 
from \(sQ\ \, sR\ \, nna\ \, nmb\ \, nnc\ \, slx\)
have \(a \ast wp \, x \, Q \, s + b \ast wp \, x \, R \, s \ominus c \leq \)
\(wp \, x \, (\lambda s. \, a \ast Q \, s + b \ast R \, s \ominus c) \, s\)
by\((\text{blast})\)
moreover 
from \(sQ\ \, sR\ \, nna\ \, nmb\ \, nnc\ \, sly\)
have \(a \ast wp \, y \, Q \, s + b \ast wp \, y \, R \, s \ominus c \leq \)
\(wp \, y \, (\lambda s. \, a \ast Q \, s + b \ast R \, s \ominus c) \, s\)
by\((\text{blast})\)
moreover 
from \(uP\) have \(0 \leq P \ast s\) and \(0 \leq 1 - P \ast s\)
by\((\text{auto simp:sign-simps})\)
ultimately
have \(P \ast (a \ast wp \, x \, Q \, s + b \ast wp \, x \, R \, s \ominus c) + \)
\(((1 - P \ast s) \ast (a \ast wp \, y \, Q \, s + b \ast wp \, y \, R \, s) \ominus c) \leq \)
\(P \ast \, wp \, x \, (\lambda s. \, a \ast Q \, s + b \ast R \, s \ominus c) \, s + \)
\(((1 - P \ast s) \ast wp \, y \, (\lambda s. \, a \ast Q \, s + b \ast R \, s \ominus c) \, s)\)
by\((\text{blast intro:add-mono mult-left-mono})\)
\}
finally
show \(a \ast (P \ast s \ast wp \, x \, Q \, s + (1 - P \ast s) \ast wp \, y \, Q \, s) + \)
\(b \ast (P \ast s \ast wp \, x \, R \, s + (1 - P \ast s) \ast wp \, y \, R \, s) \ominus c \leq \)
\(P \ast \, wp \, x \, (\lambda s. \, a \ast Q \, s + b \ast R \, s \ominus c) \, s + \)
\(((1 - P \ast s) \ast wp \, y \, (\lambda s. \, a \ast Q \, s + b \ast R \, s \ominus c) \, s).\)
qed
4.5. SUBLINERITY

lemma sublinear-wp-DC:
  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
  shows sublinear (wp (x d y))
proof (rule sublinearI, simp only: wp-eval)
  fix R::'s ⇒ real and Q::'s ⇒ real and s::'s
  shows sublinear (wp (x d y))
proof (rule sublinearI, simp only: wp-eval)
  from nna nnb
  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
  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 (simp add: min-distrib)
  also
  from slx sQ sR nna nnb nnc
  have a * wp x Q s + b * wp x R s ⊖ c ≤
    wp x (λs. a * Q s + b * R s ⊖ c) s
      by (blast)
  moreover
  from sly sQ sR nna nnb nnc
  have a * wp y Q s + b * wp y R s ⊖ c ≤
    wp y (λs. a * Q s + b * R s ⊖ c) s
      by (blast)
  ultimately
  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 (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)
      by (auto)
  qed

As for continuity, we insist on a finite support.
lemma sublinear-wp-SetPC:
 fixes p::'a ⇒ 's prog
 assumes slp: \( \forall s\ a.\ a \in\ supp\ (P\ s) \implies\ sublinear\ (wp\ (p\ a)) \)
 and sum: \( \forall s.\ (\sum a\in supp\ (P\ s).\ P\ s\ a) \leq 1 \)
 and nnP: \( \forall s.\ 0 \leq P\ s\ a \)
 and fin: \( \forall s.\ finite\ (supp\ (P\ s)) \)
 shows sublinear\ (wp\ (SetPC\ p\ P))

proof\ (rule\ sublinearI,\ simp\ add:wp-eval)
 fix R::'s ⇒ real\ and\ Q::'s ⇒ real\ and\ s::'s
 and a::real\ and\ b::real\ and\ c::real
 assume slR: sound\ R\ and\ sQ: sound\ Q
 and nna: 0 ≤ a and nmb: 0 ≤ b and nnc: 0 ≤ c
 have a * (\( \sum a\in supp\ (P\ s).\ P\ s\ a'\ wp\ (p\ a')\ Q\ s \)) +
 b * (\( \sum a\in supp\ (P\ s).\ P\ s\ a'\ wp\ (p\ a')\ R\ s \)) ∪ c =
 (\( \sum a\in supp\ (P\ s).\ P\ s\ a'\ wp\ (p\ a')\ Q\ s + b * wp\ (p\ a')\ R\ s \)) ∪ c
 by(simp add:field-simps sum-distrib-left sum_distrib)

also have ... ≤
 (\( \sum a\in supp\ (P\ s).\ P\ s\ a'\ wp\ (p\ a')\ Q\ s + b * wp\ (p\ a')\ R\ s \)) ∪ c
 by(rule tminus-right-antimono)

also from sum and nnc have ... ≤ 1 * c
 by(rule mult-right-mono)

finally show (\( \sum a\in supp\ (P\ s).\ P\ s\ a'\ wp\ (p\ a')\ Q\ s + b * wp\ (p\ a')\ R\ s \)) ∪ c ≤ c
 by(simp)

qed

also from fin
 have ... ≤ (\( \sum a\in supp\ (P\ s).\ P\ s\ a'\ wp\ (p\ a')\ Q\ s + b * wp\ (p\ a')\ R\ s \)) ∪ c
 by(blast intro:tminus-sum-mono)

also have ... = (\( \sum a\in supp\ (P\ s).\ P\ s\ a'\ wp\ (p\ a')\ Q\ s + b * wp\ (p\ a')\ R\ s \)) \cup c

by(simp add:nnP tminus-left-distrib)
also {
 from slp sQ sR nna nmb nnc
 have \( \forall a'.\ a'\in supp\ (P\ s) \implies a * wp\ (p\ a')\ Q\ s + b * wp\ (p\ a')\ R\ s \cup c \leq wp\ (p\ a')\ (\lambda s.\ a * Q\ s + b * R\ s \cup c)\ s \)
 by(blast)

with nnP
 have (\( \sum a'\in supp\ (P\ s).\ P\ s\ a'\ wp\ (p\ a')\ Q\ s + b * wp\ (p\ a')\ R\ s \cup c \)) ≤
 (\( \sum a'\in supp\ (P\ s).\ P\ s\ a'\ wp\ (p\ a')\ (\lambda s.\ a * Q\ s + b * R\ s \cup c)\ s \))
 by(blast intro:sum-mono mult-left-mono)
}

finally
 show a * (\( \sum a'\in supp\ (P\ s).\ P\ s\ a'\ wp\ (p\ a')\ Q\ s \)) +
 b * (\( \sum a'\in supp\ (P\ s).\ P\ s\ a'\ wp\ (p\ a')\ R\ s \)) ∪ c =
 (\( \sum a'\in supp\ (P\ s).\ P\ s\ a'\ wp\ (p\ a')\ (\lambda s.\ a * Q\ s + b * R\ s \cup c)\ s \))

qed
4.5. **SUBLINEARITY**

**Lemma** sublinear-wp-SetDC:

```plaintext
fixes p::a ⇒ 's prog
assumes slp: ∃s. a ∈ S s ⇒ sublinear (wp (p a))
and hp: ∃s. a ∈ S s ⇒ healthy (wp (p a))
and nc: ∃s. S s ≠ {}
shows sublinear (wp (SetDC p S))
```

**Proof** (rule sublinearI, simp add:wp-eval, rule cInf-greatest)

```plaintext
fix P::'s ⇒ real and Q::'s ⇒ real and s::'s and x y
and a::real and b::real and c::real
assume sP: sound P and sQ: sound Q
and nna: 0 ≤ a and nnb: 0 ≤ b and nnc: 0 ≤ c
from ne show (λpr. wp (p pr) (λs. a * P s + b * Q s ⊕ c) s) · S s ≠ {} by(auto)
assume yin: y ∈ (λpr. wp (p pr) (λs. a * P s + b * Q s ⊕ c) s) · S s
then obtain x where xin: x ∈ S s and rwy: y = wp (p x) (λs. a * P s + b * Q s ⊕ c) s
by(auto)
```

```plaintext
from xin hp sP nna
have a * Inf ((λa. wp (p a) P s) · S s) ≤ a * wp (p x) P s
  by(intro mult-left-mono[OF cInf-lower] bdd-belowI[where m=0], blast+)
moreover from xin hp sQ nnb
have b * Inf ((λa. wp (p a) Q s) · S s) ≤ b * wp (p x) Q s
  by(intro mult-left-mono[OF cInf-lower] bdd-belowI[where m=0], blast+)
ultimately
have a * Inf ((λa. wp (p a) P s) · S s) +
  b * Inf ((λa. wp (p a) Q s) · S s) ⊕ c ≤
  a * wp (p x) P s + b * wp (p x) Q s ⊕ c
  by(blast intro:tmminus-left-mono add-mono)
also from xin slp sP nna nnb nnc
have ... ≤ wp (p x) (λs. a * P s + b * Q s ⊕ c) s
  by(blast)
finally show a * Inf ((λa. wp (p a) P s) · S s) +
  b * Inf ((λa. wp (p a) Q s) · S s) ⊕ c ≤ y
  by(simp add:rwy)
qed
```

**Lemma** sublinear-wp-Embed:

```
sublinear t ⇒⇒ sublinear (wp (Embed t))
by(simp add:wp-eval)
```

**Lemma** sublinear-wp-repeat:

```
[sublinear (wp p); healthy (wp p)] ⇒⇒ sublinear (wp (repeat n p))
by(induct n, simp-all add:sublinear-wp-Seq sublinear-wp-Skip healthy-wp-repeat)
```
lemma sublinear-wp-Bind:
\[
\lambda s. \text{sublinear} \ (\text{wp} \ (a \ (f \ s))) \ \Longrightarrow \ \text{sublinear} \ (\text{wp} \ (\text{Bind} \ f \ a))
\]
\text{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:
\text{fixes body::'}s prog
\text{assumes sdb: sub-distrib (wp body)}
\quad \text{and hb: healthy (wp body)}
\quad \text{and nhb: nearly-healthy (wp body)}
\text{shows sub-distrib (wp (do \ G \ \longrightarrow \ body \ od))}

\text{proof --}
\text{have } \forall \ P \ s. \ \text{sound} \ P \ \longrightarrow \ \text{wp} \ (\text{do} \ G \ \longrightarrow \ \text{body} \ \text{od}) \ P \ s \ \odot \ 1 \ \leq \ \text{wp} \ (\text{do} \ G \ \longrightarrow \ \text{body} \ \text{od}) \ (\lambda s. \ P \ s \ \odot \ 1) \ s
\text{proof (rule loop-induct[(OF \ hb \ nhb), safe])}
\text{fix } S::(s \ \text{trans} \times \ s \ \text{trans}) \ \text{set and } P::s \ \text{expect and s::'}s
\text{assume saS: } \forall x \in S. \ \forall P \ s. \ \text{sound} \ P \ \longrightarrow \ \text{fst} \ x \ P \ s \ \odot \ 1 \ \leq \ \text{fst} \ x \ (\lambda s. \ P \ s \ \odot \ 1) \ s
\quad \text{and sP: sound } P
\quad \text{and } s\beta: \forall x \in S. \ \text{feasible} \ (\text{fst} \ x)

\text{from sP have } sPm: \ \text{sound} \ (\lambda s. \ P \ s \ \odot \ 1) \ \text{by} (auto \ intro:tminus-sound)
\text{have nnSup: } \lambda s. \ 0 \ \leq \ \text{Sup-trans} \ (\text{fst} \cdot \ S) \ (\lambda s. \ P \ s \ \odot \ 1) \ s
\text{proof (cases } S::\{\}, \ \text{simp add: Sup-trans-def Sup-exp-def)}
\text{fix } s
\text{assume } S \neq \{\}
\text{then obtain } x \ \text{where } xin: x \in S \ \text{by} (auto)
\text{with } s\beta \ sPm \ \text{have } 0 \ \leq \ \text{fst} \ x \ (\lambda s. \ P \ s \ \odot \ 1) \ s \ \text{by} (auto)
\text{also from } xin \ s\beta \ sPm \ \text{have } \ldots \ \leq \ \text{Sup-trans} \ (\text{fst} \cdot \ S) \ (\lambda s. \ P \ s \ \odot \ 1) \ s
\quad \text{by} (auto \ intro: le-funD[OF Sup-trans-upper2])
\text{finally show } ?\text{thesis } s .
\text{qed}

\text{have } \lambda x s. \ \text{fst} \ x \ P \ s \ \leq \ (\text{fst} \ x \ P \ s \ \odot \ 1) + 1 \ \text{by} (simp add: tminus-def)
\text{also from saS sP}
\text{have } \lambda x s. \ x \in S \ \Longrightarrow \ (\text{fst} \ x \ P \ s \ \odot \ 1) + 1 \ \leq \ \text{fst} \ x \ (\lambda s. \ P \ s \ \odot \ 1) \ s + 1
\quad \text{by} (auto \ intro:add-right-mono)
\text{also } \{\}
\text{from sP have } \text{sound} \ (\lambda s. \ P \ s \ \odot \ 1) \ \text{by} (auto \ intro:tminus-sound)
\text{with } s\beta \ \text{have } \lambda x s. \ x \in S \ \Longrightarrow \ \text{fst} \ x \ (\lambda s. \ P \ s \ \odot \ 1) \ s + 1 \ \leq
\quad \text{Sup-trans} \ (\text{fst} \cdot \ S) \ (\lambda s. \ P \ s \ \odot \ 1) \ s + 1
\quad \text{by} (blast \ intro: add-right-mono le-funD[OF Sup-trans-upper2])
\}\n\text{finally have } le: \lambda s. \ \forall x \in S. \ \text{fst} \ x \ P \ s \ \leq \ \text{Sup-trans} \ (\text{fst} \cdot \ S) \ (\lambda s. \ P \ s \ \odot \ 1) \ s + 1
\quad \text{by} (auto)
\text{moreover from nnSup have } nn: \lambda s. \ 0 \ \leq \ \text{Sup-trans} \ (\text{fst} \cdot \ S) \ (\lambda s. \ P \ s \ \odot \ 1) \ s
4.5. **SUBLINEARITY**

+ 1

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

ultimately

**have leSup:** Sup-trans \((\text{fst } S) P s \leq \text{Sup-trans } (\text{fst } S) (\lambda s. P s \ominus 1) s + 1\)

unfolding Sup-trans-def

\[ \text{by(intro le-funD[OF Sup-exp-least], auto)} \]

**show Sup-trans \((\text{fst } S) P s \ominus 1 \leq \text{Sup-trans } (\text{fst } S) (\lambda s. P s \ominus 1) s\)**

**proof** cases Sup-trans \((\text{fst } S) P s \leq 1, \text{simp-all add:nnSup})

from leSup **have** Sup-trans \((\text{fst } S) P s - 1 \leq\)

\[ \text{Sup-trans } (\text{fst } S) (\lambda s. P s \ominus 1) s + 1 - 1 \]

\[ \text{by(auto)} \]

thus Sup-trans \((\text{fst } S) P s - 1 \leq \text{Sup-trans } (\text{fst } S) (\lambda s. P s \ominus 1) s\)

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

qed

next

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

assume IH: \(\forall P s. \text{sound } P \rightarrow t P s \ominus 1 \leq t (\lambda a. P a \ominus 1) s\)

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

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

from \(sP\) **have** sound \((\lambda s. P s \ominus 1)\) by(auto intro:tminus-sound)

with \(ft\) **have** \(s2::\text{sound } (t (\lambda s. P s \ominus 1))\) by(auto)

from \(sP ft\) **have** sound \((t P)\) by(auto)

**hence** \(s3::\text{sound } (\lambda s. t P s \ominus 1)\) by(auto intro!:tminus-sound)

**show** wp \((\text{body } ; \text{Embed } t \leftarrow G \oplus \text{Skip}) P s \ominus 1 \leq\)

\[ \text{wp } (\text{body } ; \text{Embed } t \leftarrow G \oplus \text{Skip}) (\lambda a. P a \ominus 1) s\]

**proof** simp add:wp-eval

have \("G\) s * wp body \((t P) s + (1 - "G\) s) * P s \ominus 1 = \)

\[ "G\) s * wp body \((t P) s + (1 - "G\) s) * P s \ominus ("G\) s + (1 - "G\) s)\]

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

also have .. \(\leq\) \("G\) s * wp body \((t P) s \ominus "G\) s + \)

\[ (1 - "G\) s) * P s \ominus (1 - "G\) s)\]

\[ \text{by(rule tminus-add-mono)} \]

also have .. = \("G\) s * (wp body \((t P) s \ominus 1) + (1 - "G\) s) \text{*(P s } \ominus 1)\)

\[ \text{by(simp add:tminus-left-distrib)} \]

also \{ from ft \(sP\) **have** \(wp body \((t P) s \ominus 1 \leq wp body \((\lambda s. t P s \ominus 1) s\)

\[ \text{by(auto intro:sub-distribD[OF sdb])} \]

also \{ from IH \(sP\) **have** \(\lambda s. t P s \ominus 1 \vdash t (\lambda s. P s \ominus 1)\) by(auto)

with \(sP ft s2 s3\) **have** \(wp body \((\lambda s. t P s \ominus 1) s \leq wp body \((t (\lambda s. P s \ominus 1) s\)\)

\[ \text{by(blast intro:le-funD[OF mono-transD, OF healthy-monoD, OF hh])} \]

\} finally have \("G\) s * (wp body \((t P) s \ominus 1) + (1 - "G\) s) \text{*(P s } \ominus 1) \leq\)

\[ "G\) s * wp body \((t (\lambda s. P s \ominus 1)) s + (1 - "G\) s) \text{*(P s } \ominus 1)\]

\[ \text{by(auto intro:add-right-mono mult-left-mono)} \]
finally show $\langle G\rangle \ s * \wp \\text{body} \ (t \ P) \ s + (1 - \langle G\rangle \ s) * \ P \ s \ominus 1 \leq \\
\langle G\rangle \ s * \wp \\text{body} \ (t (\lambda s. \ P \ s \ominus 1)) \ s + (1 - \langle G\rangle \ s) * (P \ s \ominus 1)$. 

qed

next

fix $t \ t'::'s \ trans$ and $P::'s \ expect$ and $s::'s$

assume $IH: \forall \ P \ s. \ sound \ P \rightarrow t \ P \ s \ominus 1 \leq t (\lambda a. \ P \ a \ominus 1) \ s$

and $eq: \equiv\text{-trans} \ t \ t'$ and $sP: \ sound \ P$

from $sP$ have $t' \ P \ s \ominus 1 = t \ P \ s \ominus 1$ by (simp add: equiv-transD[OF $eq$])

also from $sP$ $IH$ have \ldots $\leq t (\lambda s. \ P \ s \ominus 1) \ s$ by (auto)

also \{ 
from $sP$ have $sound (\lambda s. \ P \ s \ominus 1)$ by (simp add: tminus-sound)

hence $t (\lambda s. \ P \ s \ominus 1) \ s = t' (\lambda s. \ P \ s \ominus 1) \ s$ by (simp add: equiv-transD[OF $eq$])
\}

finally show $t' \ P \ s \ominus 1 \leq t' (\lambda s. \ P \ s \ominus 1) \ s$.

qed

thus $?\ thesis$ by (auto intro!: sub-distribI)$

qed

For sub-additivity, we again use the limit-of-iterates characterisation. Firstly, all iterates are sublinear:

lemma sublinear-iterates:

assumes $hb: \ healthy (\wp \\text{body})$

and $sb: \ sublinear (\wp \\text{body})$

shows $sublinear (\text{iterates body G i})$

by (induct $i$, auto intro!: sublinear-wp-PC sublinear-wp-Seq sublinear-wp-Skip sublinear-wp-Embed assms healthy-intros iterates-healthy)

From this, sub-additivity follows for the limit (i.e. the loop), by appealing to the property at all steps.

lemma sub-add-wp-loop:

fixes $body::'s \ prog$

assumes $sb: \ sublinear (\wp \\text{body})$

and $cb: \ bd-cts (\wp \\text{body})$

and $hwp: \ healthy (\wp \\text{body})$

shows $sub-add (wp (do G \rightarrow \text{body od}))$

proof

fix $P \ Q::'s \ expect$ and $s::'s$

assume $sP: \ sound \ P$ and $sQ: \ sound \ Q$

from $hwp \ cb \ sP$ have $(\lambda i. \ \text{iterates body G i P s}) \longrightarrow wp \ do \ G \rightarrow \text{body od P}$

s

by (rule loop-iterates)

moreover

from $hwp \ cb \ sQ$ have $(\lambda i. \ \text{iterates body G i Q s}) \longrightarrow wp \ do \ G \rightarrow \text{body od Q}$

s

by (rule loop-iterates)
ultimately have \((\lambda \ i \ . \ it\text{erates body } G \ i \ P \ s + it\text{erates body } G \ i \ Q \ s) \longrightarrow wp \ do \ G \longrightarrow body \ od \ P \ s + wp \ do \ G \longrightarrow body \ od \ Q \ s\)
by\(\text{rule tendsto-add}\)

moreover \{
from \text{sublinear-subadd}\[\text{OF sublinear-iterates, OF hwlp sb, OF healthy-feasibleD[OF iterates-healthy, OF hwlp]}\] \(sP \ sQ\)
have \(\bigwedge \ i . \ it\text{erates body } G \ i \ P \ s + it\text{erates body } G \ i \ Q \ s \leq it\text{erates body } G \ i \ (\lambda s. P \ s + Q \ s)\) \(s\)
by\(\text{rule sub-addD}\)
\}

moreover \{
from \(sP \ sQ\) have sound \((\lambda s. P \ s + Q \ s)\) by\(\text{blast intro:sound-intros}\)
with hwlp cb have \((\lambda \ i . it\text{erates body } G \ i \ (\lambda s. P \ s + Q \ s) \ s) \longrightarrow wp \ do \ G \longrightarrow body \ od \ (\lambda s. P \ s + Q \ s)\) \(s\)
by\(\text{blast intro:loop-iterates}\)
\}

ultimately show wp do G \longrightarrow body \ od \ P \ s + wp \ do \ G \longrightarrow body \ od \ Q \ s \leq wp \ do \ G \longrightarrow body \ od \ (\lambda s. P \ s + Q \ s)\) \(s\)
by\(\text{blast intro:LIMSEQ-le}\)

qed

\textbf{lemma} sublinear-wp-loop:
\textbf{fixes} body::’s prog
\textbf{assumes} hb: healthy (wp body)
\textbf{and} sbh: nearly-healthy (wp body)
\textbf{and} sb: sublinear (wp body)
\textbf{and} cb: bd-cts (wp body)
\textbf{shows} sublinear (wp (do G \longrightarrow body od))
\textbf{using} sublinear-sub-distrib\[\text{OF sb}\] sublinear-subadd\[\text{OF sb}\]
hb healthy-feasibleD[\text{OF hb}]
by\(\text{iprover intro:sd-sa-sublinear[OF - - healthy-wp-loop[OF hb]] sub-distrib-wp-loop sub-add-wp-loop assms}\)

\textbf{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
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))
4.6. DETERMINISM

by (rule additiveI, simp add: additiveD field-simps wp-eval)

DC is not additive.

lemma additive-wp-SetPC:
\[ \forall x. \exists s. x \in \text{supp} (p s) \implies \text{additive} (\text{wp} (a x)) \]
by (rule additiveI, simp add: wp-eval additiveD distrib-left sum_distrib)

lemma additive-wp-Bind:
\[ \forall x. \text{additive} (\text{wp} (a (f x))) \implies \text{additive} (\text{wp} (\text{Bind} f a)) \]
by (simp add: wp-eval additive-def)

lemma additive-wp-Embed:
\[ \text{additive} t \implies \text{additive} (\text{wp} (\text{Embed} t)) \]
by (simp add: wp-eval)

lemma additive-wp-repeat:
\[ \text{additive} (\text{wp} a) \implies \text{well-def} a \implies \text{additive} (\text{wp} (\text{repeat} n a)) \]
by (induct n, auto simp: additive-wp-Skip intro: additive-wp-Seq wd-intros)

lemmas fa-intros =
additive-wp-Abort additive-wp-Skip
additive-wp-Apply additive-wp-Seq
additive-wp-PC additive-wp-SetPC
additive-wp-Bind additive-wp-Embed
additive-wp-repeat

