pGCL for Isabelle

David Cock

April 20, 2020
# Contents

## 1 Overview

## 2 Introduction to pGCL

### 2.1 Language Primitives

#### 2.1.1 The Basics

#### 2.1.2 Assertion and Annotation

#### 2.1.3 Probability

#### 2.1.4 Nondeterminism

#### 2.1.5 Properties of Expectations

### 2.2 Loops

#### 2.2.1 Guaranteed Termination

#### 2.2.2 Probabilistic Termination

### 2.3 The Monty Hall Problem

#### 2.3.1 The State Space

#### 2.3.2 The Game

#### 2.3.3 A Brute Force Solution

#### 2.3.4 A Modular Approach

## 3 Semantic Structures

### 3.1 Expectations

#### 3.1.1 Bounded Functions

#### 3.1.2 Non-Negative Functions

#### 3.1.3 Sound Expectations

#### 3.1.4 Unitary expectations

#### 3.1.5 Standard Expectations

#### 3.1.6 Entailment

#### 3.1.7 Expectation Conjunction

#### 3.1.8 Rules Involving Conjunction

#### 3.1.9 Rules Involving Entailment and Conjunction Together

### 3.2 Expectation Transformers

#### 3.2.1 Comparing Transformers

#### 3.2.2 Healthy Transformers

#### 3.2.3 Sublinearity
3.2.4 Determinism ........................................ 42
3.2.5 Modular Reasoning ............................... 44
3.2.6 Transforming Standard Expectations ............. 45

3.3 Induction ............................................. 46
3.3.1 The Lattice of Expectations ..................... 46
3.3.2 The Lattice of Transformers ..................... 49
3.3.3 Tail Recursion .................................... 52

4 The pGCL Language ................................. 55
4.1 A Shallow Embedding of pGCL in HOL ................. 55
4.1.1 Core Primitives and Syntax ..................... 55
4.1.2 Unfolding rules for non-recursive primitives ... 58
4.2 Healthiness ......................................... 60
4.2.1 The Healthiness of the Embedding ............... 61
4.2.2 Healthiness for Loops ............................ 63
4.3 Continuity ........................................... 66
4.3.1 Continuity of Primitives ....................... 67
4.3.2 Continuity of a Single Loop Step ............... 70
4.4 Continuity and Induction for Loops ................. 70
4.4.1 The Limit of Iterates ............................ 71
4.5 Sublinearity ......................................... 73
4.5.1 Nonrecursive Primitives ....................... 73
4.5.2 Sublinearity for Loops ......................... 75
4.6 Determinism ......................................... 76
4.6.1 Additivity ....................................... 76
4.6.2 Maximality ....................................... 77
4.6.3 Determinism .................................... 78
4.7 Well-Defined Programs ................................ 79
4.7.1 Strict Implies Liberal ........................... 80
4.7.2 Sub-Distributivity of Conjunction .............. 82
4.7.3 The Well-Defined Predicate .................... 84
4.8 The Loop Rules .................................... 86
4.8.1 Liberal and Strict Invariants .................. 86
4.8.2 Partial Correctness ............................. 87
4.8.3 Total Correctness ............................... 87
4.8.4 Unfolding ....................................... 88
4.9 The Algebra of pGCL ............................... 88
4.9.1 Program Refinement ............................. 88
4.9.2 Simple Identities ............................... 89
4.9.3 Deterministic Programs are Maximal .......... 92
4.9.4 The Algebraic Structure of Refinement ........ 93
4.9.5 Data Refinement ................................ 94
4.9.6 The Algebra of Data Refinement ................ 96
4.9.7 Structural Rules for Correspondence .......... 97
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.9.8</td>
<td>Structural Rules for Data Refinement</td>
<td>97</td>
</tr>
<tr>
<td>4.10</td>
<td>Structured Reasoning</td>
<td>98</td>
</tr>
<tr>
<td>4.10.1</td>
<td>Syntactic Decomposition</td>
<td>98</td>
</tr>
<tr>
<td>4.10.2</td>
<td>Algebraic Decomposition</td>
<td>102</td>
</tr>
<tr>
<td>4.10.3</td>
<td>Hoare triples</td>
<td>103</td>
</tr>
<tr>
<td>4.11</td>
<td>Loop Termination</td>
<td>103</td>
</tr>
<tr>
<td>4.11.1</td>
<td>Trivial Termination</td>
<td>103</td>
</tr>
<tr>
<td>4.11.2</td>
<td>Classical Termination</td>
<td>104</td>
</tr>
<tr>
<td>4.11.3</td>
<td>Probabilistic Termination</td>
<td>104</td>
</tr>
<tr>
<td>4.12</td>
<td>Automated Reasoning</td>
<td>105</td>
</tr>
<tr>
<td>4.13</td>
<td>Miscellaneous Mathematics</td>
<td>107</td>
</tr>
<tr>
<td>4.13.1</td>
<td>Truncated Subtraction</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>Additional Material</td>
<td></td>
</tr>
</tbody>
</table>