4.6.2 Maximality

lemma max-wp-Skip:
\[ \text{maximal} (\text{wp} \text{Skip}) \]
by (simp add: maximal-def wp-eval)

lemma max-wp-Apply:
\[ \text{maximal} (\text{wp} (\text{Apply} f)) \]
by (auto simp: wp-eval o-def)

lemma max-wp-Seq:
\[ \text{maximal} (\text{wp} a); \text{maximal} (\text{wp} b) \implies \text{maximal} (\text{wp} (a ; b)) \]
by (simp add: wp-eval maximal-def)

lemma max-wp-PC:
\[ \text{maximal} (\text{wp} a); \text{maximal} (\text{wp} b) \implies \text{maximal} (\text{wp} (a \oplus b)) \]
by (rule maximalI, simp add: maximalD field-simps wp-eval)

lemma max-wp-DC:
\[ \text{maximal} (\text{wp} a); \text{maximal} (\text{wp} b) \implies \text{maximal} (\text{wp} (a \sqcup b)) \]
by (rule maximalI, simp add: wp-eval maximalD)
lemma max-wp-SetPC:
  \[
  \bigwedge s. a \in \text{supp} (P s) = \Rightarrow \text{maximal (wp (p a))}; \bigwedge s. (\sum_{a \in \text{supp} (P s)} P s a) = 1 \Rightarrow \text{maximal (wp (SetPC p P))}
  \]
  by (auto simp:maximalD wp-def SetPC-def sum-distrib-right[symmetric])

lemma max-wp-SetDC:
  fixes p ::'a \Rightarrow 's prog
  assumes mp: \bigwedge s. a \in S s \Rightarrow \text{maximal (wp (p a))}
  \text{and ne:} \bigwedge s. S s \neq \{\}
  shows \text{maximal (wp (SetDC p S))}
proof (rule maximalI, rule ext, unfold wp-eval)
  fix c::real and s::'s
  assume 0 \leq c
  hence Inf ((\lambda a. wp (p a) (\lambda -. c) s) \cdot S s) = Inf ((\lambda -. c) \cdot S s)
  using mp by (simp add:maximalD cong:image-cong)
  also {
    from ne obtain a where a \in S s by blast
    hence Inf ((\lambda -. c) \cdot S s) = c
    by (auto simp add: image-constant-conv cong del: INF-cong-simp)
  } finally show Inf ((\lambda a. wp (p a) (\lambda -. c) s) \cdot S s) = c.
qed

lemma max-wp-Embed:
  maximal t \Rightarrow \text{maximal (wp (Embed t))}
by (simp add:wp-eval)

lemma max-wp-repeat:
  maximal (wp a) \Rightarrow \text{maximal (wp (repeat n a))}
by (induct n, simp-all add:max-wp-Skip max-wp-Seq)

lemma max-wp-Bind:
  assumes ma: \bigwedge s. maximal (wp (a (f s)))
  shows maximal (wp (Bind f a))
proof (rule maximalI, rule ext, simp add:wp-eval)
  fix c::real and s
  assume 0 \leq c
  with ma have wp (a (f s)) (\lambda -. c) = (\lambda -. c) \text{ by (blast)}
  thus wp (a (f s)) (\lambda -. c) s = c \text{ by (auto)}
qed

lemmas max-intros =
  max-wp-Skip max-wp-Apply
  max-wp-Seq max-wp-PC
  max-wp-DC max-wp-SetPC
  max-wp-SetDC max-wp-Embed
  max-wp-Bind max-wp-repeat
A healthy transformer that terminates is maximal.

**lemma** healthy-term-max:

- **assumes** ht: healthy t
  - and trm: λs. 1 ⊢ t (λs. 1)
- **shows** maximal t

**proof**(intro maximalI ext)

- **fix** c::real and s
  - **assume** nnc: 0 ≤ c

  - **have** t (λs. c) s = t (λs. t * c) s **by**(simp)
  - **also from** nnc healthy-scalingD[OF ht]
  - **have** ... = c * t (λs. 1) s **by**(simp add:scalingD)
  - **also** {
    - from ht **have** t (λs. 1) ⊢ λs. 1 **by**(auto)
    - with trm **have** t (λs. 1) = (λs. 1) **by**(auto)
    - hence c * t (λs. 1) s = c **by**(simp)
  }
  - **finally show** t (λs. c) s = c

**qed**

### 4.6.3 Determinism

**lemma** det-wp-Skip:

- **determ** (wp Skip)
  - **using** max-intros fa-intros **by**(blast)

**lemma** det-wp-Apply:

- **determ** (wp (Apply f))
  - **by**(intro determI fa-intros max-intros)

**lemma** det-wp-Seq:

- **determ** (wp a) ⇒ **determ** (wp b) ⇒ well-def b ⇒ **determ** (wp (a ;; b))
  - **by**(intro determI fa-intros max-intros, auto)

**lemma** det-wp-PC:

- **determ** (wp a) ⇒ **determ** (wp b) ⇒ **determ** (wp (a ⊕ b))
  - **by**(intro determI fa-intros max-intros, auto)

**lemma** det-wp-SetPC:

- (∀x s. x ∈ supp (p s) ⇒ determ (wp (a x))) ⇒
  - (∀s. finite (supp (p s))) ⇒
  - (λs. sum (p s) (supp (p s)) = 1) ⇒
  - determ (wp (SetPC a p))
  - **by**(intro determI fa-intros max-intros, auto)

**lemma** det-wp-Bind:

- (∀x. determ (wp (a (f x)))) ⇒ determ (wp (Bind f a))
  - **by**(intro determI fa-intros max-intros, auto)
\begin{verbatim}
lemma det-wp-Embed:
  determin t \rightarrow determ (wp (Embed t))
  by(simp add:wp-eval)

lemma det-wp-repeat:
  determ (wp a) \rightarrow well-def a \rightarrow determ (wp (repeat n a))
  by(intro determI fa-intros max-intros, auto)

lemmas determ-intros =
  det-wp-Skip det-wp-Apply
  det-wp-Seq det-wp-PC
  det-wp-SetPC det-wp-Bind
  det-wp-Embed det-wp-repeat
end

4.7 Well-Defined Programs.

theory WellDefined imports
  Healthiness
  Sublinearity
  LoopInduction
begin

The definition of a well-defined program collects the various notions of
healthiness and well-behavedness that we have so far established: health-
iness of the strict and liberal transformers, continuity and sublinearity of
the strict transformers, and two new properties. These are that the strict
transformer always lies below the liberal one (i.e. that it is at least as \textit{strict},
recalling the standard embedding of a predicate), and that expectation con-
junction is distributed between them in a particular manner, which will be
crucial in establishing the loop rules.

4.7.1 Strict Implies Liberal

This establishes the first connection between the strict and liberal inter-
pretations (wp and wlp).

definition
  wp-under-wlp :: 's prog \Rightarrow bool
where
  wp-under-wlp prog \equiv \forall P. unitary P \rightarrow wp prog P \vdash wlp prog P

lemma wp-under-wlpI[intro]:
  \[
  \forall P. unitary P \Rightarrow wp prog P \vdash wlp prog P \Rightarrow wp-under-wlp prog
  \]
  unfolding wp-under-wlp-def by(simp)

lemma wp-under-wlpD[dest]:
\end{verbatim}
4.7. WELL-DEFINED PROGRAMS.

\[
\begin{align*}
\text{[wp-under-wlp prog; unitary } P\text{]} & \implies wp\ prog\ P @\ wlp\ prog\ P \\
\text{unfolding wp-under-wlp-def by(simp)}
\end{align*}
\]

\textbf{lemma wp-under-le-trans:}
\[
\begin{align*}
\text{wp-under-wlp } a & \implies \text{le-utrans (wp } a)\ (wlp\ a) \\
\text{by(blast)}
\end{align*}
\]

\textbf{lemma wp-under-wlp-Abort:}
\[
\begin{align*}
\text{wp-under-wlp } \text{Abort} \\
\text{by(rule wp-under-wlpI, unfold wp-eval, auto)}
\end{align*}
\]

\textbf{lemma wp-under-wlp-Skip:}
\[
\begin{align*}
\text{wp-under-wlp } \text{Skip} \\
\text{by(rule wp-under-wlpI, unfold wp-eval, blast)}
\end{align*}
\]

\textbf{lemma wp-under-wlp-Apply:}
\[
\begin{align*}
\text{wp-under-wlp } (\text{Apply } f) \\
\text{by(auto simp:wp-eval)}
\end{align*}
\]

\textbf{lemma wp-under-wlp-Seq:}
\[
\begin{align*}
\text{assumes } h\text{-wp-a: nearly-healthy (wp } a) \\
\text{and } h\text{-wp-b: healthy (wp } b) \\
\text{and } h\text{-wp-b: nearly-healthy (wp } b) \\
\text{and } wp\text{-u-a: wp-under-wlp } a \\
\text{and } wp\text{-u-b: wp-under-wlp } b \\
\text{shows wp-under-wlp } (a ;; b)
\end{align*}
\]

\textbf{proof}(rule wp-under-wlpI, unfold wp-eval o-def)
\[
\begin{align*}
\text{fix } P::'a \Rightarrow \text{real assume } uP: \text{unitary } P \\
\text{with } h\text{-wp-b have unitary (wp } b\ P)\ \text{by(blast)} \\
\text{with } wp\text{-u-a have wp } a\ (wp\ b\ P) \vdash wp\ a\ (wp\ b\ P)\ \text{by(auto)} \\
\text{also } \{ \\
\text{from wp\text{-u-b and uP have wp } b\ P \vdash wp\ b\ P\ \text{by(blast)} } \\
\text{with h-wlp-a and h-wlp-b and h-wp-b and uP} \\
\text{have wp } a\ (wp\ b\ P) \vdash wp\ a\ (wp\ b\ P) \\
\text{by(blast intro:nearly-healthy-monoD[OF h-wlp-a])} \\
\} \\
\text{finally show wp } a\ (wp\ b\ P) \vdash wp\ a\ (wp\ b\ P). \\
\text{qed}
\]

\textbf{lemma wp-under-wlp-PC:}
\[
\begin{align*}
\text{assumes } h\text{-wp-a: healthy (wp } a) \\
\text{and } h\text{-wp-a: nearly-healthy (wp } a) \\
\text{and } h\text{-wp-b: healthy (wp } b) \\
\text{and } h\text{-wp-b: nearly-healthy (wp } b) \\
\text{and } wp\text{-u-a: wp-under-wlp } a \\
\text{and } wp\text{-u-b: wp-under-wlp } b \\
\text{and } uP: \text{unitary } P \\
\text{shows wp-under-wlp } (a \oplus b)
\end{align*}
\]

\textbf{proof}(rule wp-under-wlpI, unfold wp-eval, rule le-funI)
fix $Q::'a \Rightarrow \text{real and } s$
assume $uQ$: unitary $Q$
from $uP$ have $P \ s \leq 1$ by(blast)
hence $0 \leq 1 - P \ s$ by(simp)
moreover
from $uQ$ and $wp-u-b$ have $wp \ b \ Q \ s \leq wlp \ b \ Q \ s$ by(blast)
ultimately
have $(1 - P \ s) \ * \ wp \ b \ Q \ s \leq (1 - P \ s) \ * \ wlp \ b \ Q \ s$
  by(blast intro:mult-left-mono)
moreover {
  from $uQ$ and $wp-u-a$ have $wp \ a \ Q \ s \leq wlp \ a \ Q \ s$ by(blast)
  with $uP$ have $P \ s \ * \ wp \ a \ Q \ s \leq P \ s \ * \ wlp \ a \ Q \ s$
    by(blast intro:mult-left-mono)
} 
ultimately
show $(P \ s \ * \ wp \ a \ Q \ s + (1 - P \ s) \ * \ wp \ b \ Q \ s \leq (P \ s \ * \ wlp \ a \ Q \ s + (1 - P \ s) \ * \ wlp \ b \ Q \ s$
  by(blast intro: add-mono)
qed

lemma $wp$-$u$-$l$-$w$-$a$-$b$:
assumes $wp-u-a$: $wp$-$u$-$a$
  and $wp-u-b$: $wp$-$u$-$b$
shows $wp$-$u$-$l$-$w$($a \ \prod \ b$)
proof(rule wp-under-wlpI, unfold wp-eval, rule le-funI)
fix $Q::'a \Rightarrow \text{real and } s$
assume $uQ$: unitary $Q$
from $wp-u-a$ $uQ$ have $wp \ a \ Q \ s \leq wlp \ a \ Q \ s$ by(blast)
moreover
from $wp-u-b$ $uQ$ have $wp \ b \ Q \ s \leq wlp \ b \ Q \ s$ by(blast)
ultimately
show $\min (wp \ a \ Q \ s) (wp \ b \ Q \ s) \leq \min (wlp \ a \ Q \ s) (wlp \ b \ Q \ s)$
  by(auto)
qed

lemma $wp$-$u$-$l$-$w$-$b$-$f$-$a$-$g$:
assumes $wp-u-f$: $\forall a. \ a \in \supp (P \ s) \Rightarrow wp-under-wlp (f \ a)$
  and $nP$: $\forall a. \ a \in \supp (P \ s) \Rightarrow 0 \leq P \ s \ a$
shows $wp$-$u$-$l$-$w$-$b$-$f$-$a$-$g$($SetPC \ f \ P$)
proof(rule wp-under-wlpI, unfold wp-eval, rule le-funI)
fix $Q::'a \Rightarrow \text{real and } s$
assume $uQ$: unitary $Q$
from $wp-u-f$ $uQ$ $nP$
show $(\sum a\in \supp (P \ s). \ P \ s \ a \ * \ wp \ (f \ a) \ Q \ s) \leq (\sum a\in \supp (P \ s). \ P \ s \ a \ * \ wlp \ (f \ a) \ Q \ s)$
4.7. WELL-DEFINED PROGRAMS.

by(auto intro; sum-mono mult-left-mono)

Qed

Lemma wp-under-wlp-SetDC:

Assumes wp-u-f: \( \forall s. a \in S \implies wp-under-wlp (f a) \)

And hf: \( \forall s. a \in S \implies healthy (wp (f a)) \)

And nS: \( \forall s. S \neq \{\} \)

Shows wp-under-wlp (SetDC f S)

Proof (rule wp-under-wlpI, rule le-funI, unfold wp-eval)

Fix Q::′a ⇒ real

Assume uQ: unitary Q

Show Inf ((λa. wp (f a) Q s) ‘ S s) ≤ Inf ((λa. wlp (f a) Q s) ‘ S s)

Proof (rule cInf-mono)

From nS show (λa. wlp (f a) Q s) ‘ S s ≠ {} by (blast)

Fix x Assume xin: x ∈ (λa. wlp (f a) Q s) ‘ S s

Then Obtain a Where ain: a ∈ S s and xrw: x = wlp (f a) Q s

By (blast)

With wp-u-f uQ

Have wp (f a) Q s ≤ wlp (f a) Q s by (blast)

Moreover From ain Have wp (f a) Q s ∈ (λa. wp (f a) Q s) ‘ S s

By (blast)

Ultimately Show ∃ y ∈ (λa. wp (f a) Q s) ‘ S s. y ≤ x

By (auto simp: xrw)

Next

Fix y Assume yin: y ∈ (λa. wp (f a) Q s) ‘ S s

Then Obtain a Where ain: a ∈ S s and yrw: y = wp (f a) Q s

By (blast)

With hf uQ Have unitary (wp (f a) Q) by (auto)

With yrw Show 0 ≤ 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 (wp body)

And wp-under: wp-under-wlp body

Shows wp-under-wlp (do G → body od)

Proof (rule wp-under-wlpI)

Fix P::′s expect

Assume uP: unitary P hence sP: sound P by (auto)
let ?X Q s = «G» s * wp body Q s + «N G» s * P s
let ?Y Q s = «G» s * wlp body Q s + «N G» s * P s

show wp (do G → body od) P ⊢ wlp (do G → body od) P

proof (simp add: hwp hwlwp sP uP wp-Loop1 wlp-Loop1, rule gfp-exp-upperbound)
  thm lfp-loop-fp
  from hwp sP have lfp-exp ?X = ?X (lfp-exp ?X)
    by (rule lfp-wp-loop-unfold)
  hence lfp-exp ?X ⊢ ?X (lfp-exp ?X) by (simp)
  also {
    from hwp uP have wp body (?X = ?X (lfp-exp ?X))
      by (auto intro: wp-under-wlpD OF wp-under lfp-loop-unitary)
      by (auto intro: add-mono mult-left-mono)
  }
from hwp uP show unitary (lfp-exp ?X)
  by (auto intro: lfp-loop-unitary)
qed

lemma wp-under-wlp-repeat:
  [ healthy (wp a); nearly-healthy (wlp a); wp-under-wlp a ] =⇒
  wp-under-wlp (repeat n a)
by (induct n, auto intro: wp-under-wlp-Skip wp-under-wlp-Seq healthy-intros)

lemma wp-under-wlp-Bind:
  [ ∀ s. wp-under-wlp (a (f s)) ] =⇒ wp-under-wlp (Bind f a)
unfolding wp-under-wlp-def by (auto simp: wp-eval)

lemmas wp-under-wlp-intros =
  wp-under-wlp-Abort wp-under-wlp-Skip
  wp-under-wlp-Apply wp-under-wlp-Seq
  wp-under-wlp-PC wp-under-wlp-DC
  wp-under-wlp-SetPC wp-under-wlp-SetDC
  wp-under-wlp-Embed wp-under-wlp-loop
  wp-under-wlp-repeat wp-under-wlp-Bind

4.7.2 Sub-Distributivity of Conjunction

definition
  sub-distrib-pconj :: 's prog ⇒ bool
where
  sub-distrib-pconj prog ≡
  ∀ P Q. unitary P → unitary Q →
  wlp prog P & & wp prog Q ⊢ wp prog (P & & Q)

lemma sub-distrib-pconj[intro]:
  [∀ P Q. [ unitary P; unitary Q ] =⇒ wlp prog P & & wp prog Q ⊢ wp prog (P
4.7. WELL-DEFINED PROGRAMS.

\[ Q \] \rightarrow sub-distrib-pconj prog

unfolding sub-distrib-pconj-def by(simp)

lemma sub-distrib-pconjD[dest]:
\[ \forall P \ Q. \ [ \ sub-distrib-pconj prog; \ unitary P; \ unitary Q ] \rightarrow \ wp\ prog\ P \ \&\& \ wp\ prog\ Q \ \doteqdot \ \ wp\ prog\ (P \ \&\& \ Q) \]

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 \Rightarrow real and Q::'a \Rightarrow real
assume uP: unitary P and uQ: unitary Q

with h-wp-b and h-wlp-b
have wp a (wp b P) \&\& wp a (wp b Q) \doteqdot wp a (wp b P \&\& wp b Q)
by(blast intro!:sub-distrib-pconjD[OF sdp-a])
also {
from sdp-b and uP and uQ
have wp b P \&\& wp b Q \doteqdot 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) \doteqdot wp a (wp b (P \&\& Q))
by(blast intro!:mono-transD[OF healthy-monoD, OF h-wp-a] unitary-sound unitary-intros sound-intros)
}
finally show wp a (wp b P) \&\& wp a (wp b Q) \doteqdot wp a (wp b (P \&\& Q)) .
qed

lemma sdp-Apply:
sub-distrib-pconj (Apply f)
by(rule sub-distrib-pconjI, simp add:wp-eval)

lemma sdp-DC:
fixes a::'s prog and b
assumes $sdp-a$: $sub-distrib-pconj\ a$
and $sdp-b$: $sub-distrib-pconj\ b$
and $h-wp-a$: $healthy\ (wp\ a)$
and $h-wp-b$: $healthy\ (wp\ b)$
and $h-wlp-b$: $nearly-healthy\ (wlp\ b)$
shows $sub-distrib-pconj\ (a \bigcap\ b)$

proof
(rule $sub-distrib-pconjI$, unfold $wp$-eval, rule $le-funI$)

fix $P::'s \Rightarrow real$ and $Q::'s \Rightarrow real$ and $s::'s$
assume $uP$: unitary $P$ and $uQ$: unitary $Q$

have $((\lambda s.\ min\ (wlp\ a\ P\ s)\ (wlp\ b\ P\ s))\ \&\&$
$(\lambda s.\ min\ (wp\ a\ Q\ s)\ (wp\ b\ Q\ s)))\ s \leq$
$min\ (wlp\ a\ P\ s\ \&\ wp\ a\ Q\ s)\ (wlp\ b\ P\ s\ \&\ wp\ b\ Q\ s)$

unfolding $exp-conj-def$ by (rule $min-\text{conj}$)

also {
have $(\lambda s.\ wlp\ b\ P\ s\ \&\ wp\ a\ Q\ s) = wlp\ b\ P\ \&\ wp\ a\ Q$
by (simp add: $exp-conj-def$)
also from $sdp-a\ uP\ uQ$ have $\vdash wp\ a\ (P\ \&\&\ Q)$
by (blast dest: $sub-distrib-pconjD$)
finally have $wlp\ b\ P\ s\ \&\ wp\ a\ Q\ s \leq wp\ a\ (P\ \&\&\ Q)$
by (rule $le-funD$)
moreover {
have $(\lambda s.\ wlp\ b\ P\ s\ \&\ wp\ b\ Q\ s) = wlp\ b\ P\ \&\ wp\ b\ Q$
by (simp add: $exp-conj-def$)
also from $sdp-b\ uP\ uQ$ have $\vdash wp\ b\ (P\ \&\&\ Q)$
by (blast)
finally have $wlp\ b\ P\ s\ \&\ wp\ b\ Q\ s \leq wp\ b\ (P\ \&\&\ Q)$
by (rule $le-funD$)
}
ultimately
have $min\ (wlp\ a\ P\ s\ \&\ wp\ a\ Q\ s)\ (wlp\ b\ P\ s\ \&\ wp\ b\ Q\ s) \leq$
$min\ (wp\ a\ (P\ \&\&\ Q)\ s)\ (wp\ b\ (P\ \&\&\ Q)\ s)$ by (auto)
}
finally
show $((\lambda s.\ min\ (wlp\ a\ P\ s)\ (wlp\ b\ P\ s))\ \&\&$
$(\lambda s.\ min\ (wp\ a\ Q\ s)\ (wp\ b\ Q\ s)))\ s \leq$
$min\ (wp\ a\ (P\ \&\&\ Q)\ s)\ (wp\ b\ (P\ \&\&\ Q)\ s)$.

defined

lemma $sdp-PC$;
fixes $a::'s\ prog$ and $b$
assumes $sdp-a$: $sub-distrib-pconj\ a$
and $sdp-b$: $sub-distrib-pconj\ b$
and $h-wp-a$: $healthy\ (wp\ a)$
and $h-wp-b$: $healthy\ (wp\ b)$
and $h-wlp-b$: $nearly-healthy\ (wlp\ b)$
and $uP$: unitary $P$
shows $sub-distrib-pconj\ (a\ \bigoplus\ b)$
proof
(rule $sub-distrib-pconjI$, unfold $wp$-eval, rule $le-funI$)
4.7. WELL-DEFINED PROGRAMS.

\[
\text{fix } Q:\: \text{real } \Rightarrow \text{unitary } Q \quad \text{and } R:\: \text{real } \Rightarrow \text{unitary } R
\]

\[
\text{assume } uQ: \text{unitary } Q \quad \text{and } uR: \text{unitary } R
\]

\[
\text{have } nnA: 0 \leq P s \quad \text{and } nnB: P s \leq 1
\]

\[
\text{using } uP \quad \text{by(auto simp:sign-simps)}
\]

\[
\text{note } nn = nnA \quad nnB
\]

\[
\begin{align*}
\text{have } & ((\lambda s. P s * \text{wp a } Q s + (1 - P s) * \text{wp b } Q s) \land \land \\
& (\lambda s. P s * \text{wp a } R s + (1 - P s) * \text{wp b } R s)) \leq \\
& ((P s * \text{wp a } Q s + (1 - P s) * \text{wp b } Q s) + \\
& (P s * \text{wp a } R s + (1 - P s) * \text{wp b } R s)) \lor 1 \\
& \text{by(simp add:conj-def pconj-def)}
\end{align*}
\]

\[
\text{also have } \quad P s * (\text{wp a } Q s + \text{wp a } R s) + \\
(1 - P s) * (\text{wp b } Q s + \text{wp b } R s) \lor 1 \\
\text{by(simp add:field-simps)}
\]

\[
\begin{align*}
\text{also have } & \quad P s \leq (P s * (\text{wp a } Q s + \text{wp a } R s) \lor P s) + \\
& ((1 - P s) * (\text{wp b } Q s + \text{wp b } R s) \lor (1 - P s)) \\
& \text{by(rule tminus-zero)}
\end{align*}
\]

\[
\begin{align*}
\text{also have } & \quad (P s * ((\text{wp a } Q s + \text{wp a } R s) \lor 1)) + \\
& ((1 - P s) * (\text{wp b } Q s + \text{wp b } R s) \lor 1)) \\
& \text{by(simp add:nn tminus-zero)}
\end{align*}
\]

\[
\begin{align*}
\text{also have } & \quad P s * ((\text{wp a } Q s \land \text{wp a } R s)) + \\
& (1 - P s) * ((\text{wp b } Q s \land \text{wp b } R s)) \\
& \text{by(simp add:exp-conj-def pconj-def)}
\end{align*}
\]

\[
\begin{align*}
\text{also } \{ & \quad \text{from } \text{sdp-a } \text{sdp-b } uQ uR \\
& \quad \text{have } P s * (\text{wp a } Q s \land \text{wp a } R s) s \leq P s * \text{wp a } (Q s \land R s) \\
& \quad \text{and } (1 - P s) * (\text{wp b } Q s \land \text{wp b } R s) s \leq (1 - P s) * \text{wp b } (Q s \land R s) \\
& \quad \text{by(simpl all add: entailsD mult-left mono nn sub-distrib-pconjD)} \}
\end{align*}
\]

\[
\begin{align*}
\text{hence } & \quad P s * (\text{wp a } Q s \land \text{wp a } R s) + \\
& (1 - P s) * ((\text{wp b } Q s \land \text{wp b } R s) s \leq \\
& P s * \text{wp a } (Q s \land R s) s + (1 - P s) * \text{wp b } (Q s \land R s) s \\
& \text{by(auto)}
\end{align*}
\]

\[
\begin{align*}
\text{finally show } & \quad ((\lambda s. P s * \text{wp a } Q s + (1 - P s) * \text{wp b } Q s) \land \land \\
& (\lambda s. P s * \text{wp a } R s + (1 - P s) * \text{wp b } R s)) s \leq \\
& P s * \text{wp a } (Q s \land R s) s + (1 - P s) * \text{wp b } (Q s \land R s) s .
\end{align*}
\]

\[
\text{qed}
\]

**Lemma** \text{sdp-Embed}:
\[
[f \text{ unitary } P: \text{unitary } Q ] \Rightarrow t P \land t Q \vdash t (P \land Q)
\]

by(auto simp:wp-eval)

**Lemma** \text{sdp-repeat}:
\textbf{CHAPTER 4. THE PGCL LANGUAGE}

fixes \(a::'s prog\)
assumes \(sdp: \text{sub-distrib-pconj } a\)
and \(hwp: \text{healthy } (wp a)\) and \(hwlp: \text{nearly-healthy } (wlp a)\)
shows \(\text{sub-distrib-pconj } (\text{repeat } n a)\) (is \(?X n\))

\textbf{proof (induct }n\text{)}
show \(?X \emptyset\) by (simp add: sdpa-Skip)
fix \(n\) assume \(IH:\ ?X n\)
show \(?X (Suc n)\)
proof (rule sub-distrib-pconjI, simp add: wp-eval)
fix \(P::'s \Rightarrow \text{real}\) and \(Q::'s \Rightarrow \text{real}\)
assume \(uP: \text{unitary } P\) and \(uQ: \text{unitary } Q\)
from assms have \(hwlp a: \text{nearly-healthy } (\text{wlp } (\text{repeat } n a))\)
and \(hwp a: \text{healthy } (wp (\text{repeat } n a))\)
by (auto intro: healthy-intros)

moreover from \(uP\) and \(hwlp a\) have unitary \((\text{wlp } (\text{repeat } n a)) P\)
by (blast)
moreover from \(uQ\) and \(hwp a\) have unitary \((wp (\text{repeat } n a)) Q\)
by (blast)
ultimately have \(wlp a (wp (\text{repeat } n a)) (P \land wp (\text{repeat } n a)) Q\)
using sdpa by (blast)
also \{
from \(hwlp a\) have \(\text{nearly-healthy } (wp (\text{repeat } n a))\)
by (rule healthy-intros)
with \(uP\) have \(\text{sound } (wp (\text{repeat } n a)) P\)
by (auto)
moreover from \(hwp a\) have \(\text{sound } (wp (\text{repeat } n a)) Q\)
by (auto intro: healthy-intros)
ultimately have \(\text{sound } ((wp (\text{repeat } n a)) P \land wp (\text{repeat } n a)) Q\)
by (rule exp-conj-sound)
moreover \{
from \(uP\) \(uQ\) have \(\text{sound } (P \land wp (\text{repeat } n a)) (P \land Q)\)
by (auto intro: healthy-intros)
\}
moreover from \(uP\) \(uQ\) \(IH\)
have \(wlp (\text{repeat } n a) P \land wp (\text{repeat } n a) Q \vdash wp (\text{repeat } n a) (P \land wp (\text{repeat } n a)) Q\)
by (blast)
ultimately have \(wp a (wp (\text{repeat } n a)) (P \land wp (\text{repeat } n a)) Q\)
\(\vdash wp a (wp (\text{repeat } n a)) (P \land wp (\text{repeat } n a)) Q\).
\}
finally show \(wp a (wp (\text{repeat } n a) P \land wp a (wp (\text{repeat } n a) Q) \vdash wp a (wp (\text{repeat } n a) (P \land wp (\text{repeat } n a))) Q\).

qed

\textbf{lemma} \(sdp-SetPC:\)
fixes \(pa::'s prog\)
assumes \(sdp: \forall s. a \in \text{supp } (P s) \Rightarrow \text{sub-distrib-pconj } (p a)\)
and \(fin: \forall s. \text{finite } (\text{supp } (P s))\)
4.7. WELL-DEFINED PROGRAMS.

\[
\begin{align*}
\text{and } & \text{nnp: } \bigwedge s a. \ 0 \leq P s a \\
\text{and } & \text{sub: } \bigwedge s. \text{ sum } (P s) (\text{ supp } (P s)) \leq 1 \\
\text{shows } & \text{ sub-distrib-pconj } (\text{ SetPC } p \ P) \\
\text{proof} \ & (\text{ rule sub-distrib-pconjI } , \text{ simp add: wp-eval } , \text{ rule le-funI }) \\
\text{fix } & Q::'s \Rightarrow \text{ real } \text{ and } R::'s \Rightarrow \text{ real } \text{ and } s::'s \\
\text{assume } & uQ:: \text{ unitary } Q \text{ and } uR:: \text{ unitary } R \\
\text{have } & ((\lambda s . \sum a \in \text{ supp } (P s). \ P s a \ast \text{ wlp } (p a) \ Q s) \ & \& (\lambda s . \sum a \in \text{ supp } (P s). \ P s a \ast \ \text{ wp } (p a) \ R s)) s = \sum a \in \text{ supp } (P s). \ P s a \ast \text{ wlp } (p a) \ Q s + (\sum a \in \text{ supp } (P s). \ P s a \ast \text{ wp } (p a) \ R s) \leftarrow 1 \\
\text{also have } & \ldots = (\sum a \in \text{ supp } (P s) \ P s a \ast (\text{ wlp } (p a) \ Q s + \text{ wp } (p a) \ R s)) \leftarrow 1 \\
\text{by (simp add: sum. distrib field-simps) } \\
\text{also from } & \text{ sub } \\
\text{have } & \ldots \leq (\sum a \in \text{ supp } (P s) \ P s a \ast (\text{ wlp } (p a) \ Q s + \text{ wp } (p a) \ R s)) \leftarrow 1 \\
\text{by (rule tminus-right-antimono) } \\
\text{also from } & \text{ fin } \\
\text{have } & \ldots \leq (\sum a \in \text{ supp } (P s) \ P s a \ast (\text{ wlp } (p a) \ Q s + \text{ wp } (p a) \ R s) \leftarrow P s a) \\
\text{by (rule tminus-sum-mono) } \\
\text{also from } & \text{ nnp } \\
\text{have } & \ldots = (\sum a \in \text{ supp } (P s) \ P s a \ast (\text{ wlp } (p a) \ Q s + \text{ wp } (p a) \ R s) \leftarrow 1)) \\
\text{by (simp add: tminus-left-distrib) } \\
\text{also have } & \ldots = (\sum a \in \text{ supp } (P s) \ P s a \ast (\text{ wlp } (p a) \ Q \ & \& \text{ wp } (p a) \ R s)) s \\
\text{by (simp add: pconj-def exp-conj-def) } \\
\text{also } & \{ \\
\text{from } & \text{ sdp uQ auR } \\
\text{have } & ((\lambda a. \ a \in \text{ supp } (P s) \Rightarrow \text{ wlp } (p a) \ Q \ & \& \ \text{ wp } (p a) \ R) \leftarrow wp (p a) (Q \ & \& R) \\
\text{by (blast intro: sub-distrib-pconjD) } \\
\text{with } & \text{ nnp } \\
\text{have } & (\sum a \in \text{ supp } (P s) \ P s a \ast (\text{ wlp } (p a) \ Q \ & \& \text{ wp } (p a) \ R) s) \leq (\sum a \in \text{ supp } (P s) \ P s a \ast (\text{ wp } (p a) (Q \ & \& R) s)) s \\
\text{by (blast intro: sum-mono mult-left-mono) } \\
\text{ } \\
\text{finally show } & ((\lambda s . \sum a \in \text{ supp } (P s) \ P s a \ast (\text{ wlp } (p a) \ Q s) \ & \& (\lambda s . \sum a \in \text{ supp } (P s) \ P s a \ast (\text{ wp } (p a) \ R s)) s \leq (\sum a \in \text{ supp } (P s) \ P s a \ast (\text{ wp } (p a) (Q \ & \& R) s)) s. \\
\text{qed}
\end{align*}
\]

**Lemma sdp-SetDC:**

**Fixes** p::'a ⇒ 's prog

**Assumes** sdp: \( \bigwedge s a. \ a \in S s \Rightarrow \text{ sub-distrib-pconj } (p a) \)

**And** hw: \( \bigwedge s a. \ a \in S s \Rightarrow \text{ healthy } (wp \ (p a)) \)

**And** hwlp: \( \bigwedge s a. \ a \in S s \Rightarrow \text{ nearly-healthy } (\text{ wlp } (p a)) \)

**And** nc: \( \bigwedge S \ s \neq \{\} \)

**Shows** sub-distrib-pconj (SetDC p S)

**Proof** (rule sub-distrib-pconjI, rule le-funI)

**Fix** P::'s ⇒ real and Q::'s ⇒ real and s::'s
assume \( uP \): unitary \( P \) and \( uQ \): unitary \( Q \)

from \( uP \) hwlp
have \( \forall x. x \in (\lambda a. \text{wlp}(p a) P) \cdot S \Rightarrow \text{unitary } x \) by(auto)
hence \( \forall y. y \in (\lambda a. \text{wlp}(p a) P) s \cdot S \Rightarrow 0 \leq y \) by(auto)
hence \( \forall a. a \in S s \Rightarrow \text{wlp}(\text{SetDC} p S) P s \leq \text{wlp}(p a) P s \)

unfolding \( \text{wp-eval} \) by(intro cInf-lower bdd-belowI, auto)
moreover {
from \( uQ \) hwp have \( \forall a. a \in S s \Rightarrow 0 \leq \text{wp}(p a) Q s \) by(blast)
hence \( \forall a. a \in S s \Rightarrow \text{wp}(\text{SetDC} p S) Q s \leq \text{wp}(p a) Q s \)

unfolding \( \text{wp-eval} \) by(intro cInf-lower bdd-belowI, auto)
}
ultimately
have \( \forall a. a \in S s \Rightarrow \text{wlp}(\text{SetDC} p S) P s \leq \text{wp}(p a) P s \)

unfolding \( \text{wp-eval} \) by(intro cInf-lower bdd-belowI, auto)
also have \( \forall a. \text{wp}(p a) P s \leq \text{wp}(p a) \) \( Q s \) by(auto)
also from \( \text{sdp} \) \( uP \) \( uQ \) have \( \forall a. a \in S s \Rightarrow ... a \leq \text{wp}(p a) \) \( P \) \&\& \( Q \) s
by(blast)
also have \( \forall a. ... a = \text{wp}(p a) \) \( \lambda s. P s + Q s \) \( \cup \) \( I \) 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 \( \text{exp-conj-def} \) \( \text{pconj-def} \) \( \text{wp-eval} \)
using \( \text{ne} \) by(blast intro!:cInf-greatest)
qed

lemma \( \text{sdp-\text{Bind}} \):
\[
[ \forall s. \text{sub-distrib-pconj} (p (f s)) ] \Rightarrow \text{sub-distrib-pconj} (\text{Bind} f p)
\]
unfolding \( \text{sub-distrib-pconj-def} \) \( \text{wp-eval} \) \( \text{exp-conj-def} \) \( \text{pconj-def} \)
by(blast)

For loops, we again appeal to our transfinite induction principle, this time taking advantage of the simultaneous treatment of both strict and liberal transformers.

lemma \( \text{sdp-loop} \):
fixes \( \text{body} \): `'s prog
assumes \( \text{sdp-body} \): \( \text{sub-distrib-pconj body} \)
and \( \text{hwlp} \): nearly-healthy \( (\text{wlp body}) \)
and \( \text{hwp} \): healthy \( (\text{wp body}) \)
shows \( \text{sub-distrib-pconj} (\text{do } G \rightarrow \text{body od}) \)

proof (rule \( \text{sub-distrib-pconjI} \), rule \( \text{loop-induct} \)(OF \( \text{hwlp} \))
fix \( P Q :: \text{expect and } S :: (\text{trans } \times \text{trans}) \) \( \text{set} \)
assume \( uP :: \text{unitary } P \) and \( uQ :: \text{unitary } Q \)
and \( \text{ffst}: \forall x \in S. \text{feasible } (\text{fst } x) \)
and $\forall x \in S. \forall Q. $ unitary $Q \rightarrow$ unitary $(\operatorname{snd} x Q)$
and $\forall x \in S. \operatorname{snd} x P \& \& \operatorname{fst} x Q \vdash \operatorname{fst} x (P \& \& Q)$

show $\operatorname{Inf-trans} (\operatorname{snd} \ ' S) P \& \& \operatorname{Sup-trans} (\operatorname{fst} \ ' S) Q \vdash$
$\operatorname{Sup-trans} (\operatorname{fst} \ ' S) (P \& \& Q)$

proof(cases)
assume $S = \{\}$
thus $?\text{thesis}$
by(simp add:$\operatorname{Inf-trans-def} \\operatorname{Sup-trans-def} \\operatorname{Inf-trans-def}$
$\operatorname{Inf-exp-def} \\operatorname{Sup-exp-def} \\operatorname{exp-conj-def}$)

next
assume $ne: S \neq \{\}$

let $?f \ s = 1 + \operatorname{Sup-trans} (\operatorname{fst} \ ' S) (P \& \& Q) s - \operatorname{Inf-trans} (\operatorname{snd} \ ' S) P s$

from $ne$ obtain $t$ where $tin: t \in \operatorname{fst} \ ' S$ by(auto)
from $ne$ obtain $u$ where $uin: u \in \operatorname{snd} \ ' S$ by(auto)

from $tin \ \operatorname{ffst} uP uQ$ have $utPQ$: unitary $(t (P \& \& Q))$
by(auto intro:$\operatorname{exp-conj-unitary}$

hence $\forall s. 0 \leq t (P \& \& Q) s$ by(auto)
also {
from $\operatorname{ffst} tin$ have $le: \operatorname{le-trans} t (\operatorname{Sup-trans} (\operatorname{fst} \ ' S))$
by(auto intro:$\operatorname{Sup-trans-upper}$

with $uP uQ$ have $\forall s. t (P \& \& Q) s \leq \operatorname{Sup-trans} (\operatorname{fst} \ ' S) (P \& \& Q) s$
by(auto intro:$\operatorname{exp-conj-unitary}$
}
finally have $nn-rhs: \forall s. 0 \leq \operatorname{Sup-trans} (\operatorname{fst} \ ' S) (P \& \& Q) s .$

have $\forall R. \operatorname{Inf-trans} (\operatorname{snd} \ ' S) P \& \& R \vdash \operatorname{Sup-trans} (\operatorname{fst} \ ' S) (P \& \& Q) \Rightarrow$
$R \leq ?f$

proof(rule contrapos-pp, assumption)
fix $R$
assume $\neg R \leq ?f$
then obtain $s$ where $\neg R s \leq ?f s$ by(auto)

hence $gt: ?f s < R s$ by(simp)

from $nn-rhs$ have $g1: 1 \leq 1 + \operatorname{Sup-trans} (\operatorname{fst} \ ' S) (P \& \& Q) s$ by(auto)

hence $\operatorname{Sup-trans} (\operatorname{fst} \ ' S) (P \& \& Q) s = \operatorname{Inf-trans} (\operatorname{snd} \ ' S) P s \& \& ?f s$
by(simp add:$\operatorname{pconj-def}$)
also from $g1$ have $\ldots = \operatorname{Inf-trans} (\operatorname{snd} \ ' S) P s + ?f s - 1$
by(simp)
also from $gt$ have $\ldots < \operatorname{Inf-trans} (\operatorname{snd} \ ' S) P s + R s - 1$
by(simp)
also {
with $g1$ have $1 \leq \operatorname{Inf-trans} (\operatorname{snd} \ ' S) P s + R s$
by(simp)

hence $\operatorname{Inf-trans} (\operatorname{snd} \ ' S) P s + R s - 1 = \operatorname{Inf-trans} (\operatorname{snd} \ ' S) P s \& \& R s$
by(simp add:$\operatorname{pconj-def}$)
170  

CHAPTER 4. THE PGCL LANGUAGE

} 

final
have \( \neg (\inf\text{-}utrans (\snd \cdot S) P \&\& R) s \leq \sup\text{-}trans (\fst \cdot S) (P \&\& Q) s \)
by(simp add:exp-conj-def)
thus \( \neg \inf\text{-}utrans (\snd \cdot S) P \&\& R \vdash \sup\text{-}trans (\fst \cdot S) (P \&\& Q) \)
by(auto)
qed

moreover have \( \forall t \in \fst \cdot S. \inf\text{-}utrans (\snd \cdot S) P \&\& t Q \vdash \sup\text{-}trans (\fst \cdot S) (P \&\& Q) \)
proof 
fix t 
assume tin: \( t \in \fst \cdot S \)
then obtain x where xin: \( x \in S \) and fx: \( t = \fst x \)
by(auto)
from xin have \( \snd x \in \snd \cdot S \)
by(auto)
with uP usnd have Inf-utrans (snd \cdot S) P \vdash snd x P
by(auto intro:le-utransD[OF Inf-utrans-lower])
hence Inf-utrans (snd \cdot S) P \&\& \fst x Q \vdash snd x P \&\& \fst x Q
by(auto intro:entails-frame)
also from xin IH have ...
\( \vdash \fst x (P \&\& Q) \)
by(auto)
also from xin ffst exp-conj-unitary[OF uP uQ]
have ...
\( \vdash \sup\text{-}trans (\fst \cdot S) (P \&\& Q) \)
by(auto intro:le-utransD[OF Sup-trans-upper])
finally show Inf-utrans (snd \cdot S) P \&\& t Q \vdash \sup\text{-}trans (\fst \cdot S) (P \&\& Q)
by(simp add:fx)
qed

ultimately have bt: \( \forall t \in \fst \cdot S. t Q \vdash \?f \)
by(blast)

have Sup-trans (fst \cdot S) Q = Sup-exp \( \{ t Q | t \in \fst \cdot S \} \)
by(simp add:Sup-trans-def)
also have ...
\( \vdash \?f \)
proof(rule Sup-exp-least)
from bt show \( \forall R \in \{ t Q | t \in \fst \cdot S \}, R \vdash \?f \)
by(blast)
from ne obtain t where tin: \( t \in \fst \cdot S \)
by(auto)
with ffst uQ have unitary (t Q) by(auto)
therefore \( \lambda s. 0 _{=\:} t Q \)
by(auto)
also from tin bt have ...
\( \vdash \?f \)
by(auto)
finally show nneg (\( \lambda s. 1 + \sup\text{-}trans (\fst \cdot S) (P \&\& Q) s \) - \inf\text{-}utrans (snd \cdot S) P s)
by(auto)
qed

finally have Inf-utrans (snd \cdot S) P \&\& Sup-trans (fst \cdot S) Q \vdash Inf-utrans (snd \cdot S) P \&\& \?f
by(auto intro:entails-frame)
also from nn-rhs have ...
\( \vdash \sup\text{-}trans (\fst \cdot S) (P \&\& Q) \)
by(simp add:exp-conj-def pconj-def)
finally show \( \?\text{thesis} \).
qed
4.7. WELL-DEFINED PROGRAMS.

next

fix \( P \) \( Q \) :: 's expect and \( t \) :: 's trans
assume \( uP \) : unitary \( P \) and \( uQ \) : unitary \( Q \)
and \( ft \) : feasible \( t \)
and \( uu \) : \( \{ \lambda Q. \ unitary \ Q \imp unitary (u \ Q) \} \)
and \( IH \) : \( uP \) \&\& \( t \) \( Q \) \( \vdash \) \( (P \) \&\& \( Q) \)

show \( wp \) \( (\text{body} ; ; \text{Embed} \ uG @ \text{Skip}) \) \( P \) \&\&

wp \( (\text{body} ; ; \text{Embed} \ t * G @ \text{Skip}) \) \( Q \) \( \vdash \)
wp \( (\text{body} ; ; \text{Embed} \ t * G @ \text{Skip}) \) \( (P \) \&\& \( Q) \)

proof (rule le-funI, simp add: wp-eval exp-conj-def pconj-def)

fix \( s \) ::

have \( \langle G \rangle \ s * wp \) \( \text{body} \ (uP) \ s + \ (1 - \langle G \rangle \ s) * \ P \) \( s + \)

\( \langle G \rangle \ s * wp \) \( \text{body} \ (tQ) \ s + \ (1 - \langle G \rangle \ s) * \ Q \) \( \sinter \ 1 \) \( = \)

\( \langle G \rangle \ s * wp \) \( \text{body} \ (uP) \ s + \langle G \rangle \ s * wp \) \( \text{body} \ (tQ) \ s + \)

\( \langle 1 - \langle G \rangle \ s \rangle * \ P \) \( s + \langle 1 - \langle G \rangle \ s \rangle * \ Q \) \( s \sinter (\langle G \rangle \ s + (1 - \langle G \rangle \ s)) \)

by (simp add: ac-simps)
also have \( \ldots \leq \)

\( \langle G \rangle \ s * (wp \) \( \text{body} \ (uP) \ s + wp \) \( \text{body} \ (tQ) \ s \sinter 1 \) \( + \)