**4.13 Miscellaneous Mathematics**

- 4.13.1 Truncated Subtraction 107
Chapter 1

Overview

pGCL is both a programming language and a specification language that incorporates both probabilistic and nondeterministic choice, in a unified manner. Program verification is by refinement or annotation (or both), using either Hoare triples, or weakest-precondition entailment, in the style of GCL [Dijkstra, 1975].

This document is divided into three parts: Chapter 2 gives a tutorial-style introduction to pGCL, and demonstrates the tools provided by the package; Chapter 3 covers the development of the semantic interpretation: expectation transformers; and Chapter 4 covers the formalisation of the language primitives, the associated healthiness results, and the tools for structured and automated reasoning. This second part follows the technical development of the pGCL theory package, in detail. It is not a great place to start learning pGCL. For that, see either the tutorial or McIver and Morgan [2004].

This formalisation was first presented (as an overview) in Cock [2012]. The language has previously been formalised in HOL4 by Hurd et al. [2005]. Two substantial results using this package were presented in Cock [2013], Cock [2014a] and Cock [2014b].
Chapter 2

Introduction to pGCL

2.1 Language Primitives

theory Primitives imports ..:/pGCL begin

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

2.1.1 The Basics

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

```plaintext
datatype coin = Heads | Tails

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

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

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

(proof)

We can treat the record’s fields as the names of variables. Note that the right-hand side of an assignment is always a function of the current state. Thus we may use a record accessor directly, for example $\text{Apply} \ (\lambda s. s[(a := b \ s)])$, 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 $P :: bool$ is embedded as « $P$ » :: 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 « $P$ » ⊢ « $Q$ », « ?P » ⊢ « ?Q »; ?P ?s] ⊢ ?Q ?s. Thus a valid specification of \( \text{Apply} (\lambda s. s(a := b s)) \) is:

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

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

2.1.3 Probability

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

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

So far, we haven’t done anything that required probabilities, or expectations other than 0 and 1. As an example of both, we show that a single coin toss is fair. We introduce the syntax $x p \oplus y$ for a probabilistic choice between $x$ and $y$. This program behaves as $x$ with probability $p$, and as $y$ with probability $(1::’a) - p$. The probability may depend on the state, and is therefore of
type ’s ⇒ real. The following annotation states that the probability of heads
is exactly 1/2:

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

lemma
(λs. 1/2) = wp (flip-a (1/2)) «λs. a = Heads»
⟨proof⟩

2.1.4 Nondeterminism

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

lemma
λs. 1/3 ⊏ wp (flip-a (2/3) ⊏ flip-a (1/3)) «λs. a = Heads»
⟨proof⟩

2.1.5 Properties of Expectations

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

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

lemma
(λs. 1/4) = wp (flip-a (1/2) ;; flip-b (1/2)) «λs. a = Heads ∧ b = Heads»
⟨proof⟩

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

record dice =
  red :: nat
  blue :: nat

definition Puniform :: ’a set ⇒ (’a ⇒ real)
where \( \text{Puniform} \ S = (\lambda x. \text{if } x \in S \text{ then } 1 / \text{card } S \text{ else } 0) \)

**lemma** \( \text{Puniform-in} \):
\[
x \in S \implies \text{Puniform} \ S \ x = 1 / \text{card } S
\]
\langle proof \rangle

**lemma** \( \text{Puniform-out} \):
\[
x \not\in S \implies \text{Puniform} \ S \ x = 0
\]
\langle proof \rangle

**lemma** \( \text{supp-Puniform} \):
\[