\( 1 - \langle G \rangle \ s \rangle * (P \ s + Q \ s \sinter 1) \)

by (simp add: tminus-left-distrib distrib-left)
also \{ 

from \( uP \) \( uQ \) \( ft \) \( uu \)

have \( wp \) \( \text{body} \ (uP) \ ) \&\& \( wp \) \( \text{body} \ (tQ) \ ) \( \vdash \) \( wp \) \( \text{body} \ (uP) \ ) \&\& \( tQ) \)

by (auto intro: sub-distrib-pconjD[OF sdp-body])
also from \( IH \) \( \text{unitary-sound}[OF } uP[\text{OF } uQ] \) \( ft \)

\( \text{unitary-sound}[OF } uu[\text{OF } uP]\]

have \( \ldots \leq wp \) \( \text{body} \ (t \ (P \ ) \&\& \( Q)\) \)

by (blast intro: mono-transD[OF healthy-monoD, OF huq], exp-conj-sound)

finally have \( wp \) \( \text{body} \ (uP) \ s + wp \) \( \text{body} \ (tQ) \ s \sinter 1 \leq \)

\( wp \) \( \text{body} \ (t \ (\lambda s. \ P s + Q \ s \sinter 1)) \ s \)

by (auto simp: exp-conj-def pconj-def)

hence \( \langle G \rangle \ s * (wp \) \( \text{body} \ (uP) \ s + wp \) \( \text{body} \ (tQ) \ s \sinter 1 \) \( + \)

\( 1 - \langle G \rangle \ s \rangle * (P \ s + Q \ s \sinter 1) \leq \)

\( \langle G \rangle \ s * wp \) \( \text{body} \ (t \ (\lambda s. \ P s + Q \ s \sinter 1)) \ s + \)

\( 1 - \langle G \rangle \ s \rangle * (P \ s + Q \ s \sinter 1) \)

by (auto intro: add-right-mono mult-left-mono)
\}

finally

show \( \langle G \rangle \ s * wp \) \( \text{body} \ (uP) \ s + (1 - \langle G \rangle \ s) * \ P \) \( s + \)

\( \langle G \rangle \ s * wp \) \( \text{body} \ (tQ) \ s + (1 - \langle G \rangle \ s) * \ Q \) \( \sinter 1 \leq \)

\( \langle G \rangle \ s * wp \) \( \text{body} \ (t \ (\lambda s. \ P s + Q \ s \sinter 1)) \ s + \)
\( 1 - \langle G \rangle \ s \rangle * (P \ s + Q \ s \sinter 1) \).

done
\textbf{fix} P, Q::'s expect \textbf{and} t t' u u'::'s trans
\textbf{assume} unitary P unitary Q
\quad equiv-trans t t' equiv-utrans u u'
\quad u P \&\& t Q \vdash t (P \&\& Q)
\textbf{thus} u' P \&\& t' Q \vdash t' (P \&\& Q)
\quad \textbf{by}(simp add:equiv-transD unitary-sound equiv-utransD exp-conj-unitary)
\textbf{qed}

\textbf{lemmas} sdp-intros =
\qquad sdp-Abort sdp-Skip sdp-Apply
\qquad sdp-Seq sdp-DC sdp-PC
\qquad sdp-SetPC sdp-SetDC sdp-Embed
\qquad sdp-repeat sdp-Bind sdp-loop

\subsection{The Well-Defined Predicate.}
\textbf{definition}
\quad well-def :: 's prog \Rightarrow bool
\textbf{where}
\quad well-def prog \equiv healthy (wp prog) \land nearly-healthy (wlp prog)
\quad \quad \land wp-under-wlp prog \land sub-distrib-pconj prog
\quad \quad \land sublinear (wp prog) \land bd-cts (wp prog)

\textbf{lemma} well-defI[intro]:
\quad \quad \begin{array}{l}
\begin{array}{l}
\text{healthy (wp prog); nearly-healthy (wlp prog);
\quad wp-under-wlp prog; sub-distrib-pconj prog; sublinear (wp prog);
\quad bd-cts (wp prog)}
\end{array}
\end{array}
\quad \Rightarrow
\quad \text{well-def prog}
\quad \quad \textbf{unfolding} well-def-def \textbf{by}(simp)

\textbf{lemma} well-def-wp-healthy[dest]:
\quad \text{well-def prog} \Rightarrow healthy (wp prog)
\quad \quad \textbf{unfolding} well-def-def \textbf{by}(simp)

\textbf{lemma} well-def-wlp-nearly-healthy[dest]:
\quad \text{well-def prog} \Rightarrow nearly-healthy (wlp prog)
\quad \quad \textbf{unfolding} well-def-def \textbf{by}(simp)

\textbf{lemma} well-def-wp-under[dest]:
\quad \text{well-def prog} \Rightarrow wp-under-wlp prog
\quad \quad \textbf{unfolding} well-def-def \textbf{by}(simp)

\textbf{lemma} well-def-sdp[dest]:
\quad \text{well-def prog} \Rightarrow sub-distrib-pconj prog
\quad \quad \textbf{unfolding} well-def-def \textbf{by}(simp)

\textbf{lemma} well-def-wp-sublinear[dest]:
\quad \text{well-def prog} \Rightarrow sublinear (wp prog)
\quad \quad \textbf{unfolding} well-def-def \textbf{by}(simp)
4.7. WELL-DEFINED PROGRAMS.

```
lemma well-def-wp-cts[dest]:
  well-def prog ⇒ bd-cts (wp prog)
unfolding well-def-def by(simp)

lemmas wd-dests =
  well-def-wp-healthy well-def-wlp-nearly-healthy
  well-def-wp-undcr well-def-sdp
  well-def-wp-sublinear well-def-wp-cts

lemma wd-Abort:
  well-def Abort
by(blast intro: healthy-wp-Abort nearly-healthy-wlp-Abort
  wp-under-wlp-Abort sdp-Abort sublinear-wp-Abort
  cts-wp-Abort)

lemma wd-Skip:
  well-def Skip
by(blast intro: healthy-wp-Skip nearly-healthy-wlp-Skip
  wp-under-wlp-Skip sdp-Skip sublinear-wp-Skip
  cts-wp-Skip)

lemma wd-Apply:
  well-def (Apply f)
by(blast intro: healthy-wp-Apply nearly-healthy-wlp-Apply
  wp-under-wlp-Apply sdp-Apply sublinear-wp-Apply
  cts-wp-Apply)

lemma wd-Seq:
  [ well-def a; well-def b ] ⇒ well-def (a ;; b)
by(blast intro: healthy-wp-Seq nearly-healthy-wlp-Seq
  wp-under-wlp-Seq sdp-Seq sublinear-wp-Seq
  cts-wp-Seq)

lemma wd-PC:
  [ well-def a; well-def b; unitary P ] ⇒ well-def (a ⊎ b)
by(blast intro: healthy-wp-PC nearly-healthy-wlp-PC
  wp-under-wlp-PC sdp-PC sublinear-wp-PC
  cts-wp-PC)

lemma wd-DC:
  [ well-def a; well-def b ] ⇒ well-def (a ⋂ b)
by(blast intro: healthy-wp-DC nearly-healthy-wlp-DC
  wp-under-wlp-DC sdp-DC sublinear-wp-DC
  cts-wp-DC)

lemma wd-SetDC:
  [ ∀x s. x ∈ S s ⇒ well-def (a x); ∃s. S s ≠ {}; ∃s. finite (S s) ] ⇒ well-def (SetDC a S)
```
by (simp add: cts-wp-SetDC ex-in-conv healthy-intros(17) healthy-intros(18) sdp-intros(8) sublinear-intros(8) well-def-def wp-under-wlp-intros(8))

lemma wd-SetPC:
\[
\begin{align*}
\forall x. s \in (\text{supp}\ (p\ s)) \Rightarrow \text{well-def}\ (a\ x); \\
\forall s. \text{unitary}\ (p\ s); \\
\forall s. \text{finite}\ (\text{supp}\ (p\ s)); \\
\forall s. \text{sum}\ (p\ s) (\text{supp}\ (p\ s)) \leq 1 \Rightarrow \text{well-def}\ (\text{SetPC}\ a\ p)
\end{align*}
\]
by (iprover intro: well-defI healthy-wp-SetPC nearly-healthy-wlp-SetPC wp-under-wlp-SetPC sdp-SetPC sublinear-wp-SetPC cts-wp-SetPC dest:wd-dests unitary-sound sound-nneg)

lemma wd-Embed:
fixes t :: 's trans
assumes ht: healthy t and st: sublinear t and ct: bd-cts t
shows well-def (Embed t)
proof (intro well-defI)
from ht show healthy (wp (Embed t)) nearly-healthy (wlp (Embed t))
  by (simp add: wp-def wp-def Embed-def healthy-nearly-healthy)+
from st show sublinear (wp (Embed t)) by (simp add: wp-def 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]]
      simp: exp-conj-def pconj-def wp-def wlp-def Embed-def)
from ct show bd-cts (wp (Embed t))
  by (simp add: wp-def Embed-def)
qed

lemma wd-repeat:
well-def a \Rightarrow well-def (repeat n a)

lemma wd-Bind:
\[
\begin{align*}
\forall s. \text{well-def}\ (a\ (f\ s)) \Rightarrow \text{well-def}\ (\text{Bind}\ f\ a)
\end{align*}
\]

lemma wd-loop:
well-def body \Rightarrow well-def (do G \rightarrow body od)

lemmas wd-intros =
wdblind L wdblind R

wd-Abort wd-Skip wd-Apply
wd-Embed wd-Seq wd-PC
wd-DC wd-SetPC wd-SetDC
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:
Λ 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:
Λ 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 ≤ wlp do G → body od ?P)
proof –
  let ?f Q s = «G» s * wlp body Q s + «N G» s * ?P s
  have I ⊢ ⊢ gfp-exp ?f
    proof (rule gfp-exp-upperbound[OF - uI])
      have I = (λs. («G» s + «N G» s) * I s)
        by(simp add:negate-embed)
      also have ... = (λs. «G» s * I s + «N G» s * I s)
        by(simp add:algebra-simps)
      also have ... = (λs. «G» s * («G» s * I s) + «N G» s * («N G» s * I s))
        by(simp add:embed-bool-idem algebra-simps)
      also have ... ⊢ (λs. «G» s * wlp body I s + «N G» s * «N G» s * I s)
        using inv by(auto dest:wlp-invD intro:add-mono mult-left-mono)
      finally show I ⊢ (λs. «G» s * wlp body I s + «N G» s * «N G» s * I s)) .
    qed
  also from uI well-def-wlp-nearly-healthy[OF wd] have ...
    = wlp do G → body od ?P
    by(auto intro:wlp-Loop1[symmetric] unitary-intros)
  finally show thesis .
qed

4.8.3 Total Correctness

The first total correctness lemma for loops which terminate with probability 1 [McIver and Morgan, 2004, Lemma 7.3.1, §7, p. 186].

lemma wp-Loop:
  assumes wd: well-def body
            and inv: wlp-inv G body I
and unit: unitary I
shows I & & wp (do G → body od) (λs. 1) ⊢ wp (do G → body od) (λs. «N
G» s * I s) (is I & & ?T ⊢ wp ?loop ?X)
proof −

We first appeal to the liberal loop rule:

from assms have I & & ?T ⊢ wp ?loop ?X & & ?T
by (blast intro: exp-conj-mono-left wlp-Loop)

Next, by sub-conjunctivity:

also {
from wd have sdp-loop: sub-distrib-pconj (do G → body od)
by (blast intro: sdp-intros)

from wd unit have wp ?loop ?X & & ?T ⊢ wp (?X & & (λs. 1))
by (blast intro: sub-distrib-pconjD sdp-intros unitary-intros)
}

Finally, the conjunction collapses:

finally show ?thesis
by (simp add: exp-conj-1-right sound-intros sound-nneg unit unitary-sound)
qed

4.8.4 Unfolding

lemma wp-loop-unfold:
fixes body :: ′s prog
assumes sP: sound P
and h: healthy (wp body)
shows wp (do G → body od) P =
(λs. «N G» s * P s + «G» s * wp body (wp (do G → body od) P) s)
proof (simp only: wp-eval)
let ?X t = wp (body ;; Embed t « G » ⊕ Skip)
have equiv-trans (lfp-trans ?X)
  (wp (body ;; Embed (lfp-trans ?X) « G » ⊕ Skip))
proof (intro lfp-trans-unfold)
fix t::′s trans and P::′s expect
assume st: ∨ Q. sound Q → sound (t Q)
  and sP: sound P
with h show sound (?X t P)
by (rule wp-loop-step-sound)
next
fix t u::′s trans
assume le-trans t u (∨ P. sound P → sound (t P))
  (∨ P. sound P → sound (u P))
with h show le-trans (wp (body ;; Embed t « G » ⊕ Skip))
  (wp (body ;; Embed u « G » ⊕ Skip))
by (iprover intro: wp-loop-step-mono)
next
  let ?v = \ P s. bound-of P
from h show le-trans (wp (body ;; Embed ?v « G » ⊕ Skip)) ?v
  by (intro le-transI, simp add: wp-eval lfp-loop-fp unfolded negate-embed)
fix P::'s expect
assume sound P thus sound (?v P) by (auto)
qed
also have equiv-trans ... (\ P s. « N » s * P s + « G » s * wp body (wp (Embed (lfp-trans ?X)) P) s)
  by (rule equiv-transI, simp add: wp-eval algebra-simps negate-embed)
finally show lfp-trans ?X P = (\ s. « N » s * P s + « G » s * wp body (lfp-trans ?X P) s)
  using sP unfolding wp-eval by (blast)
qed

lemma wp-loop-nguard:
  [ healthy (wp body); sound P; \ G s ] \⇒ wp do G → body od P s = P s
  by (subst wp-loop-unfold, simp-all)

lemma wp-loop-guard:
  [ healthy (wp body); sound P; G s ] \⇒ wp do G → body od P s = wp (body ;; do G → body od) P s
  by (subst wp-loop-unfold, simp-all add: wp-eval)

end

4.9 The Algebra of pGCL

theory Algebra imports WellDefined begin

Programs in pGCL have a rich algebraic structure, largely mirroring that for GCL. We show that programs form a lattice under refinement, with \cap b and \cup 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 ⇒ 's prog ⇒ bool (infix ⊑)
where
  prog ⊑ prog' ≡ \ P. sound P → wp prog P ⊢ wp prog' P

lemma refinesI [intro]:
4.9. *THE ALGEBRA OF PGCL*

\[ \forall P. \text{sound } P \implies \wp \text{ prog } P \Downarrow \wp \text{ prog}' P \implies \text{ prog } \sqsubseteq \text{ prog}' \]

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

**lemma** \text{refinesD[dest]}:
\[ \text{[ prog } \sqsubseteq \text{ prog}', \text{ sound } P ] \implies \wp \text{ prog } P \Downarrow \wp \text{ prog}' P \]

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

The equivalence relation below will turn out to be that induced by refinement. It is also the application of \text{equiv-trans} to the weakest precondition.

**definition**
\[ \text{pequiv} :: \langle \text{ prog } \Rightarrow \text{ prog } \Rightarrow \text{ bool } \rangle \text{ (infix } \simeq \text{ 70)} \]

**where**
\[ \text{ prog } \simeq \text{ prog}' \equiv \forall P. \text{ sound } P \implies \wp \text{ prog } P = \wp \text{ prog}' P \]

**lemma** \text{pequivI[ intro]}:
\[ \text{[ } \forall P. \text{ sound } P \implies \wp \text{ prog } P = \wp \text{ prog}' P \text{ ] } \implies \text{ prog } \simeq \text{ prog}' \]

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

**lemma** \text{pequivD[dest,simp]}:
\[ \text{[ prog } \simeq \text{ prog}', \text{ sound } P ] \implies \wp \text{ prog } P = \wp \text{ prog}' P \]

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

**lemma** \text{pequiv-equiv-trans}:
\[ a \simeq b \iff \text{equiv-trans } (\wp a) (wp b) \]
\text{by(auto)}

**4.9.2 Simple Identities**

The following identities involve only the primitive operations as defined in Section 4.1.1, and refinement as defined above.

**Laws following from the basic arithmetic of the operators separately**

**lemma** \text{DC-comm[ac-simps]}:
\[ a \sqcap b = b \sqcap a \]

**unfolding** \text{ DC-def} \text{by(simp add:ac-simps)}

**lemma** \text{DC-assoc[ac-simps]}:
\[ a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c \]

**unfolding** \text{ DC-def} \text{by(simp add:ac-simps)}

**lemma** \text{DC-idem}:
\[ a \sqcap a = a \]

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

**lemma** \text{AC-comm[ac-simps]}:
\[ a \sqcup b = b \sqcup a \]
unfolding AC-def by(simp add:ac-simps)

lemma AC-assoc[ac-simps]:
  a ⨆ (b ⨆ c) = (a ⨆ b) ⨆ c
unfolding AC-def by(simp add:ac-simps)

lemma AC-idem:
  a ⨆ a = a
unfolding AC-def by(simp)

lemma PC-quasi-comm:
  a ⊕ b = b (λs. t − p) ⊕ a
unfolding PC-def by(simp add:algebra-simps)

lemma PC-idem:
  a ⊕ a = a
unfolding PC-def by(simp add:algebra-simps)

lemma Seq-assoc[ac-simps]:
  A ;; (B ;; C) = A ;; B ;; C
by(simp add:Seq-def o-def)

lemma Abort-refines[intro]:
  well-def a ⇒ Abort ⊑ a
by(rule refinesI, unfold wp-eval, auto dest!:well-def-wp-healthy)

Laws relating demonic choice and refinement

lemma left-refines-DC:
  (a ⨆ b) ⊑ a
by(auto intro!:refinesI simp:wp-eval)

lemma right-refines-DC:
  (a ⨆ b) ⊑ b
by(auto intro!:refinesI simp:wp-eval)

lemma DC-refines:
  fixes a::'s prog and b and c
  assumes rab: a ⊑ b and rac: a ⊑ c
  shows a ⊑ (b ⨆ c)
proof
  fix P::'s ⇒ real assume sP: sound P
  with assms have wp a P ⊨ wp b P and wp a P ⊨ wp c P
  by(auto dest:refinesD)
  thus wp a P ⊨ wp (b ⨆ 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 $rab: a \sqsubseteq b$ and $rcd: c \sqsubseteq d$ 
shows $(a \sqcap c) \sqsubseteq (b \sqcap d)$ 
proof(rule refinesI, unfold wp-eval, rule le-funI) 
fix $P::'s$ $\Rightarrow$ real and $s::'s$ 
assume $sP$: 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: 
$a \sqsubseteq (a \sqcup b)$ 
by(auto intro!:refinesI simp:wp-eval)

lemma right-refines-AC: 
$b \sqsubseteq (a \sqcup b)$ 
by(auto intro!:refinesI simp:wp-eval)

lemma AC-refines: 
fixes $a::'s$ prog and $b$ and $c$ 
assumes $rac: a \sqsubseteq c$ and $rbc: b \sqsubseteq c$ 
shows $(a \sqcup b) \sqsubseteq c$ 
proof 
fix $P::'s$ $\Rightarrow$ real assume $sP$: sound $P$ 
with assms have $\bigwedge s. wp a P s \leq wp b P s$ 
and $\bigwedge s. wp b P s \leq wp c P s$ 
by(auto dest:refinesD) 
thus $wp (a \sqcup b) P \gg wp c P$ 
unfolding wp-eval by(auto) 
qed

lemma AC-mono: 
fixes $a::'s$ prog 
assumes $rab: a \sqsubseteq b$ and $rcd: c \sqsubseteq d$ 
shows $(a \sqcup c) \sqsubseteq (b \sqcup d)$ 
proof(rule refinesI, unfold wp-eval, rule le-funI) 
fix $P::'s$ $\Rightarrow$ real and $s::'s$ 
assume $sP$: 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 (rule $\mathsf{refinesI}$, unfold $wp$-eval, rule $\mathsf{le-funI}$)
  fix $s$ and $P :: \forall a \Rightarrow \mathbb{R}$
  assume sound: sound $P$
  from unit have $nn$-$p$: $0 \leq p \ s$ by (blast)
  from unit have $p \ s \leq 1$ by (blast)
  hence $nn$-$np$: $0 \leq 1 - p \ s$ by (simp)
  show $\min (wp \ a \ P \ s) (wp \ b \ P \ s) \leq p \ s * wp \ a \ P \ s + (1 - p \ s) * wp \ b \ P \ s$
proof (cases $wp \ a \ P \ s \leq wp \ b \ P \ s$,
  simp-all add: $\min$.absorb1 $\min$.absorb2)
  case True
  note $le$ = this
  have $wp \ a \ P \ s = (p \ s + (1 - p \ s)) * wp \ a \ P \ s$ by (simp)
  also have ... = $p \ s * wp \ a \ P \ s + (1 - p \ s) * wp \ a \ P \ s$
    by (simp only: distrib-right)
  also {
    from $le$ and $nn$-$np$ have $(1 - p \ s) * wp \ a \ P \ s \leq (1 - p \ s) * wp \ b \ P \ s$
      by (rule $\mathsf{mult-left-mono}$)
    hence $p \ s * wp \ a \ P \ s + (1 - p \ s) * wp \ a \ P \ s \leq$
      $p \ s * wp \ b \ P \ s + (1 - p \ s) * wp \ b \ P \ s$
      by (rule $\mathsf{add-left-mono}$)
  }
  finally show $wp \ a \ P \ s \leq p \ s * wp \ a \ P \ s + (1 - p \ s) * wp \ b \ P \ s$.
next
  case False
  then have $le$: $wp \ b \ P \ s \leq wp \ a \ P \ s$ by (simp)
  have $wp \ b \ P \ s = (p \ s + (1 - p \ s)) * wp \ b \ P \ s$ by (simp)
  also have ... = $p \ s * wp \ b \ P \ s + (1 - p \ s) * wp \ b \ P \ s$
    by (simp only: distrib-right)
  also {
    from $le$ and $nn$-$p$ have $p \ s * wp \ b \ P \ s \leq p \ s * wp \ a \ P \ s$
      by (rule $\mathsf{mult-left-mono}$)
    hence $p \ s * wp \ b \ P \ s + (1 - p \ s) * wp \ b \ P \ s \leq$
      $p \ s * wp \ a \ P \ s + (1 - p \ s) * wp \ b \ P \ s$
      by (rule $\mathsf{add-right-mono}$)
  }
  finally show $wp \ b \ P \ s \leq p \ s * wp \ a \ P \ s + (1 - p \ s) * wp \ b \ P \ s$.
qed
qed

Laws depending on the arithmetic of $a \oplus b$ and $a \sqcup b$ together

lemma $PC$-refines-$AC$:
  assumes unit: unitary $p$
  shows $(a \oplus b) \sqsubseteq (a \sqcup b)$
proof (rule $\mathsf{refinesI}$, unfold $wp$-eval, rule $\mathsf{le-funI}$)
  fix $s$ and $P :: \forall a \Rightarrow \mathbb{R}$
  assume sound: sound $P$
from unit have \( p s \leq 1 \) by (blast)
hence \( \text{nn-np}: 0 \leq 1 - p s \) by (simp)

show \( p s \cdot \wp a P s + (1 - p s) \cdot \wp b P s \leq \max (\wp a P s) (\wp b P s) \)

proof (cases \( \wp a P s \leq \wp b P s \))
  case True note leab = this
  with unit nn-np
  have \( p s \cdot \wp a P s + (1 - p s) \cdot \wp b P s \leq \)
    \( p s \cdot \wp b P s + (1 - p s) \cdot \wp a P s \)
  by (auto intro: add-mono mult-left-mono)
  also have \( \ldots = \wp b P s \)
    by (auto simp: field-simps)
  also from leab
  have \( \ldots = \max (\wp a P s) (\wp b P s) \)
    by (auto)
  finally show \( ?\text{thesis} \).
next
  case False note leba = this
  with unit nn-np
  have \( p s \cdot \wp a P s + (1 - p s) \cdot \wp b P s \leq \)
    \( p s \cdot \wp a P s + (1 - p s) \cdot \wp a P s \)
  by (auto intro: add-mono mult-left-mono)
  also have \( \ldots = \wp a P s \)
    by (auto simp: field-simps)
  also from leba
  have \( \ldots = \max (\wp a P s) (\wp b P s) \)
    by (auto)
  finally show \( ?\text{thesis} \).
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 \text{real} assume sP: sound P
  with prAB have \( \wp a P \vdash \wp b P \) by (blast)
  also from sP and prBC have \ldots \vdash \wp C P by (blast)
  finally show \( \wp a P \vdash \ldots \).
chapter 4. the pgcl language

qed

lemma pequiv-refl[intro,simp]:
  \( a \simeq a \)
  by(auto)

lemma pequiv-comm[ac-simps]:
  \( a \simeq b \iff b \simeq a \)
  unfolding pequiv-def
  by(rule iffI, safe, simp-all)

lemma pequiv-pr[dest]:
  \( a \simeq b \rightarrow a \sqsubseteq b \)
  by(auto)

lemma pequiv-trans[intro,trans]:
  \[ [ a \simeq b ; b \sqsubseteq c ] \] \( \Rightarrow a \simeq c \)
  unfolding pequiv-def by(auto intro:order-trans)

lemma pequiv-pr-trans[intro,trans]:
  \[ [ a \sqsubseteq b ; b \simeq c ] \] \( \Rightarrow a \sqsubseteq c \)
  unfolding pequiv-def refines-def by(simp)

lemma pr-pequiv-trans[intro,trans]:
  \[ [ a \sqsubseteq b ; b \simeq c ] \] \( \Rightarrow a \sqsubseteq c \)
  unfolding pequiv-def refines-def by(simp)

Refinement induces equivalence by antisymmetry:

lemma pequiv-antisym:
  \[ [ a \sqsubseteq b ; b \sqsubseteq a ] \] \( \Rightarrow a \simeq b \)
  by(auto intro:antisym)

lemma pequiv-DC:
  \[ [ a \simeq c ; b \simeq d ] \] \( \Rightarrow (a \sqcap b) \simeq (c \sqcap d) \)
  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

\begin{align*}
\text{and } wa &\text{: well-def } a \\
\text{and } wb &\text{: well-def } b \\
\text{and } dr &\text{: } a \sqsubseteq b \\
\text{shows } a &\simeq b
\end{align*}

Proof by contradiction.

\begin{proof}
\text{proof (rule pequivI, rule contrapos-pp)} \\
\text{from } wb \text{ have feasible (wp b) by(auto)} \\
\text{with } wb \text{ have } sab \text{: sub-add (wp b)} \\
\text{by (auto dest: sublinear-subadd OF well-def-wp-sublinear)} \\
\text{fix } P \text{:'s } \Rightarrow \text{ real assume } sP \text{: sound } P
\end{proof}

Assume that \( a \) and \( b \) are not equivalent:

\begin{proof}
\text{assume } ne \text{: wp } a \neq wp b P
\end{proof}

Find a point at which they differ. As \( a \sqsubseteq b \), \( wp b P s \) must be strictly greater than \( wp a P s \) here:

\begin{proof}
\text{hence } \exists s. \ wp a P s < wp b P s \\
\text{proof (rule contrapos-np)} \\
\text{assume } \lnot (\exists s. \ wp a P s < wp b P s) \\
\text{hence } \forall s. \ wp b P s \leq wp a P s \text{ by(auto simp: not-less)} \\
\text{hence } wp b P \vdash wp a P \text{ by(auto)} \\
\text{moreover from } sP \text{ dr have wp } a P \vdash wp b P \text{ by(auto)} \\
\text{ultimately show wp } a P = wp b P \text{ by(auto)} \\
\text{qed}
\end{proof}

\begin{proof}
\text{then obtain } s \text{ where less: wp } a P s < wp b P s \text{ by(blast)}
\end{proof}

Take a carefully constructed expectation:

\begin{proof}
\text{let } ?Pc = \lambda s. \text{ bound-of } P - P s \\
\text{have } sPc \text{: sound } ?Pc \\
\text{proof (rule soundI)} \\
\text{from } sP \text{ have } \forall s. \ 0 \leq P s \text{ by(auto)} \\
\text{hence } \forall s. \ ?Pc s \leq \text{bound-of } P \text{ by(auto)} \\
\text{thus } \text{bounded } ?Pc \text{ by(blast)} \\
\text{from } sP \text{ have } \forall s. \ P s \leq \text{bound-of } P \text{ by(auto)} \\
\text{hence } \forall s. \ 0 \leq ?Pc s \text{ by(auto simp:sign-simps)} \\
\text{thus } \text{nneg } ?Pc \text{ by(auto)} \\
\text{qed}
\end{proof}

We then show that \( wp b \) violates feasibility, and thus healthiness.

\begin{proof}
\text{from } sP \text{ have } 0 \leq \text{bound-of } P \text{ by(auto)} \\
\text{with } da \text{ have } \text{bound-of } P = wp a (\lambda s. \text{bound-of } P) s \\
\text{by(simp add:maximalD determ-maximalD)} \\
\text{also have } ... = wp a (\lambda s. ?Pc s + P s) s \\
\text{by(simp)} \\
\text{also from } da sP sPc \text{ have } ... = wp a ?Pc s + wp a P s \\
\text{by(subst additiveD[OF determ-additiveD], simp-all add:sP sPc)} \\
\text{also from } sPc \text{ dr have } ... \leq wp b ?Pc s + wp a P s \\
\text{by(auto)}
\end{proof}
also from \texttt{less} have ... \( < \text{wp } b \ ?Pc s + \text{wp } b \ P \ s \) 
 by(\texttt{auto})
also from \texttt{sab} \( sPc \) \texttt{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,

\begin{verbatim}
fix \( P::'s \Rightarrow \text{real} \) assume \( sP::\text{sound } P \)

hence \( \neg \text{neq} \ (\lambda s. \ \text{bound-of } P) \) by(\texttt{auto})
moreover have \( \text{bounded-by} \ (\text{bound-of } P) \ (\lambda s. \ \text{bound-of } P) \) by(\texttt{auto})

ultimately

show \( \text{bounded-by} \ (\text{bound-of } P) \ (\text{wp } b \ (\lambda s. \ \text{bound-of } P)) \) 
 using \( \text{wb} \) by(\texttt{auto dest!:well-def-wp-healthy})
\end{verbatim}

\texttt{qed}

4.9.4 The Algebraic Structure of Refinement

Well-defined programs form a half-bounded semilattice under refinement, where \( \text{Abort} \) is bottom, and \( a \sqcap b \) is \( \text{inf} \). There is no unique top element, but all fully-deterministic programs are maximal.

The type that we construct here is not especially useful, but serves as a convenient way to express this result.

\texttt{quotient-type 's program =}

\texttt{'s prog / partial : \lambda a. \ a \simeq b \land well-def a \land well-def b}

\texttt{proof(\texttt{rule part-equivpI})}

\texttt{have Skip \simeq Skip and well-def Skip by(\texttt{auto intro:wd-intros})}

\texttt{thus \( \exists x. \ x \simeq x \land \text{well-def } x \land \text{well-def } x \) by(\texttt{blast})}

\texttt{show \text{symp} \ (\lambda a. \ a \simeq b \land \text{well-def } a \land \text{well-def } b) \ (\texttt{rule sympI, safe})}

\texttt{fix \( a::'a \text{ prog and } b \)}

\texttt{assume \( a \simeq b \)}

\texttt{hence \text{equiv-trans} \ (wp a) \ (wp b)}

\texttt{by(\texttt{simp add:pequiv-equiv-trans})}

\texttt{thus \( b \simeq a \) by(\texttt{simp add:ac-simps pequiv-equiv-trans})}

\texttt{qed}

\texttt{show \text{transp} \ (\lambda a. \ a \simeq b \land \text{well-def } a \land \text{well-def } b) \ (\texttt{rule transpI, safe, rule pequiv-trans})}

\texttt{qed}

\texttt{instantiation \text{program} :: \text{(type) \text{semilattice-inf}} begin}

\texttt{lift-definition \texttt{less-eq-program} :: 'a \text{ program } \Rightarrow 'a \text{ program } \Rightarrow \text{bool} \Rightarrow \text{refines}}

\texttt{proof(\texttt{safe})}

\texttt{fix \( a::'a \text{ prog and } b \ c \ d \)}
assume \( a \simeq b \) hence \( b \simeq a \) by \((simp add:ac-simps)\)
also assume \( a \subseteq c \)
also assume \( c \simeq d \)
finally show \( b \subseteq d \).

next
fix \( a::'a prog \) and \( b \, c \, d \)
assume \( a \simeq b \)
also assume \( b \subseteq d \)
also assume \( c \simeq d \) hence \( d \simeq c \) by \((simp add:ac-simps)\)
finally show \( a \subseteq c \).

qed

lift-definition
\texttt{less-program} :: 'a program ⇒ 'a program ⇒ bool
is \( \lambda a \, b. \, a \subseteq b \land \neg b \subseteq a \)

proof\((\text{safe})\)
fix \( a::'a prog \) and \( b \, c \, d \)
assume \( a \simeq b \) hence \( b \simeq a \) by \((simp add:ac-simps)\)
also assume \( a \subseteq c \)
also assume \( c \simeq d \)
finally show \( b \subseteq d \).

next
fix \( a::'a prog \) and \( b \, c \, d \)
assume \( a \simeq b \)
also assume \( b \subseteq d \)
also assume \( c \simeq d \) hence \( d \simeq c \) by \((simp add:ac-simps)\)
finally show \( a \subseteq c \).

next
fix \( a \, b \) and \( c::'a prog \) and \( d \)
assume \( c \simeq d \)
also assume \( d \subseteq b \)
also assume \( a \simeq b \) hence \( b \simeq a \) by \((simp add:ac-simps)\)
finally have \( c \subseteq a \).
moreover assume \( \neg c \subseteq a \)
ultimately show \( \text{False} \) by\((auto)\)

next
fix \( a \, b \) and \( c::'a prog \) and \( d \)
assume \( c \simeq d \) hence \( d \simeq c \) by \((simp add:ac-simps)\)
also assume \( c \subseteq a \)
also assume \( a \simeq b \)
finally have \( d \subseteq b \).
moreover assume \( \neg d \subseteq b \)
ultimately show \( \text{False} \) by\((auto)\)

qed

lift-definition
\texttt{inf-program} :: 'a program ⇒ 'a program ⇒ 'a program is \texttt{DC}

proof\((\text{safe})\)
fix \( a \, b \, c \, d::'s prog \)
assume $a \approx b$ and $c \approx d$
thus $(a \sqcap c) \approx (b \sqcap d)$ 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 prog
show $(x < y) = (x \leq y \land \neg y \leq x)$
by (transfer, simp)

show $x \leq x$
by (transfer, auto)

show $\text{inf } x \leq x$
by (transfer, rule left-refines-DC)

show $\text{inf } x \leq y$
by (transfer, rule right-refines-DC)

assume $x \leq y$ and $y \leq x$ hence $x = y$
by (transfer, iprover intro: pequiv-antisym)

next
fix $x$ $y$ $z$ :: 'a prog
assume $x \leq y$ and $y \leq z$
thus $x \leq z$
by (transfer, iprover intro: pr-trans)

next
fix $x$ $y$ $z$ :: 'a prog
assume $x \leq y$ and $x \leq z$
thus $x \leq \text{inf } y z$
by (transfer, iprover intro: DC-refines)

qed

end

instantiation program :: (type) bot begin
lift-definition
bot-program :: 'a prog is Abort
by (auto intro: wd-intros)

instance ..
end

lemma eq-det: \(\forall a \ b\::\ 's\ prog. \ [ a \approx b;\ \text{determ}\ \wp\ a \ ] \implies\ \text{determ}\ \wp\ b\)

proof
(intro determI additiveI maximalI)
fix $a$ :: 's prog and $P$ :: 's ⇒ real
and Q.'s \Rightarrow real and s.'s
assume da: determ (wp a)
assume sP: sound P and sQ: sound Q
and eq: a \simeq b
hence wp b (\lambda s. P s + Q s) s =
wp a (\lambda s. P s + Q s) s
by(simp add:sound-intros)
also from da sP sQ
have ...
by(simp add: additiveD determ-additiveD)
also from eq sP sQ
have ...
by(simp add: pequivD)
finally show wp b (\lambda s. P s + Q s) s = wp a P s + wp a Q s.

next
fix a b::'s prog and c::real
assume a: determ (wp a)
assume a \simeq b hence b \simeq a by(simp add: ac-simps)
moreover assume nn: 0 \leq c
ultimately have wp b (\lambda-. c) = wp a (\lambda-. c)
by(simp add: pequivD const-sound)
also from da nn have ...
by(simp add: determ-maximalD maximalD)
finally show wp b (\lambda-. c) = (\lambda-. c).

qed

lift-definition
p determ :: 's program \Rightarrow bool
is \lambda a. determ (wp a)
proof(safe)
fix a b::'s prog
assume a \simeq b and determ (wp a)
thus determ (wp b) by(rule eq-det)

next
fix a b::'s prog
assume a \simeq b hence b \simeq a by(simp add: ac-simps)
moreover assume determ (wp b)
ultimately show determ (wp a) by(rule eq-det)

qed

lemma determ-maximal:
[ p determ a; a \leq x ] \Rightarrow a = x
by(transfer, auto intro:refines-determ)

4.9.5 Data Refinement
A projective data refinement construction for pGCL. By projective, we mean
that the abstract state is always a function (\varphi) of the concrete state. Refinements may be predicated (G) on the state.
定义

defines :: ('b ⇒ 'a) ⇒ ('b ⇒ bool) ⇒ 'a prog ⇒ 'b prog ⇒ bool

where

defines ϕ G A B ≡ ∀ P Q. (unitary P ∧ unitary Q ∧ (P ⊢ wp A Q)) →
("G" & (P o ϕ) ⊢ wp B (Q o ϕ))

定理 drefinesD[dest]:

[ drefines ϕ G A B; unitary P; unitary Q; P ⊢ wp A Q ] →
("G" & (P o ϕ) ⊢ wp B (Q o ϕ)

折叠 drefines-def by blast

我们也可以用 G 作为假设。

定理 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

证明 –

从 uP 有 0 ≤ (P o ϕ) s 不折叠 a-def by blast

以 G 有 (P o ϕ) s = ("G" & (P o ϕ)) s

以 simp add: exp-conj-def

也从 assum 有 … ≤ wp B (Q o ϕ) s by blast

最后显示 (P o ϕ) s ≤ ...

定理 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

证明 –

让 ?L s' = wp a Q s'

从 uQ wa 有 sl: sound ?L by blast

从 uQ wa 有 bl: bounded-by 1 ?L by blast

有 ?L ⊢ ?L by simp

与 sl 和 bl 和 assum

显示 ?thesis

by (blast intro: drefinesD2[OF dr, where P=?L, simplified])

定理 drefinesI[intro]:

[ A P Q, [ unitary P; unitary Q; P ⊢ wp A Q ] →
("G" & (P o ϕ) ⊢ wp B (Q o ϕ)) ] →

defines ϕ G A B

定理
The algebra of PGCL

Unfolding \textit{drefines-def} by (\texttt{blast})

Use G as an assumption, when showing refinement:

\textbf{lemma} \texttt{drefines}\texttt{l2}:
\begin{itemize}
    \item \texttt{fixes A::'a prog}
    \item and \texttt{B::'b prog}
    \item and \texttt{\varphi::'b \Rightarrow 'a}
    \item and \texttt{G::'b \Rightarrow bool}
    \item \texttt{assumes wB: well-def B}
    \item and \texttt{withAs: \\\& \\& (P \\&\& (Q \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \\&\& \"}
ers, modulo a guard. It is the analogue, for data refinement, of program equivalence for program refinement.

**definition**

\[ \text{pcorres} :: \left( 'b \Rightarrow 'a \right) \Rightarrow 'a \text{ prog} \Rightarrow 'b \text{ prog} \Rightarrow \text{bool} \]

**where**

\[ \text{pcorres} \varphi \ G \ A \ B \leftrightarrow \]

\[ (\forall Q. \text{unitary } Q \rightarrow «G» \&\& (\text{wp } A \ Q \ o \ \varphi) = «G» \&\& \text{wp } B \ (Q \ o \ \varphi)) \]

**lemma** \(\text{pcorresI}1\):\[
[ \forall Q. \text{unitary } Q \rightarrow «G» \&\& (\text{wp } A \ Q \ o \ \varphi) = «G» \&\& \text{wp } B \ (Q \ o \ \varphi) ] \implies \text{pcorres} \ \varphi \ G \ A \ B
\]

by (simp add:pcorres-def)

Often easier to use, as it allows one to assume the precondition.

**lemma** \(\text{pcorresI2}[intro]\):

fixes \(A::'a\text{ prog}\) and \(B::'b\text{ prog}\)

assumes \(\text{withG}: \left[ \forall Q. s \right. \left[ \text{unitary } Q; \ G \ s \right] \rightarrow \text{wp } A \ Q \ (\varphi \ s) = \text{wp } B \ (Q \ o \ \varphi)\ s\)

and \(wA::\text{well-def } A\)

and \(wB::\text{well-def } B\)

shows \(\text{pcorres} \ \varphi \ G \ A \ B\)

**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 \(\left( «G» \&\& (wp A Q o \ \varphi) \right) \ s = \left( «G» \&\& \text{wp } B \ (Q \ o \ \varphi) \right) \ s\)

**proof** (cases \(G \ s\))

- case True

  note this

  moreover

  from well-def-wp-healthy[OF wA] uQ have \(0 \leq \text{wp } A \ Q \ (\varphi \ s)\) by (blast)

  moreover

  from well-def-wp-healthy[OF wB] uQ\varphi have \(0 \leq \text{wp } B \ (Q \ o \ \varphi) \ s\) 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 \(\text{wp } A \ Q \ (\varphi \ s) \leq 1\) by (blast)

  moreover

  from well-def-wp-healthy[OF wB] uQ\varphi have \(\text{wp } B \ (Q \ o \ \varphi) \ s \leq 1\)

  by (blast dest!:healthy-bounded-byD intro:sound-nneg)

  ultimately show \(?thesis\) by (simp add:exp-conj-def)

qed

**lemma** \(\text{pcorresD}:\)

\[ [ \text{pcorres} \ \varphi \ G \ A \ B; \ \text{unitary } Q ] \rightarrow «G» \&\& (\text{wp } A \ Q \ o \ \varphi) = «G» \&\& \text{wp } B \ (Q \ o \ \varphi) \]

**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 \( \varphi \) \( G \ A \ B \)
- \( uQ: \) unitary \( Q \)
- \( wA: \) well-def \( A \) and \( wB: \) well-def \( B \)
- \( G: \) \( G \ s \)

**Shows**

\( \text{wp} \ A \ Q \ (\varphi \ s) = \text{wp} \ B \ (Q \circ \varphi \ s) \)

**Proof**

- From \( uQ \) well-def-wp-healthy\([\text{OF} \ wA]\) have \( \theta \leq \text{wp} \ A \ Q \ (\varphi \ s) \) by(auto)
- With \( G \) have \( \text{wp} \ A \ Q \ (\varphi \ s) = \langle G \rangle \ s \) & \( \text{wp} \ A \ Q \ (\varphi \ s) \) by(simp)

**Also**

- From \( pc \ uQ \) have \( \langle G \rangle \ & \ (\text{wp} \ A \ Q \circ \varphi) = \langle G \rangle \ & \ \text{wp} \ B \ (Q \circ \varphi) \) by(rule pcorresD)
- Hence \( \langle G \rangle \ s \) & \( \text{wp} \ A \ Q \ (\varphi \ s) = \langle G \rangle \ s \) & \( \text{wp} \ B \ (Q \circ \varphi) \ s \)

- 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 \( \text{well-def-wp-healthy}[\text{OF} \ wB] \) have \( \theta \leq \text{wp} \ B \ (Q \circ \varphi) \ s \) by(auto)
- With \( G \) have \( \langle G \rangle \ s \) & \( \text{wp} \ B \ (Q \circ \varphi) \ s = \text{wp} \ B \ (Q \circ \varphi) \ s \) by(simp)

**Finally show** \( \text{thesis} \).

**Qed**

**4.9.6 The Algebra of Data Refinement**

Program refinement implies a trivial data refinement:

**Lemma refines-drefines:**

- **Fixes** \( a::'s \ \text{prog} \)
- **Assumes** \( \text{rab} \): \( a \subseteq b \) and \( \text{wb}: \) well-def \( b \)
- **Shows** drefines \( (\lambda s. s) \ G \ a \ b \)

**Proof**

- **Intro** drefinesI2 \( \text{wb} \), simp add:o-def
- **Fix** \( P::'s \Rightarrow \text{real} \) and \( Q::'s \Rightarrow \text{real} \) and \( s::'s \)
- **Assume** \( sQ: \text{unitary} \ G \)
- **Assume** \( P \vdash \text{wp} \ a \ Q \) hence \( P \ s \leq \text{wp} \ a \ Q \ s \) by(auto)
- **Also from** \( \text{rab} \ sQ \) have \( ... \leq \text{wp} \ b \ Q \ s \) by(auto)
- **Finally show** \( P \ s \leq \text{wp} \ b \ Q \ s \)

**Qed**

Data refinement is transitive:

**Lemma dr-trans[trans]:**

- **Fixes** \( A::'a \ \text{prog} \) and \( B::'b \ \text{prog} \) and \( C::'c \ \text{prog} \)
- **Assumes** \( \text{drAB}: \) drefines \( \varphi \) \( G \ A \ B \)
- **And** \( \text{drBC}: \) drefines \( \varphi' \) \( G' \ B \ C \)
- **And** \( \text{Gimp}: \forall s. \ G' \ s \Longrightarrow G \ (\varphi' \ s) \)
- **Shows** drefines \( (\varphi \circ \varphi') \) \( G' \ A \ C \)

**Proof**(rule drefinesI)
CHAPTER 4. THE PGCL LANGUAGE

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

have \( "G'" \& \& "G \varphi'" = "G'" \)
proof\(\text{(rule ext, unfold exp-conj-def)}\)
  fix \( x \)
  show \( "G'" \ x \& "G \varphi'" \ x = "G'" \ x \ (\text{is } ?X) \)
  proof\(\text{(cases } G' \ x)\)
    case False then show \( ?X \) by\(\text{(simp)}\)
next
  case True moreover with \( G \text{imp} \) have \( (G \varphi') \ x \) by\(\text{(simp add: o-def)}\)
  ultimately show \( ?X \) by\(\text{(simp)}\)
qed

with \( uP \)
have \( "G'" \& (P \ (\varphi \ o \varphi')) = "G'" \& (\="G" \& (P \ \varphi)) \ o \varphi') \)
  by\(\text{(simp add: exp-conj-assoc o-assoc)}\)

also \{ 
  from \( uP \ uQ \ wpA \) and \( drAB \)
  have \( "G" \& (P \ \varphi) \vdash wp \ (Q \ o \varphi) \)
    by\(\text{(blast intro: drefinesD)}\)

  with \( drBC \) and \( uP \ uQ \)
  have \( "G'" \& (("G" \& (P \ \varphi)) \ o \varphi') \vdash wp \ C ((Q \ o \varphi) \ o \varphi') \)
    by\(\text{(blast intro: unitary-intros drefinesD)}\)
\}

finally
  show \( "G'" \& (P \ (\varphi \ o \varphi')) \vdash wp \ C ((Q \ o \varphi) \ o \varphi') \)
    by\(\text{(simp add: o-assoc)}\)
qed

Data refinement composes with program refinement:

lemma \( pr-dr-trans[trans] \):
assumes \( prAB :: A \sqsubseteq B \)
  and \( drBC :: \text{drefines} \varphi \ G \ B \ C \)
shows \( \text{drefines} \varphi \ G \ A \ C \)
proof\(\text{(rule drefinesI)}\)
  fix \( P \) and \( Q \)
  assume \( uP :: \text{unitary} \ P \)
    and \( uQ :: \text{unitary} \ Q \)
    and \( wpA :: P \vdash wp \ A \ Q \)
4.9. THE ALGEBRA OF PGCL

note wpA
also from uQ and prAB have wp A Q ⊨ wp B Q by (blast)
finally have P ⊨ wp B Q .
with uP uQ drBC
show «G» & & (P o ϕ) ⊨ wp C (Q o ϕ) by (blast intro: drefinesD)
qed

lemma dr-pr-trans[trans]:
assumes drAB: drefines ϕ G A B
assumes prBC: B ⊆ C
shows drefines ϕ G A C
proof (rule drefinesI)
fix P and Q
assume uP: unitary P
and uQ: unitary Q
and wpA: P ⊨ wp A Q

with drAB have «G» & & (P o ϕ) ⊨ wp B (Q o ϕ) by (blast intro: drefinesD)
also from uQ prBC have ... ⊨ wp C (Q o ϕ) by (blast)
finally show «G» & & (P o ϕ) ⊨ ... .
qed

If the projection ϕ commutes with the transformer, then data refinement is reflexive:

lemma dr-refl:
assumes wa: well-def a
and comm: ∀ Q. unitary Q → wp a Q o ϕ ⊨ wp a (Q o ϕ)
shows drefines ϕ G a a
proof (intro drefinesI2 wa)
fix P and Q and s
assume wp: P ⊨ wp a Q
assume uQ: unitary Q

have (P o ϕ) s = P (ϕ s) by (simp)
also from wp have ... ≤ wp a Q (ϕ s) by (blast)
also {
from comm uQ have wp a Q o ϕ ⊨ wp a (Q o ϕ) by (blast)
hence (wp a Q o ϕ) s ≤ wp a (Q o ϕ) s by (blast)
hence wp a Q (ϕ s) ≤ ... by (simp)
}
finally show (P o ϕ) s ≤ wp a (Q o ϕ) s .
qed

Correspondence implies data refinement

lemma pcorres-drefine:
assumes corres: pcorres ϕ G A C
and wC: well-def C
shows drefines ϕ G A C
proof
fix $P$ and $Q$
assume $uP$: unitary $P$ and $uQ$: unitary $Q$
and $wpA$: $P \vdash wp A Q$

from $wpA$ have $P \circ \varphi \vdash wp A Q \circ \varphi$ by (simp add: o-def le-fun-def)

hence $\langle G \rangle \&\& (P \circ \varphi) \vdash \langle G \rangle \&\& (wp A Q \circ \varphi)$

by (rule exp-conj-monoid-right)
also from $\text{corres} uQ$

have $... = \langle G \rangle \&\& (wp C (Q \circ \varphi))$ by (rule pcorresD)
also

have $... \vdash wp C (Q \circ \varphi)$
proof (rule le-funI)

fix $s$
from $uQ$ have 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 $(\langle G \rangle \&\& wp C (Q \circ \varphi)) s \leq wp C (Q \circ \varphi) s$

proof (cases $G s$)

case True

with $nn-wpC$ show $?\text{thesis}$ by (simp add: exp-conj-def)

next

case False note this

moreover {

from $uQ$ have 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 $?\text{thesis}$ by (simp add: exp-conj-def)

qed

qed

finally show $\langle G \rangle \&\& (P \circ \varphi) \vdash wp C (Q \circ \varphi)$.

qed

Any data refinement of a deterministic program is correspondence. This is the analogous result to that relating program refinement and equivalence.

lemma drefines-determ:
fixes $a::{'}a\text{ prog}$ and $b::{'}b\text{ prog}$
assumes $da$: determ $(wp a)$
and $wa$: well-def $a$
and $wb$: well-def $b$
and $dr$: drefines $G a b$
shows $\text{pcorres} \varphi G a b$

The proof follows exactly the same form as that for program refinement: Assuming that correspondence doesn’t hold, we show that $wp b$ is not feasible, and thus not healthy, contradicting the assumption.

proof (rule pcorresI, rule contrapos-pp)
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

197

from wb have sublinear (wp b) by(auto)
morer from hb have feasible (wp b) by(auto)
ultimately have sub: sub-add (wp b) by(rule sublinear-subadd)

fix Q::a ⇒ real
assume uQ: unitary Q
hence uQϕ: unitary (Q o ϕ) by(auto)
assume ne: «G» & & (wp a Q o ϕ) ≠ «G» & & wp b (Q o ϕ)
hence ne': wp a Q o ϕ ≠ wp b (Q o ϕ) by(auto)

From refinement, « G » & & (wp a Q o ϕ) lies below « G » & & wp b (Q o ϕ).

from ha uQ
have gle: «G» & & (wp a Q o ϕ) ⊨ wp b (Q o ϕ) by(blast intro:drefinesD[OF dr])

have lc: «G» & & (wp a Q o ϕ) ⊨ «G» & & wp b (Q o ϕ)
unfolding exp-conj-def
proof(rule le-funI)

fix s
from gle have «G» s . & (wp a Q o ϕ) s ≤ wp b (Q o ϕ) s
unfolding exp-conj-def by(auto)
hence «G» s . & («G» s . & (wp a Q o ϕ) s) ≤ «G» s . & wp b (Q o ϕ) s
by(auto intro:pconj-mono)
moreover from uQ ha have wp a Q (ϕ s) ≤ 1
by(auto dest:healthy-bounded-byD)
moreover from uQ ha have θ ≤ wp a Q (ϕ s)
by(auto)
ultimately
show « G » s . & (wp a Q o ϕ) s ≤ « G » s . & wp b (Q o ϕ) s
by(simp add:pconj-assoc)

qed

If the programs do not correspond, the terms must differ somewhere, and given the previous result, the second must be somewhere strictly larger than the first:

have rle: ∃ s. («G» & & (wp a Q o ϕ)) s < («G» & & wp b (Q o ϕ)) s
proof(rule contrapos-np[OF ne], rule ext, rule antisym)

fix s
from le show («G» & & (wp a Q o ϕ)) s ≤ («G» & & wp b (Q o ϕ)) s
by(blast)

next
fix s
assume ¬ (∃ s. («G» & & (wp a Q o ϕ)) s < («G» & & wp b (Q o ϕ)) s)
thus («G» & & (wp b (Q o ϕ))) s ≤ («G» & & (wp a Q o ϕ)) s
by(simp add:not-less)

qed

from this obtain s where less-s:
(«G» & & (wp a Q o ϕ)) s < («G» & & wp b (Q o ϕ)) s
by(blast)

The transformers themselves must differ at this point:
hence \( \text{larger: } \text{wp a } Q (\varphi \ s) < \text{wp b } (Q \circ \varphi) \ s \)

**proof** *(cases \(G \ s\))*

- **case** \(True\)
  - moreover from \(ha uQ\) **have** \(0 \leq \text{wp a } Q (\varphi \ s)\)
    - by(blast)
  - moreover from \(hb uQ\ varphi\) **have** \(0 \leq \text{wp b } (Q \circ \varphi) \ s\)
    - by(blast)
  - moreover note the-s
  - ultimately show \(?\text{thesis by}(simp add:exp-conj-def)\)

**next**

- **case** \(False\)
  - moreover from \(ha uQ\) **have** \(wp a \ Q (\varphi \ s) \leq 1\)
    - by(blast)
  - moreover {
    - from \(uQ\) **have** \(\text{bounded-by 1 } (Q \circ \varphi)\)
      - by(blast)
    - moreover from \(\text{unitary-sound}[OF uQ]\)
      - have sound \((Q \circ \varphi)\) by(auto)
      - ultimately have \(wp b (Q \circ \varphi) \ s \leq 1\)
        - using \(hb\) by(auto)
    }
  - moreover note the-s
  - ultimately show \(?\text{thesis by}(simp add:exp-conj-def)\)

**qed**

- from less-s have \((G \ s \land (wp a \ Q \circ \varphi)) \ s \neq (G \ s \land wp b (Q \circ \varphi)) \ s\)
  - by(force)

\(G\) must also hold, as otherwise both would be zero.

- **hence** \(G-s: \ G \ s\)

**proof** *(rule contrapos-np)*

- assume \(nG: \neg G \ s\)
  - moreover from \(ha uQ\) **have** \(wp a \ Q (\varphi \ s) \leq 1\)
    - by(blast)
  - moreover {
    - from \(uQ\) **have** \(\text{bounded-by 1 } (Q \circ \varphi)\)
      - by(blast)
    - moreover from \(\text{unitary-sound}[OF uQ]\)
      - have sound \((Q \circ \varphi)\) by(auto)
      - ultimately have \(wp b (Q \circ \varphi) \ s \leq 1\)
        - using \(hb\) by(auto)
    }
  - ultimately show \((G \ s \land (wp a \ Q \circ \varphi)) \ s = (G \ s \land wp b (Q \circ \varphi)) \ s\)
    - by(simp add:exp-conj-def)

**qed**

Take a carefully constructed expectation:

- let \(?Qc = \lambda s. \text{bound-of } Q \ s\)
  - have \(bQc: \text{bounded-by 1 } ?Qc\)
proof (rule bounded-byI)
  fix s
  from uQ have bound-of Q ≤ 1 and 0 ≤ Q s by (auto)
  thus bound-of Q − Q s ≤ 1 by (auto)
  qed
have sQc: sound ?Qc
proof (rule soundI)
  from bQc show bounded ?Qc by (auto)
  show nneg ?Qc
    proof (rule nnegI)
      fix s from uQ have Q s ≤ bound-of Q by (auto)
      thus 0 ≤ bound-of Q − Q s by (auto)
    qed
  qed

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 (λ s. bound-of Q) (φ s) by (simp add: maximalD determ-maximalD)
also have wp a (λ s. bound-of Q) (φ s) = wp a (λ s. Q s + ?Qc s) (φ s)
  by (simp)
also {
  from da have additive (wp a) by (blast)
  with uQ sQc have wp a (λ s. Q s + ?Qc s) (φ s) =
    wp a Q (φ s) + wp a ?Qc (φ s) by (subst additiveD, blast+)
}
also {
  from ha and sQc and bQc have «G» && (wp a (?Qc o φ) ⊢ wp b (?Qc o φ))
    by (blast intro: drefinesD [OF dr])
  hence («G» && (wp a ?Qc o φ)) s ≤ wp b (?Qc o φ) s
    by (blast)
moreover from sQc and ha have 0 ≤ wp a (λ s. bound-of Q − Q s) (φ s)
  by (blast)
ultimately have wp a ?Qc (φ s) ≤ wp b (?Qc o φ) s
  using G-s by (simp add: exp conj-def)
  hence wp a Q (φ s) + wp a ?Qc (φ s) ≤ wp a Q (φ s) + wp b (?Qc o φ) s
    by (rule add-left-mono)
also with larger
have wp a Q (φ s) + wp b (?Qc o φ) s <
  wp b (Q o φ) s + wp b (?Qc o φ) s
  by (auto)
finally
have \( wp a Q \varphi s + wp a ?Qc \varphi s < \)
\( wp b (Q o \varphi) s + wp b (?Qc o \varphi) s . \)
\}

also from \( sab \) and \( \text{unitary-sound}[OF uQ] \) 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(blast)
also have \( \ldots = wp b (\lambda s. \text{bound-of } Q) s \)
by(simp)
finally
show \( \neg \text{feasible} \) (wp b)
proof(rule contrapos-pn)
assume \( fb: \text{feasible} \) (wp b)
have bounded-by (bound-of Q) (\( \lambda s. \text{bound-of } Q \)) by(blast)

hence bounded-by (bound-of Q) (wp b (\( \lambda s. \text{bound-of } Q \))
using uQ by(blast intro:feasible-boundedD[OF fb])

hence wp b (\( \lambda s. \text{bound-of } Q \)) s \leq bound-of Q by(blast)
thus \( \neg \text{bound-of } Q < wp b (\lambda s. \text{bound-of } Q) s \) by(simp)
qed

qed

4.9.7 Structural Rules for Correspondence

lemma pcorres-Skip:
\( \text{pcorres } \varphi G \text{ Skip Skip} \)
by(simp add:pcorres-def wp-eval)

Correspondence composes over sequential composition.

lemma 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 pcAB: pcorres \( \varphi G A B \)
and pcCD: pcorres \( \varphi H C D \)
and wA: well-def A and wB: well-def B
and wC: well-def C and wD: well-def D
and p3p2: \( \\forall Q. \text{unitary } Q \Rightarrow (I \Rightarrow wp B Q = wp B (H \Rightarrow \&\& Q) \)
and p1p3: \( \forall s. G s \Rightarrow I s \)

shows pcorres \( \varphi G (A;C) (B;D) \)
proof(rule pcorresI)
fix \( Q::'b \Rightarrow \text{ real} \)
assume uQ: unitary Q
with well-def-up-healthy[OF wC] have uCQ: unitary (wp C Q) by(auto)
from uQ well-def-up-healthy[OF wD] have uDQ: unitary (wp D (Q o \varphi))
by(auto dest:unitary-comp)

have p3p1: \( \forall R S. [ \text{ unitary } R; \text{ unitary } S; I \Rightarrow R = I \Rightarrow \&\& S ] \Rightarrow G \Rightarrow \&\& R = G \Rightarrow \&\& S \)
proof(rule ext)
**4.9. THE ALGEBRA OF PGCL**

```
fix R::c ⇒ real and S::c ⇒ real and s::c
assume a3: «I» && R = «I» && S
and uR: unitary R and uS: unitary S
show (*G* && R) s = (*G* && S) s
proof (simp add: exp-conj-def, cases G s)
  case False
    moreover from uR have R s ≤ 1 by (blast)
    moreover from uS have S s ≤ 1 by (blast)
    ultimately show «G» s .& R s = «G» s .& S s
    by (simp)
next
  case True
    note this
    moreover from uR have R s ≤ 1 by (blast)
    moreover from uS have S s ≤ 1 by (blast)
    ultimately show «G» s .& R s = «G» s .& S s
    by (simp)
qed
```
\text{fix } P::'b \Rightarrow \text{real and } Q::'b \Rightarrow \text{real and } s::'c \\
\text{assume } P \vdash \text{wp Skip } Q \\
\text{hence } (P \circ \varphi) s \leq \text{wp Skip } Q (\varphi s) \text{ by(simp, blast)} \\
\text{thus } (P \circ \varphi) s \leq \text{wp Skip } (Q \circ \varphi) s \text{ by(simp add:wp-eval)} \\
\text{qed}

\text{lemma } \text{dr-Abort:} \\
\text{fixes } \varphi::'c \Rightarrow 'b \\
\text{shows } \text{drefines } \varphi \text{ G Abort Abort} \\
\text{proof} \text{ (intro drefinesI2 wd-intros)} \\
\text{fix } P::'b \Rightarrow \text{real and } Q::'b \Rightarrow \text{real and } s::'c \\
\text{assume } P \vdash \text{wp Abort } Q \\
\text{hence } (P \circ \varphi) s \leq \text{wp Abort } Q (\varphi s) \text{ by(auto)} \\
\text{thus } (P \circ \varphi) s \leq \text{wp Abort } (Q \circ \varphi) s \text{ by(simp add:wp-eval)} \\
\text{qed}

\text{lemma } \text{dr-Apply:} \\
\text{fixes } \varphi::'c \Rightarrow 'b \\
\text{assumes } \text{commutes: } f \circ \varphi = \varphi \circ g \\
\text{shows } \text{drefines } \varphi \text{ G (Apply f) (Apply g)} \\
\text{proof} \text{ (intro drefinesI2 wd-intros)} \\
\text{fix } P::'b \Rightarrow \text{real and } Q::'b \Rightarrow \text{real and } s::'c \\
\text{assume wp: } P \vdash \text{wp (Apply f) } Q \\
\text{hence } P \vdash (Q \circ f) \text{ by(simp add:wp-eval)} \\
\text{hence } P (\varphi s) \leq (Q \circ f) (\varphi s) \text{ by(blast)} \\
\text{also have } ... = Q ((f \circ \varphi) s) \text{ by(simp)} \\
\text{also with commutes} \\
\text{have } ... = (((Q \circ \varphi) o g) s) \text{ by(simp)} \\
\text{also have } ... = \text{wp (Apply g) } (Q \circ \varphi) s \\
\text{by(simp add:wp-eval)} \\
\text{finally show } (P \circ \varphi) s \leq \text{wp (Apply g) } (Q \circ \varphi) s \text{ by(simp)} \\
\text{qed}

\text{lemma } \text{dr-Seq:} \\
\text{assumes } \text{drAB: } \text{drefines } \varphi \text{ P A B} \\
\text{and } \text{drBC: } \text{drefines } \varphi \text{ Q C D} \\
\text{and } \text{wpB: } \text{ «P» } \vdash \text{ wp B «Q»} \\
\text{and } \text{wB: } \text{well-def B} \\
\text{and } \text{wC: } \text{well-def C} \\
\text{and } \text{wD: } \text{well-def D} \\
\text{shows } \text{drefines } \varphi \text{ P (A;;C) (B;;D)} \\
\text{proof} \\
\text{fix } R \text{ and } S \\
\text{assume uR: unitary R and uS: unitary S} \\
\text{and } \text{wpAC: } R \vdash \text{ wp (A;;C) S} \\
\text{from uR} \\
\text{have } \text{ «P» \&\& (R o \varphi) = «P» \&\& («P» \&\& (R o \varphi))}
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*:

$$ (\lambda s. 0) \vdash wp \text{ Abort } Q $$

*unfolding* *wp-eval* *by* *(simp)*

**lemma** *wlp-Abort*:

$$ (\lambda s. 1) \vdash wlp \text{ Abort } Q $$

*unfolding* *wp-eval* *by* *(simp)*

**lemma** *wp-Skip*:

$$ P \vdash wp \text{ Skip } P $$

*unfolding* *wp-eval* *by* *(blast)*

**lemma** *wlp-Skip*:

$$ P \vdash wlp \text{ Skip } P $$

*unfolding* *wp-eval* *by* *(blast)*

**lemma** *wp-Apply*:

$$ Q \circ f \vdash wp \left( \text{Apply } f \right) Q $$

*unfolding* *wp-eval* *by* *(simp)*

**lemma** *wlp-Apply*:

$$ Q \circ f \vdash wlp \left( \text{Apply } f \right) Q $$

*unfolding* *wp-eval* *by* *(simp)*

**lemma** *wp-Seq*:

*assumes* *ent-a*:

$$ P \vdash wp \ a \ Q $$

*and* *ent-b*:

$$ Q \vdash wp \ b \ R $$

*and* *wa*:

*well-def a*

*and* *wb*:

*well-def b*

*and* *s-Q*:

*sound Q*

*and* *s-R*:

*sound R*

*shows* $$ P \vdash wp \left( a \ ;; \ b \right) R $$

*proof* —

*note* *ha* = *well-def-wp-healthy*[*OF wa*]

*note* *hb* = *well-def-wp-healthy*[*OF wb*]

*note* *ent-a*

*also from* *ent-b* *ha* *hb* *s-Q* *s-R* *have* $$ wp \ a \ Q \vdash wp \ a \ (wp \ b \ R) $$

*by* *(blast intro:healthy-monoD2)*

*finally show* ?*thesis* *by* *(simp add:* wp-eval*)

qed

**lemma** *wlp-Seq*:
assumes \text{ent-a} \colon P \vdash \text{wlp} a Q \\
\text{and ent-b} \colon Q \vdash \text{wlp} b R \\
\text{and wa} \colon \text{well-def} a \\
\text{and wb} \colon \text{well-def} b \\
\text{and u-Q} \colon \text{unitary} Q \\
\text{and u-R} \colon \text{unitary} R \\
shows P \vdash \text{wlp} (a ;; b) R \\

proof \\
\text{note ha} = \text{well-def-wlp-nearly-healthy}[\text{OF wa}] \\
\text{note hb} = \text{well-def-wlp-nearly-healthy}[\text{OF wb}] \\
\text{note ent-a also from ent-b ha hb u-Q u-R have wlp a Q} \\
\text{by (blast intro: nearly-healthy-monoD[OF ha])} \\
\text{finally show ?thesis by (simp add: wp-eval)}\\n\text{qed}\\n
\text{lemma wp-PC}: \\
(\lambda s. P s \ast \text{wp} a Q s + (1 - P s) \ast \text{wp} b Q s) \vdash \text{wp} (a \oplus b) Q \\
\text{by (simp add: wp-eval)}\\n
\text{lemma wlp-PC}: \\
(\lambda s. P s \ast \text{wlp} a Q s + (1 - P s) \ast \text{wlp} b Q s) \vdash \text{wlp} (a \oplus b) Q \\
\text{by (simp add: wp-eval)}\\n
A simpler rule for when the probability does not depend on the state.\\n
\text{lemma PC-fixed}: \\
\text{assumes wpa} \colon P \vdash a ab R \\
\text{and wpb} \colon Q \vdash b ab R \\
\text{and np: 0} \leq p \text{ and bp: p} \leq 1 \\
\text{shows (\lambda s. p \ast P s + (1 - p) \ast Q s) \vdash (a (\lambda s. p) \oplus b) ab R} \\
\text{unfolding PC-def} \\
\text{proof (rule le-funI)} \\
\text{fix s} \\
\text{from wpa and np have p \ast P s} \leq p \ast a ab R s \\
\text{by (auto intro: mult-left-monotone)} \\
\text{moreover \{} \\
\text{from bp have 0 \leq 1 - p by (simp)} \\
\text{with wpb have (1 - p) \ast Q s} \leq (1 - p) \ast b ab R s \\
\text{by (auto intro: mult-left-monotone)} \\
\text{\}} \\
\text{ultimately show p \ast P s + (1 - p) \ast Q s} \leq \\
p \ast a ab R s + (1 - p) \ast b ab R s \\
\text{by (rule add-monotone)}\\n\text{qed}\\n
\text{lemma wp-PC-fixed}: \\
[ P \vdash \text{wp} a R; Q \vdash \text{wp} b R; 0 \leq p; p \leq 1 ] \implies \\
(\lambda s. p \ast P s + (1 - p) \ast Q s) \vdash \text{wp} (a (\lambda s. p) \oplus b) R \\
\text{by (simp add: wp-def PC-fixed)}
lemma wlp-PC-fixed:
\[ [ P \vdash \text{wlp a R}; Q \vdash \text{wlp b R}; 0 \leq p; p \leq 1 ] \implies (\lambda s. p \ast P s + (1 - p) \ast Q s) \vdash \text{wlp} (a (\lambda s. p) \oplus b) R \]
by(simp add:wlp-def PC-fixed)

lemma wp-DC:
(\lambda s. min (wp a Q s) (wp b Q s)) \vdash wp (a \sqcap b) Q
unfolding wp-eval by(simp)

lemma wlp-DC:
(\lambda s. min (wlp a Q s) (wlp b Q s)) \vdash wlp (a \sqcap b) Q
unfolding wp-eval by(simp)

Combining annotations for both branches:

lemma DC-split:
fixes a::'s prog and b
assumes wpa: P \vdash a ab R
and wpb: Q \vdash b ab R
shows (\lambda s. min (P s) (Q s)) \vdash (a \sqcap b) ab R
unfolding DC-def
proof(rule le-funI)
fix s
from wpa wpb
have P s \leq a ab R s and Q s \leq b ab R s by(auto)
thus min (P s) (Q s) \leq min (a ab R s) (b ab R s) by(auto)
qed

lemma wp-DC-split:
[ [ P \vdash \text{wp prog R}; Q \vdash \text{wp prog' R} ] \implies
(\lambda s. min (P s) (Q s)) \vdash \text{wp} (\text{prog} \sqcap \text{prog'}) R
by(simp add:wp-def DC-split)

lemma wlp-DC-split:
[ [ P \vdash \text{wlp prog R}; Q \vdash \text{wlp prog' R} ] \implies
(\lambda s. min (P s) (Q s)) \vdash \text{wlp} (\text{prog} \sqcap \text{prog'}) R
by(simp add:wlp-def DC-split)

lemma wp-DC-split-same:
[ [ P \vdash \text{wp prog Q}; P \vdash \text{wp prog' Q} ] \implies P \vdash \text{wp} (\text{prog} \sqcap \text{prog'}) Q
unfolding wp-eval by(blast intro:min.boundedI)

lemma wlp-DC-split-same:
[ [ P \vdash \text{wlp prog Q}; P \vdash \text{wlp prog' Q} ] \implies P \vdash \text{wlp} (\text{prog} \sqcap \text{prog'}) Q
unfolding wp-eval by(blast intro:min.boundedI)

lemma SetPC-split:
fixes f::'x \Rightarrow 'y prog
and p::'y \Rightarrow 'x \Rightarrow real
4.10. STRUCTURED REASONING

assumes \( \forall x. s. x \in \text{supp} (p s) \implies P x \vdash f x ab Q \)
and \( \text{nnp: } \forall s. \text{nneg} (p s) \)
shows \( (\forall s. \sum x \in \text{supp} (p s). p s x \cdot P x s) \vdash \text{SetPC} f p ab Q \)

unfolding \( \text{SetPC-def} \)

proof \( (\text{rule le-funI}) \)

fix \( s \)

from \( \text{rec have } \forall x. x \in \text{supp} (p s) \implies P x s \leq f x ab Q s \text{ by(blast)} \)
moreover from \( \text{nnp have } \forall x. \theta \leq p s x \text{ by(blast)} \)
ultimately have \( (\forall s. \sum x \in \text{supp} (p s). p s x \cdot P x s) \leq (\sum x \in \text{supp} (p s). p s x \cdot f x ab Q s) \)
by \( (\text{rule sum-mono}) \)

qed

lemma \( \text{wp-SetPC-split:} \)
\[ (\forall x. x \in \text{supp} (p s) \implies P x \vdash wp (f x) Q; \forall s. \text{nneg} (p s) ) \implies (\forall s. \sum x \in \text{supp} (p s). p s x \cdot P x s) \vdash wp (\text{SetPC} f p) Q \]
by \( (\text{simp add:wp-def SetPC-split}) \)

lemma \( \text{wlp-SetPC-split:} \)
\[ (\forall x. x \in \text{supp} (p s) \implies P x \vdash wlp (f x) Q; \forall s. \text{nneg} (p s) ) \implies (\forall s. \sum x \in \text{supp} (p s). p s x \cdot P x s) \vdash wlp (\text{SetPC} f p) Q \]
by \( (\text{simp add:wp-def SetPC-split}) \)

lemma \( \text{wp-SetDC-split:} \)
\[ (\forall s. x. x \in S s \implies P \vdash wp (f x) Q; \forall s. S s \neq \{\} ) \implies P \vdash wp (\text{SetDC} f s) Q \]
by \( (\text{rule le-funI}, \text{unfold wp-eval, blast intro!:cInf-greatest}) \)

lemma \( \text{wlp-SetDC-split:} \)
\[ (\forall s. x. x \in S s \implies P \vdash wlp (f x) Q; \forall s. S s \neq \{\} ) \implies P \vdash wlp (\text{SetDC} f s) Q \]
by \( (\text{rule le-funI}, \text{unfold wp-eval, blast intro!:cInf-greatest}) \)

lemma \( \text{wp-SetDC:} \)
assumes \( \text{wp: } \forall s. x. x \in S s \implies P x \vdash wp (f x) Q \)
and \( \text{nc: } \forall s. \text{S s \neq \{}\} \)
and \( \text{sP: } \forall x. \text{sound} (P x) \)
shows \( (\forall s. \text{Inf} ((\forall x. P x s \cdot S s)) \vdash wp (\text{SetDC} f s) Q \)
using \( \text{assms by(intro le-funI, simp add:wp-eval, blast intro!:cInf-mono}) \)

lemma \( \text{wlp-SetDC:} \)
assumes \( \text{wp: } \forall s. x. x \in S s \implies P x \vdash wlp (f x) Q \)
and \( \text{nc: } \forall s. \text{S s \neq \{}\} \)
and \( \text{sP: } \forall x. \text{sound} (P x) \)
shows \( (\forall s. \text{Inf} ((\forall x. P x s \cdot S s)) \vdash wlp (\text{SetDC} f s) Q \)
using \( \text{assms by(intro le-funI, simp add:wp-eval, blast intro!:cInf-mono}) \)

lemma \( \text{wp-Embed:} \)
\[ P \vdash t \ Q \implies P \vdash \text{wp} (\text{Embed} \ t) \ Q \]
by (simp add: wp-def Embed-def)

lemma wlp-Embed:
\[ P \vdash t \ Q \implies P \vdash \text{wlp} (\text{Embed} \ t) \ Q \]
by (simp add: wlp-def Embed-def)

lemma wp-Bind:
\[
\[ \forall s. \ P \ s \leq \text{wp} (a \ (f \ s)) \ Q \ s \] \implies P \vdash \text{wp} (\text{Bind} \ f \ a) \ Q
\]
by (auto simp: wp-def Bind-def)

lemma wlp-Bind:
\[
\[ \forall s. \ P \ s \leq \text{wlp} (a \ (f \ s)) \ Q \ s \] \implies P \vdash \text{wlp} (\text{Bind} \ f \ a) \ Q
\]
by (auto simp: wlp-def Bind-def)

lemma wp-repeat:
\[
[ P \vdash \text{wp} \ a \ Q; \ Q \vdash \text{wp} (\text{repeat} \ n \ a) \ R; \hspace{1cm} \text{well-def} \ a; \ \text{sound} \ Q; \ \text{sound} \ R \] \implies P \vdash \text{wp} (\text{repeat} \ (\text{Suc} \ n) \ a) \ R
\]
by (auto intro: wp-Seq wd-intros)

lemma wlp-repeat:
\[
[ P \vdash \text{wlp} \ a \ Q; \ Q \vdash \text{wlp} (\text{repeat} \ n \ a) \ R; \hspace{1cm} \text{well-def} \ a; \ \text{unitary} \ Q; \ \text{unitary} \ R \] \implies P \vdash \text{wlp} (\text{repeat} \ (\text{Suc} \ n) \ a) \ R
\]
by (auto intro: wlp-Seq wd-intros)

Note that the loop rules presented in section Section 4.8 are of the same form, and would belong here, had they not already been stated.

The following rules are specialisations of those for general transformers, and are easier for the unifier to match.

lemmas wp-strengthen-post=
entails-strengthen-post[where t=wp a for a]

lemma wp-strengthen-post:
\[ P \vdash \text{wp} \ a \ Q \implies \text{nearly-healthy} (\text{wp} \ a) \implies \text{unitary} \ R \implies Q \vdash R \implies \text{unitary} \]
Q \implies P \vdash \text{wp} \ a \ R
by (blast intro: entails-trans)

lemmas wp-weaken-pre=
entails-weaken-pre[where t=wp a for a]

lemmas wlp-weaken-pre=
entails-weaken-pre[where t=wlp a for a]

lemmas wp-scale=
entails-scale[where t=wp a for a, OF - well-def-wp-healthy]
4.10.2 Algebraic Decomposition

Refinement is a powerful tool for decomposition, belied by the simplicity of the rule. This is an axiomatic formulation of refinement (all annotations of the $a$ are annotations of $b$), rather than an operational version (all traces of $b$ are traces of $a$).

**Lemma** $\text{wp-refines}:$

\[
\left[ a \sqsubseteq b; P \vdash wp a Q \right] \implies P \vdash wp b Q
\]

by (auto intro: entails-trans)

**Lemmas** $\text{wp-drefines} = \text{drefinesD}$

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** $\text{wp-valid} :: (\Tri{\mathcal{a} \Rightarrow \mathcal{real}}) \Rightarrow (\Tri{\mathcal{a} \Rightarrow \mathcal{real}}) \Rightarrow \mathcal{bool}$

\[
\{\mathcal{P}\} \mathcal{a} \{\mathcal{Q}\} \mathcal{p} \Rightarrow \{\mathcal{P}\} \mathcal{b} \{\mathcal{R}\} \mathcal{p}
\]

where

\[
\text{wp-valid} P \text{ prog } Q \equiv P \vdash \vdash \text{wp prog } Q
\]

**Lemma** $\text{wp-validI}:

\[
P \vdash \text{wp prog } Q \implies \{P\} \text{ prog } \{Q\} \mathcal{p}
\]

**Unfolding** $\text{wp-valid-def}$ by (assumption)

**Lemma** $\text{wp-validD}:

\[
\{P\} \text{ prog } \{Q\} \mathcal{p} \implies P \vdash \text{wp prog } Q
\]

**Unfolding** $\text{wp-valid-def}$ by (assumption)

**Lemma** $\text{valid-Seq}:

\[
\left[ \{P\} \mathcal{a} \{Q\} \mathcal{p}; \{Q\} \mathcal{b} \{R\} \mathcal{p}; \text{well-def } a; \text{well-def } b; \text{sound } Q; \text{sound } R \right] \implies \\
\{P\} \mathcal{a} \mathcal{b} \{R\} \mathcal{p}
\]

**Unfolding** $\text{wp-valid-def}$ by (rule wp-Seq)

We make it available to the computational reasoner:

**Declare** $\text{valid-Seq[trans]}$

end

4.11 Loop Termination

**Theory** $\text{Termination imports Embedding StructuredReasoning Loops begin}$

Termination for loops can be shown by classical means (using a variant, or a measure function), or by probabilistic means: We only need that the loop terminates with probability one.
4.11.1 Trivial Termination

A maximal transformer (program) doesn’t affect termination. This is essentially saying that such a program doesn’t abort (or diverge).

**lemma maximal-Seq-term:**
fixes $r$::'s prog and $s$::'s prog
assumes $mr$: maximal ($wp$ $r$) and $ws$: well-def $s$
and $ts$: ($\lambda$s. 1) $\vdash$ $wp$ ($r$ ;; $s$) ($\lambda$s. 1)
shows ($\lambda$s. 1) $\vdash$ $wp$ ($r$ ;; $s$) ($\lambda$s. 1)

**proof**

note $hs$ = well-def-wp-healthy[OF $ws$]
have $wp$ $s$ ($\lambda$s. 1) = ($\lambda$s. 1)
proof(rule antisym)
  show ($\lambda$s. 1) $\vdash$ $wp$ $s$ ($\lambda$s. 1) by(rule $ts$)
  have bounded-by 1 ($wp$ $s$ ($\lambda$s. 1))
    by(auto intro!:healthy-bounded-byD[OF $hs$])
  thus $wp$ $s$ ($\lambda$s. 1) $\vdash$ ($\lambda$s. 1) by(auto intro!:le-funI)
qed

with $mr$ show ?thesis
  by(simp add:$wp$-eval embed-bool-def maximalD)
qed

From any state where the guard does not hold, a loop terminates in a single step.

**lemma term-onestep:**
assumes $wb$: well-def body
shows $\langle N \ G \rangle$ $\vdash$ $wp$ do $G$ $\longrightarrow$ body od ($\lambda$s. 1)
proof(rule le-funI)
note $hb$ = well-def-wp-healthy[OF $wb$]
fix $s$
show $\langle N \ G \rangle$ $s$ $\leq$ $wp$ do $G$ $\longrightarrow$ body od ($\lambda$s. 1) $s$
proof(cases $G$ $s$, simp-all add:$wp$-loop-nguard $hb$)
  from $hb$ have sound ($wp$ do $G$ $\longrightarrow$ body od ($\lambda$s. 1))
    by(auto intro!:healthy-sound[OF healthy-wp-loop])
  thus $0$ $\leq$ $wp$ do $G$ $\longrightarrow$ body od ($\lambda$s. 1) $s$ by(auto)
qed

4.11.2 Classical Termination

The first non-trivial termination result is quite standard: If we can provide a natural-number-valued measure, that decreases on every iteration, and implies termination on reaching zero, the loop terminates.

**lemma loop-term-nat-measure-noinv:**
fixes $m$ :: 's $\Rightarrow$ nat and $body$ :: 's prog
assumes $wb$: well-def body
and $guard$: $\forall s. m\ s = 0$ $\longrightarrow$ $\neg G\ s$
and \textbf{variant:} \(\forall n. \langle \lambda s. m s = \text{Suc } n \rangle \vdash \wp \text{ body } \langle \lambda s. m s = n \rangle\)

shows \(\lambda s. 1 \vdash \wp \text{ do G } ightarrow \text{ body od } (\lambda s. 1)\)

\textbf{proof –}

\textbf{note} \(hb = \text{well-def-wp-healthy}[\text{OF wb}]\)

have \(\forall n. (\forall s. m s = n \rightarrow I \leq \wp \text{ do G } ightarrow \text{ body od } (\lambda s. 1) s)\)

\textbf{proof}(\text{induct-tac } n)

\begin{align*}
\text{fix } n \\
\text{show } \forall s. m s = 0 \rightarrow I \leq \wp \text{ do G } ightarrow \text{ body od } (\lambda s. 1) s \\
\text{proof}(\text{clarify}) \\
\text{fix } s \\
\text{assume } m s = 0 \\
\text{with guard have } \neg G s \text{ by (blast)} \\
\text{with } hb \text{ show } I \leq \wp \text{ do G } ightarrow \text{ body od } (\lambda s. 1) s \\
\text{by (simp add:wp-loop-nguard)} \\
\text{qed}
\end{align*}

\begin{align*}
\text{assume } IH; \forall s. m s = n \rightarrow I \leq \wp \text{ do G } ightarrow \text{ body od } (\lambda s. 1) s \\
\text{hence } IH'; \forall s. m s = n \rightarrow I \leq \wp \text{ do G } ightarrow \text{ body od } \langle \lambda s. \text{True} \rangle s \\
\text{by (simp add:embed-bool-def)} \\
\text{have } \forall s. m s = \text{Suc } n \rightarrow I \leq \wp \text{ do G } ightarrow \text{ body od } \langle \lambda s. \text{True} \rangle s \\
\text{proof}(\text{intro fold-premise healthy-intros hb, rule le-funI}) \\
\text{fix } s \\
\text{show } \langle \lambda s. m s = \text{Suc } n \rangle s \leq \wp \text{ do G } ightarrow \text{ body od } \langle \lambda s. \text{True} \rangle s \\
\text{proof}(\text{cases G s}) \\
\text{case False} \\
\text{hence } I = \langle \text{N } G \rangle s \text{ by (auto)} \\
\text{also from wb have } ... \leq \wp \text{ do G } ightarrow \text{ body od } (\lambda s. 1) s \\
\text{by (rule le-funD[\text{OF term-onestep}])} \\
\text{finally show } ?\text{thesis by (simp add:embed-bool-def)} \\
\text{next} \\
\text{case True note } G = \text{this} \\
\text{from } IH' \text{ have } \langle \lambda s. m s = n \rangle \vdash \wp \text{ do G } ightarrow \text{ body od } \langle \lambda s. \text{True} \rangle \\
\text{by (blast intro:use-premise healthy-intros hb)} \\
\text{with variant wb} \\
\text{have } \langle \lambda s. m s = \text{Suc } n \rangle \vdash (\text{body };; \text{ do G } ightarrow \text{ body od}) \langle \lambda s. \text{True} \rangle \\
\text{by (blast intro:wp-Seq wd-intros)} \\
\text{hence } \langle \lambda s. m s = \text{Suc } n \rangle s \leq \wp \text{ (body };; \text{ do G } ightarrow \text{ body od}) \langle \lambda s. \text{True} \rangle s \\
\text{by (auto)} \\
\text{also from hb G have } ... = \wp \text{ do G } ightarrow \text{ body od } \langle \lambda s. \text{True} \rangle s \\
\text{by (simp add:wp-loop-guard)} \\
\text{finally show } ?\text{thesis} . \\
\text{qed}
\end{align*}

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 \( \Rightarrow \) nat and body :: \('s\) prog
- assumes \( \text{wb} :: \) well-def body
- and guard: \( \forall s. m s = 0 \rightarrow \sim G s \)
- and variant: \( \forall n. \lambda s. m s = \text{Suc} n \land \lambda I s \rightarrow 1 \leq \text{wp} G \rightarrow \text{body od} \langle \lambda s. \text{True} \rangle s \)
- and \( \text{inv} : \) wp-inv \( G \) body \( \langle \lambda s. I \rangle \)
- shows \( \langle I \rangle \vdash \text{wp} G \rightarrow \text{body od} (\lambda s. I) \)

**proof**

- **note** \( \text{hb} = \) well-def-wp-healthy[\( \text{OF wb} \)]
- **note** \( \text{scb} = \) sublinear-sub-conj[\( \text{OF well-def-wp-sublinear, OF wb} \)]
- **have** \( \langle I \rangle \vdash \text{wp} G \rightarrow \text{body od} \langle \lambda s. \text{True} \rangle s \)
- **proof** (rule use-premise, intro healthy-intros \( \text{hb} \))
  - fix \( n \)
  - show \( \forall s. m s = n \land I s \rightarrow 1 \leq \text{wp} G \rightarrow \text{body od} \langle \lambda s. \text{True} \rangle s \)
  - **proof** (clarify)
    - fix \( s \)
    - assume \( m s = 0 \)
    - with **guard** have \( \sim G s \) by (blast)
    - with \( \text{hb} \) show \( 1 \leq \text{wp} G \rightarrow \text{body od} \langle \lambda s. \text{True} \rangle s \)
    - by (simp add: wp-loop-nguard)
  - qed
  - assume \( \forall s. m s = n \land I s \rightarrow 1 \leq \text{wp} G \rightarrow \text{body od} \langle \lambda s. \text{True} \rangle s \)
  - show \( \forall s. m s = \text{Suc} n \land I s \rightarrow 1 \leq \text{wp} G \rightarrow \text{body od} \langle \lambda s. \text{True} \rangle s \)
  - **proof** (intro fold-premise healthy-intros \( \text{hb} \) le-funI)
    - fix \( s \)
    - show \( \langle \lambda s. m s = \text{Suc} n \land I s \rangle \leq \text{wp} G \rightarrow \text{body od} \langle \lambda s. \text{True} \rangle s \)
    - **proof** (cases \( G \) \( s \))
      - case **False** with \( \text{hb} \) show \( ?\)thesis
      - by (simp add: wp-loop-nguard)
  - **next**
    - case **True** note \( G = \) this
      - **have** \( \langle \lambda s. m s = \text{Suc} n \land \lambda I s \rangle \leq \text{wp} G \rightarrow \text{body od} \langle \lambda s. \text{True} \rangle s \)
      - **proof**
        - case **False**
          - **note** \( G = \) this
          - **have** \( \langle \lambda s. m s = \text{Suc} n \land \lambda I s \rangle \leq \text{wp} G \rightarrow \text{body od} \langle \lambda s. \text{True} \rangle s \)
          - **proof**
            - from **inv** \( \text{hb} \) have \( \langle G \rangle \land \langle I \rangle \vdash \text{wp body} \langle I \rangle \)
            - by (rule wp-inv-stdD)
            - with **variant**
              - **have** \( \langle \lambda s. m s = \text{Suc} n \rangle \land \langle I \rangle \) \( \vdash \) wp body \( \langle \lambda s. m s = n \rangle \land \text{wp body} \langle I \rangle \)
              - by (rule entails-frame)
4.11. LOOP TERMINATION

4.11.3 Probabilistic Termination

Any loop that has a non-zero chance of terminating after each step terminates with probability 1.

**lemma** termination-0-1:

- **fixes** body :: 's prog
- **assumes** wb: well-def body
  - The loop terminates in one step with nonzero probability
- **and** onestep: (λs. p) ⊢ wp body «N G»
- **and** nzp: 0 < p
  - The body is maximal i.e. it terminates absolutely.
- **and** mb: maximal (wp body)
- **shows** λs. 1 ⊬ wp do G → body od (λs. 1)
proof

note \( hb = \text{well-def-wp-healthy}[\text{OF } wb] \)

note \( sh = \text{healthy-scalingD}[\text{OF } hb] \)

note \( sab = \text{sublinear-subadd}[\text{OF well-def-wp-sublinear}, \text{OF } wb, \text{OF healthy-feasibleD}, \text{OF } hb] \)

from \( hb \) have \( hloop: \text{healthy} (wp \text{ do } G \rightarrow \text{body od}) \)
by (rule healthy-intros)

hence \( swp: \text{sound} (wp \text{ do } G \rightarrow \text{body od } (\lambda s. 1)) \) by (blast)

\( p \) is no greater than 1, by feasibility.

from onestep have onestep': \( \forall s. p \leq wp \text{ body } «N G» s \) by (auto)
also { from \( hb \) have unitary \( wp \text{ body } «N G» \) by (auto) hence \( \forall s. wp \text{ body } «N G» s \leq 1 \) by (auto) }
finally have \( p1: p \leq 1 \).

This is the crux of the proof: that given a lower bound below 1, we can find another, higher one.

have new-bound: \( \forall k. 0 \leq k \implies k \leq 1 \implies (\lambda s. k) \vdash wp \text{ do } G \rightarrow \text{body od } (\lambda s. 1) \)
proof (rule le-funI)
fix \( k \) \( s \)
assume \( X: \lambda s. k \vdash wp \text{ do } G \rightarrow \text{body od } (\lambda s. 1) \)
and \( k0: 0 \leq k \) and \( k1: k \leq 1 \)
from \( k1 \) have nz1k: \( 0 \leq 1 - k \) by (auto)
with \( p1 \) have \( p * (1 - k) + k \leq 1 * (1 - k) + k \)
by (blast intro: mult-right-mono add-mono)
hence \( p * (1 - k) + k \leq 1 \)
by (simp)

The new bound is \( p * (1 - k) + k \).

hence \( p * (1 - k) + k \leq «N G» s + «G» s * (p * (1 - k) + k) \)
by (cases G s, simp-all)

By the one-step termination assumption:
also from onestep' nz1k
have ... \( \leq «N G» s + «G» s * (wp \text{ body } «N G» s * (1 - k) + k) \)
by (simp add: mult-right-mono ordered-comm-semiring-class.comm-mult-left-mono)

By scaling:
also from nz1k
have ... = \( «N G» s + «G» s * (wp \text{ body } (\lambda s. «N G» s s * (1 - k)) s + k) \)
by (simp add: right-scalingD[OF sb])

By the maximality (termination) of the loop body:
also from mb k0
have ... = «N G» s + «G» s * (wp body (λs. «N G» s *(1 – k)) s + wp body (λs. k) s)
  by(simp add:maximalD)

By sub-additivity of the loop body:
also from k0 nz1k
have ... ≤ «N G» s + «G» s * (wp body (λs. «N G» s *(1 – k) + k) s)
  by(auto intro!:add-left-mono multi-left-mono sub-addD[OF sub] sound-intros)
also
have ... = «N G» s + «G» s * (wp body (λs. «N G» s + «G» s * k) s)
  by(simp add:negate-embed algebra-simps)

By monotonicity of the loop body, and that k is a lower bound:
also from k0 bloop le-funD[OF X]
have ... ≤ «N G» s + «G» s * (wp body (λs. «N G» s + «G» s * wp do G → body od (λs. 1)) s)
  by(auto intro!:add-left-mono multi-left-mono add-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 \( \lambda s. «N G» s + «G» s * \) (wp body (wp do G → body od (λs. 1)) s) =
    «N G» s + «G» s *(wp body (λs. «N G» s + «G» s * wp body (wp do G → body od (λs. 1)) s)
  \( \lambda s. \) (wp body od (λs. 1)) s)
  by(simp only:distrib-left mult.assoc[symmetric] embed-bool-idem embed-bool-cancel)
also have \( \lambda s. \) (wp do G → body od (λs. 1)) s
  by(simp add:fun-cong[OF wp-loop-unfold[ symmetric, where P=λs. 1, simplified, OF hb]]
                   simp only:X)

finally have X: \( \lambda s. «N G» s + «G» s * \) (wp body (wp do G → body od (λs. 1)) s) =
    «N G» s + «G» s * (wp body (\( \lambda s. «N G» s + «G» s * \) wp do G → body od (λs. 1)) s) s
  by(simp only:X)

Lastly, by folding two loop iterations:
also
have <N G> s + <G> s * (wp body (\( \lambda s. <N G» s + «G» s * \) wp body (wp do G → body od (λs. 1)) s) s) =
    wp do G → body od (λs. 1) s
  by(simp add:wp-loop-unfold[OF - hb, where P=λs. 1, simplified, symmetric]
                 simp only: X)

...
finally show \( p \ast (1-k) + k \leq wp \rightarrow body od (\lambda s. 1) s \).
qed

If the previous bound lay in \([0, 1)\), the new bound is strictly greater. This is where we appeal to the fact that \( p \) is nonzero.

from nzp have inc: \( \forall k. 0 \leq k \Rightarrow k < 1 \Rightarrow k < p \ast (1 - k) + k \)
  by(auto intro:mult-pos-pos)

The result follows by contradiction.

show ?thesis
proof (rule ccontr)

If the loop does not terminate everywhere, then there must exist some state from which the probability of termination is strictly less than one.

assume \( \neg \ ?thesis \)

hence \( \neg (\forall s. 1 \leq wp \rightarrow body od (\lambda s. 1) s) \) by(auto)

then obtain \( s \) where point: \( \neg 1 \leq wp \rightarrow body od (\lambda s. 1) s \) by(auto)

let \( ?k = \text{Inf} (\text{range} (wp \rightarrow body od (\lambda s. 1))) \)

from hloop have Inflb: \( \forall s. ?k \leq wp \rightarrow body od (\lambda s. 1) s \)
  by(intro cInf-lower bdd-belowI, auto)

also from point have wp do G \( \rightarrow body od (\lambda s. 1) s < 1 \) by(auto)

Thus the least (infimum) probability of termination is strictly less than one.

finally have k1: \( ?k < 1 \).

hence \( ?k \leq 1 \) by(auto)

moreover from hloop have k0: \( \theta \leq ?k \)
  by(intro cInf-greatest, auto)

The infimum is, naturally, a lower bound.

moreover from Inflb have \((\lambda s. ?k) \vdash wp \rightarrow body od (\lambda s. 1)) \) by(auto)

ultimately

We can therefore use the previous result to find a new bound, …

have \( \forall s. p \ast (1 - ?k) + ?k \leq wp \rightarrow body od (\lambda s. 1) s \)
  by(blast intro:le-funD[OF new-bound])

… which is lower than the infimum, by minimality, …

hence \( p \ast (1 - ?k) + ?k \leq ?k \)
  by(blast intro:cInf-greatest)

… yet also strictly greater than it.

moreover from k0 k1 have ?k < p \ast (1 - ?k) + ?k by(rule inc)

We thus have a contradiction.

ultimately show False by(simp)
4.12 Automated Reasoning

theory Automation imports StructuredReasoning begin

This theory serves as a container for automated reasoning tactics for pGCL, implemented in ML. At present, there is a basic verification condition generator (VCG).

named-theorems wd
theorems to automatically establish well-definedness

named-theorems pwp-core
core probabilistic wp rules, for evaluating primitive terms

named-theorems pwp
user-supplied probabilistic wp rules

named-theorems pwlp
user-supplied probabilistic wlp rules

ML-file ⟨pVCG.ML⟩

method-setup pvecg =
 ⟨Scan.succeed (fn ctxt => SIMPLE-METHOD' (pVCG.pVCG-tac ctxt))⟩
Probabilistic weakest preexpectation tactic

declare wd-intros[wd]

lemmas core-wp-rules =
 wp-Skip wp-Skip
 wp-Abort wp-Abort
 wp-Apply wp-Apply
 wp-Seq wp-Seq
 wp-DC-split wp-DC-split
 wp-PC-fixed wp-PC-fixed
 wp-SetDC wp-SetDC
 wp-SetPC-split wp-SetPC-split

declare core-wp-rules[pwp-core]

end
4.13 Miscellaneous Mathematics

theory Misc
imports "HOL-Analysis.Analysis"
begin

lemma sum-UNIV:
  fixes S :: 'a::finite set
  assumes complete: \( \forall x. x \notin S \rightarrow f x = 0 \)
  shows \( \sum f S = \sum f \text{UNIV} \)

proof
  from complete have \( \sum f S = \sum f (\text{UNIV} - S) + \sum f S \) by (simp)
  also have \( \ldots = \sum f \text{UNIV} \)
    by (auto intro: sum.subset-diff[symmetric])
  finally show \(?thesis\).
qed

lemma cInf-mono:
  fixes A :: 'a::conditionally-complete-lattice set
  assumes lower: \( \forall b. b \in B \rightarrow \exists a \in A. a \leq b \)
    and bounded: \( \forall a. a \in A \rightarrow c \leq a \)
    and ne: \( B \neq \{\} \)
  shows \( \inf A \leq \inf B \)

proof (rule cInf-greatest[OF ne])
  fix b assume bin: \( b \in B \)
  with lower obtain a where ain: \( a \in A \) and le: \( a \leq b \) by (auto)
  from ain bounded have \( \inf A \leq a \) by (intro cInf-lower bdd-belowI, auto)
  also note le
  finally show \( \inf A \leq b \).
qed

lemma max-distrib:
  fixes c :: real
  assumes nn: \( 0 \leq c \)
  shows \( c \cdot \max a b = \max (c \cdot a) (c \cdot b) \)

proof (cases a \leq b)
  case True
  moreover with nn have \( c \cdot a \leq c \cdot b \) by (auto intro: mult-left-mono)
  ultimately show \(?thesis\) by (simp add: max.absorb2)

219
next
  case False then have \( b \leq a \) by (auto)
moreover with nn have \( c \cdot b \leq c \cdot a \) by (auto intro: mult-left-mono)
ultimately show \( ?\)thesis by (simp add: max.absorb1)
qed

lemma mult-div-mono-left:
  fixes \( c :: \text{real} \)
  assumes nnc: \( 0 \leq c \) and nzc: \( c \neq 0 \)
  and inv: \( a \leq \text{inverse} \ c \cdot b \)
  shows \( c \cdot a \leq b \)
proof –
  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 \( ... = b \) by (simp)
finally show \( c \cdot a \leq b \).
qed

lemma mult-div-mono-right:
  fixes \( c :: \text{real} \)
  assumes nnc: \( 0 \leq c \) and nzc: \( c \neq 0 \)
  and inv: \( \text{inverse} \ c \cdot a \leq b \)
  shows \( a \leq c \cdot b \)
proof –
  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

lemma min-distrib:
  fixes \( c :: \text{real} \)
  assumes nnc: \( 0 \leq c \)
  shows \( c \cdot \text{min} \ a \ b = \text{min} (c \cdot a) (c \cdot b) \)
proof (cases \( a \leq b \))
  case True moreover with nnc have \( c \cdot a \leq c \cdot b \)
    by (blast intro: mult-left-mono)
ultimately show \( ?\)thesis by (auto)
next
  case False hence \( b \leq a \) by (auto)
moreover with nnc have \( c \cdot b \leq c \cdot a \)
    by (blast intro: mult-left-mono)
ultimately show \( ?\)thesis by (simp add: min.absorb2)
qed

lemma finite-set-least:
  fixes \( S :: 'a :: \text{linorder \ set} \)
  assumes finite: \( \text{finite} \ S \)
  and ne: \( S \neq \{\} \)
4.13. MISCELLANEOUS MATHEMATICS

shows $\exists x \in S. \forall y \in S. x \leq y$

proof -

have $S = \{\} \lor (\exists x \in S. \forall y \in S. x \leq y)$

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

fix $a$ and $S :: 'a set$

assume IH: $S = \{\} \lor (\exists x \in S. \forall y \in S. x \leq y)$

show $(\forall y \in S. x \leq y) \lor (\exists x' \in S. x' \leq x \land (\forall y \in S. x' \leq y))$

proof (cases $S = \{\}$)

case True then show ?thesis by (auto)

next

case False with IH

have $\exists x \in S. \forall y \in S. x \leq y$ by (auto)

then obtain $z$ where $zin: z \in S$ and $zmin: \forall y \in S. z \leq y$ by (auto)

thus ?thesis by (cases $z \leq x$, auto)

qed

with ne show ?thesis by (auto)

qed

lemma cSup-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 cSup-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 $c * \text{Sup} S = \text{Sup} \{c * x | x. x \in S\}$

proof (cases)

assume $c = 0$

moreover from ne have $\exists x. x \in S$ by (auto)

ultimately show ?thesis by (simp)

next
assume \( cnz: c \neq 0 \)

show \( \neg \text{thesis} \)

proof (rule antisym)
  from \( bS \) have \( b\alpha S: \text{bdd-above} \ S \) by (intro \text{bdd-aboveI}, auto)
  with \( ne \ nnc \) show \( \sup \ { c \times x \mid x \in S \} \leq c \times \sup S \)
    by (blast intro: \text{cSup-least} \ \text{mult-left-mono} \ \text{OF cSup-upper})
  have \( \sup S \leq \inverse c \times \sup \ { c \times x \mid x \in S \} \)
    proof (intro \text{cSup-least} \ \text{ne})
      fix \( x \) assume \( \xin: x \in S \)
      moreover from \( bS \) \( nnc \) have \( \bigwedge x. x \in S \implies c \times x \leq c \times b \)
        by (auto intro: \text{mult-left-mono})
      ultimately have \( c \times x \leq \sup \ { c \times x \mid x \in S \} \)
        by (auto intro: \text{cSup-upper} \ \text{bdd-aboveI})
    qed
  with \( cnz \) show \( \sup S \leq c \times \sup \ { c \times x \mid x \in S \} \)
    proof (intro \text{cSup-least} \ \text{ne})
      fix \( x \) assume \( \xin: x \in S \)
      moreover from \( nnc \) have \( 0 \leq \inverse c \)
        by (auto)
      ultimately have \( \inverse c \times (c \times x) \leq \inverse c \times \sup \ { c \times x \mid x \in S \} \)
        by (simp add: \text{mult. assoc}[\text{symmetric}])
    qed
  qed

lemma \text{closure-contains-Sup}: 
fixes \( S :: \text{real set} \)
assumes \( neS: \ S \neq \{\} \) and \( bS: \ \forall x \in S. \ x \leq B \)
shows \( \sup S \in \text{closure} S \)

proof –
  let \( ?T = \uminus ' S \)
  from \( neS \) have \( \neg \ T: \ ? T \neq \{\} \) by (auto)
  from \( bS \) have \( bT: \ \forall x \in ? T. \ \neg B \leq x \) by (auto)
  hence \( b\alpha T: \ \text{bdd-below} \ ? T \) by (intro \text{bdd-belowI}, blast)
  have \( \sup S = - \inf ? T \)
    proof (rule antisym)
      from \( \neg \ T \) \( b\alpha T \)
      have \( \bigwedge x. x \in S \implies \inf (\uminus ' S) \leq - x \)
        by (blast intro: \text{cInf-lower})
      hence \( \bigwedge x. x \in S \implies -1 \times - x \leq -1 \times \inf (\uminus ' S) \)
        by (rule \text{mult-left-mono-neg, auto})
      hence \( \inf \inf: \ \bigwedge x. x \in S \implies x \leq - \inf (\uminus ' S) \)
        by (simp)
      with \( neS \) \( bS \) show \( \sup S \leq - \inf ? T \)
        by (blast intro: \text{cSup-least})
    qed
  have \( - \sup S \leq \inf ? T \)
proof (rule cInf-greatest[OF neT])
  fix \( x \) assume \( x \in \text{uminus} \cdot S \)
  then obtain \( y \) where \( yin: y \in S \) and \( rwx: x = -y \)
  by (auto)
  from \( yin \) \( bS \) have \( y \leq \text{Sup} \ S \)
  by (intro cSup-upper bdd-belowI, auto)
  hence \(-1 \cdot \text{Sup} \ S \leq -1 \cdot y \)
  by (simp add: mult-left-mono-neg)
  with \( rwx \) show \(- \text{Sup} \ S \leq x \)
  by (simp)
  qed
hence \(-1 \cdot \text{Inf} \ ?T \leq -1 \cdot (- \text{Sup} \ S) \)
by (simp add: mult-left-mono-neg)
thus \(- \text{Inf} \ ?T \leq \text{Sup} \ S \)
by (simp)
qed
also {
  from \( neT \) \( bbT \) have \( \text{Inf} \ ?T \in \text{closure} \ ?T \)
  by (rule closure-contains-Inf)
  hence \(- \text{Inf} \ ?T \in \text{uminus} \cdot \text{closure} \ ?T \)
  by (auto)
}
also {
  have linear \text{uminus} by (auto intro: linearI)
  hence \text{uminus} \cdot \text{closure} \ ?T \subseteq \text{closure} \ (\text{uminus} \cdot ?T)
  by (rule closure-linear-image-subset)
}
also {
  have \text{uminus} \cdot ?T \subseteq S by (auto)
  hence \text{closure} \ (\text{uminus} \cdot ?T) \subseteq \text{closure} \ S by (rule closure-mono)
}
finally show \( \text{Sup} \ S \in \text{closure} \ S \).
qed

lemma tendsto-min:
  fixes \( x \ y \cdot \text{real} \)
  assumes ta: \( a \longrightarrow x \)
  and \( tb: b \longrightarrow y \)
  shows \((\lambda i. \text{min} \ (a \ i) \ (b \ i)) \longrightarrow \text{min} \ x \ y \)
proof (rule LIMSEQ-I, simp)
  fix \( e \cdot \text{real} \) assume \( pe: 0 < e \)
from ta pe obtain noa where balla: \( \forall \ n \geq \text{noa}. \ \text{abs} \ (a \ n - x) < e \)
  by (auto dest: LIMSEQ-D)
from tb pe obtain nob where ballb: \( \forall \ n \geq \text{nob}. \ \text{abs} \ (b \ n - y) < e \)
  by (auto dest: LIMSEQ-D)
{
  fix \( n \)
  assume ge: \( \text{max} \ \text{noa} \ \text{nob} \leq n \)
  hence gea: \( \text{noa} \leq n \) and geb: \( \text{nob} \leq n \)
  by (auto)
  have \( \text{abs} \ (\text{min} \ (a \ n) \ (b \ n)) - \text{min} \ x \ y < e \)
  proof cases
    assume le: \( \text{min} \ (a \ n) \ (b \ n) \leq \text{min} \ x \ y \)
show \( \exists \text{thesis} \)

proof cases

assume \( a \ n \leq b \ n \)

hence \( \text{runmin: min} \ (a \ n) \ (b \ n) = a \ n \) by(auto)

with \( \text{le} \) have \( a \ n \leq \text{min} \ x \ y \) by(simp)

moreover from \( \text{gea balla} \) have \( \text{abs} \ (a \ n - x) < e \) by(simp)

moreover have \( \text{min} \ x \ y \leq x \) by(auto)

ultimately have \( \text{abs} \ (a \ n - \text{min} \ x \ y) < e \) by(auto)

with \( \text{runmin} \) show \( \text{abs} \ (\text{min} \ (a \ n) \ (b \ n) - \text{min} \ x \ y) < e \) by(simp)

next

assume \( \neg a \ n \leq b \ n \)

hence \( b \ n \leq a \ n \) by(auto)

hence \( \text{runmin: min} \ (a \ n) \ (b \ n) = b \ n \) by(auto)

with \( \text{le} \) have \( b \ n \leq \text{min} \ x \ y \) by(simp)

moreover from \( \text{geb ballb} \) have \( \text{abs} \ (b \ n - y) < e \) by(simp)

moreover have \( \text{min} \ x \ y \leq y \) by(auto)

ultimately have \( \text{abs} \ (b \ n - \text{min} \ x \ y) < e \) by(auto)

with \( \text{runmin} \) show \( \text{abs} \ (\text{min} \ (a \ n) \ (b \ n) - \text{min} \ x \ y) < e \) by(simp)

qed

next

assume \( \neg \text{min} \ (a \ n) \ (b \ n) \leq \text{min} \ x \ y \)

hence \( \text{le} \) min \( x \ y \leq \text{min} \ (a \ n) \ (b \ n) \) by(auto)

show \( \exists \text{thesis} \)

proof cases

assume \( x \leq y \)

hence \( \text{runmin: min} \ x \ y = x \) by(auto)

with \( \text{le} \) have \( x \leq \text{min} \ (a \ n) \ (b \ n) \) by(simp)

moreover from \( \text{gea balla} \) have \( \text{abs} \ (a \ n - x) < e \) by(simp)

moreover have \( \text{min} \ (a \ n) \ (b \ n) \leq a \ n \) by(auto)

ultimately have \( \text{abs} \ (\text{min} \ (a \ n) \ (b \ n) - x) < e \) by(auto)

with \( \text{runmin} \) show \( \text{abs} \ (\text{min} \ (a \ n) \ (b \ n) - \text{min} \ x \ y) < e \) by(simp)

next

assume \( \neg x \leq y \)

hence \( y \leq x \) by(auto)

hence \( \text{runmin: min} \ x \ y = y \) by(auto)

with \( \text{le} \) have \( y \leq \text{min} \ (a \ n) \ (b \ n) \) by(simp)

moreover from \( \text{geb ballb} \) have \( \text{abs} \ (b \ n - y) < e \) by(simp)

moreover have \( \text{min} \ (a \ n) \ (b \ n) \leq b \ n \) by(auto)

ultimately have \( \text{abs} \ (\text{min} \ (a \ n) \ (b \ n) - y) < e \) by(auto)

with \( \text{runmin} \) show \( \text{abs} \ (\text{min} \ (a \ n) \ (b \ n) - \text{min} \ x \ y) < e \) by(simp)

qed

qed

\}

thus \( \exists \text{no.} \ \forall \text{n} \geq \text{no.} \ |\text{min} \ (a \ n) \ (b \ n) - \text{min} \ x \ y| < e \) by(blast)

qed

definition \( \text{supp} :: \ (\text{'s} \Rightarrow \text{real}) \Rightarrow \text{'s set} \)

where \( \text{supp} \ f = \{x. \ f \ x \neq 0\} \)
4.13. MISCELLANEOUS MATHEMATICS

**Definition** dist-remove :: ('s ⇒ real) ⇒ 's ⇒ 's ⇒ real

**Where** dist-remove p x = (λy. if y=x then 0 else p y / (1 - p x))

**Lemma** supp-dist-remove:

\[ p x \neq 0 \implies p x \neq 1 \implies \text{supp}(\text{dist-remove} p x) = \text{supp} p - \{x\} \]

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** sum-supp:

fixes f :: 'a :: finite ⇒ real

shows \( \sum f (\text{supp} f) = \sum f \text{ UNIV} \)

**Proof**

- have \( \sum f (\text{UNIV} - \text{supp} f) = 0 \)
  
  by(simp add:supp-def)

- hence \( \sum f (\text{supp} f) = \sum f (\text{UNIV} - \text{supp} f) + \sum f (\text{supp} f) \)
  
  by(simp)

- also have ... = \( \sum f \text{ UNIV} \)
  
  by(simp add:sum subset-diff[symmetric])

- finally show ?thesis .

qed

4.13.1 Truncated Subtraction

**Definition**

tminus :: real ⇒ real ⇒ real (infixl ⊖ 60)

**Where**

\[ x \ominus y = \max (x - y) 0 \]

**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]:

\[ 0 \leq x \implies x \ominus 0 = x \]

by(simp add:tminus-def)

**Lemma tminus-left-mono:**
\[ a \leq b \implies a \ominus c \leq b \ominus c \]

**Unfolding tminus-def by:** (case-tac a ≤ c, simp-all)

**Lemma tminus-less:**
\[ \begin{array}{l}
[0 \leq a; 0 \leq b] \implies a \ominus b \leq a
\end{array} \]

**Unfolding tminus-def by:** (force)

**Lemma tminus-left-distrib:**
assumes nna: 0 ≤ a
shows \( a \ast (b \ominus c) = a \ast b \ominus a \ast c \)
**Proof:**
\[
\begin{array}{l}
\text{by (cases } b \leq c) \\
\text{case True note } le = this \\
\text{hence } a \ast \max (b - c) 0 = 0 \text{ by (simp add:max.absorb2)} \\
\text{also } \\
\text{ from nna le have } a \ast b \leq a \ast c \text{ by (blast intro:mult-left-mono)} \\
\text{hence } 0 = \max (a \ast b - a \ast c) 0 \text{ by (simp add:max.absorb1)} \\
\end{array}
\]

**Finally show ?thesis by:** (simp add:tminus-def)

**Next**
\[
\begin{array}{l}
\text{case False hence } le: c \leq b \text{ by (auto)} \\
\text{hence } a \ast \max (b - c) 0 = 0 \text{ by (simp only:max.absorb1)} \\
\text{also } \\
\text{ from nna le have } a \ast c \leq a \ast b \text{ by (blast intro:mult-left-mono)} \\
\text{hence } a \ast (b - c) = \max (a \ast b - a \ast c) 0 \text{ by (simp add:max.absorb1)} \\
\text{field-simps) } \\
\text{finally show ?thesis by:** (simp add:tminus-def) }
\end{array}
\]

**Qed**

**Lemma tminus-le[simp]:**
\[ b \leq a \implies a \ominus b = a - b \]

**Unfolding tminus-def by:** (simp)

**Lemma tminus-le-alt[simp]:**
\[ a \leq b \implies a \ominus b = 0 \]

**By:** (simp add:tminus-def)

**Lemma tminus-nle[simp]:**
\[ \neg b \leq a \implies a \ominus b = 0 \]

**Unfolding tminus-def by:** (simp)

**Lemma tminus-add-mono:**
\[ (a+b) \ominus (c+d) \leq (a\ominus c) + (b\ominus d) \]

**Proof:** (cases 0 ≤ a − c)
\[
\begin{array}{l}
\text{case True note } pac = this
\end{array}
\]
4.13. MISCELLANEOUS MATHEMATICS

show ?thesis
proof (cases $0 \leq b - d$)
  case True note pbd = this
  from pac and pbd have $(c + d) \leq (a + b)$ by (simp)
  with pac and pbd show ?thesis by (simp)
next
  case False with pac show ?thesis
  by (cases $c + d \leq a + b$, auto)
qed
next
  case False note nac = this
  show ?thesis
  proof (cases $0 \leq b - d$)
    case True with nac show ?thesis
    by (cases $c + d \leq a + b$, auto)
  next
    case False note nbd = this
    with nac have $\neg (c + d) \leq (a + b)$ by (simp)
    with nac and nbd show ?thesis by (simp)
  qed
qed

lemma tminus-sum-mono:
  assumes fS: finite S
  shows $\sum f S \ominus \sum g S \leq \sum (\lambda x. f x \ominus g x) S$
  (is $?X S$)
proof (rule finite-induct)
  from fS show finite S .
  show $?X \{\}$ by (simp)
  fix $x$ and $F$
  assume fF: finite $F$ and xniF: $x \notin F$
  and IH: $?X F$
  have $f x + \sum f F \ominus g x + \sum g F \leq$
    $(f x \ominus g x) + (\sum x \in F. f x \ominus g x)$
  by (rule tminus-add-mono)
  also from IH have ... $\leq (f x \ominus g x) + (\sum x \in F. f x \ominus g x)$
  by (rule add-left-mono)
  finally show $?X (\text{insert } x F)$
  by (simp add: sum.insert[OF fF xniF])
qed

lemma tminus-nneg[simp,intro]:
  $0 \leq a \ominus b$
  by (cases $b \leq a$, auto)

lemma tminus-right-antimono:
  assumes clb: $c \leq b$
shows $a \odot b \leq a \odot c$

proof (cases $b \leq a$)
  case True
  moreover with $clb$ have $c \leq a$ by (auto)
  moreover note $clb$
  ultimately show $\textup{thesis}$ by (simp)
next
  case False then show $\textup{thesis}$ by (simp)
  qed

lemma min-tminus-distrib:
  $\min a b \odot c = \min (a \odot c) (b \odot c)$
  unfolding tminus-def by (auto)
Bibliography