\text{finite } S \implies \text{supp} (\text{Puniform} \ S) = S
\]
\langle proof \rangle

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

**lemma**
\[
(\lambda s. 7 / 2) = \text{wp} ((\text{bind } v \text{ at } (\lambda s. \text{Puniform} \ \{1..6\} \ v) \text{ in red := } (\lambda -. v)) \text{ red}
\]
\langle proof \rangle

The expectations of independent variables add:

**lemma**
\[
(\lambda s. \red s + \blue s)
\]
\langle proof \rangle

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:

**definition** countdown :: int prog
where
\[
countdown = \text{do } (\lambda x. \ 0 < x) \rightarrow \text{Apply } (\lambda s. s - 1) \text{ od}
\]
2.2. LOOPS

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

**definition** inv-count :: int ⇒ bool
**where**
inv-count = (\(\lambda x. 0 \leq x\))

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

**lemma** wp-inv-count:
\(wp-inv (\lambda x. 0 < x) (Apply (\lambda s. s - 1)) « inv-count\)
(proof)

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

**lemma** term-countdown:
« inv-count » ⊢ ⊢ wp countdown (\(\lambda s. 1\))
(proof)

### 2.2.2 Probabilistic Termination

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

**type-synonym** coin = bool
**definition** Heads = True
**definition** Tails = False

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

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

**definition**
\(wait-for-heads :: coin prog\)
**where**
\(wait-for-heads = do ((\neq) Heads) \rightarrow flip od\)

Nonetheless, we can show termination .

**lemma** wait-for-heads-term:
\(\lambda s. 1 \vdash wp wait-for-heads (\lambda s. 1)\)
(proof)
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.

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

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

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

Invariants

We prove explicitly that only valid doors are ever chosen.

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. THE MONTY HALL PROBLEM

2.3.2 The Game

Hide the prize behind door $D$.

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

Choose door $D$.

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

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

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

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

**definition** hide-prize :: game prog
where hide-prize $\equiv$ hide-behind 1 $\sqcap$ hide-behind 2 $\sqcap$ hide-behind 3

Guess uniformly at random.

**definition** make-guess :: game prog
where make-guess $\equiv$ guess-behind 1 ($\lambda s. 1/3$) $\oplus$

$\quad$ guess-behind 2 ($\lambda s. 1/2$) $\oplus$ guess-behind 3

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

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

Switch your guess to the other unopened door.

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

The complete game, either with or without switching guesses.

**definition** monty :: bool $\Rightarrow$ game prog
where

$\quad$ monty switch $\equiv$ hide-prize $;$
$\quad$ make-guess $;$
$\quad$ reveal $;$
$\quad$ (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]:
$\quad\quad p = g \implies \langle \text{player-wins} \rangle (s\| \text{prize := p, guess := g, clue := c}) = 1$
Lemma eval-loss[simp]:
\[ p \neq g \implies «player-wins» (s \parallel \text{prize} := p, \text{guess} := g, \text{clue} := c \parallel) = 0 \]

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

Lemma wp-monty-noswitch:
\[ (\lambda s. 1/3) = \text{wp (monty False) «player-wins»} \]

Lemma swap-upd:
\[ s \parallel \text{prize} := p, \text{clue} := c, \text{guess} := g \parallel = s \parallel \text{prize} := p, \text{guess} := g, \text{clue} := c \parallel \]

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

Lemma wp-monty-switch-bruteforce:
\[ (\lambda s. 2/3) = \text{wp (monty True) «player-wins»} \]

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:
\[ \text{well-def hide-prize} \]

Lemma wd-make-guess:
\[ \text{well-def make-guess} \]

Lemma wd-reveal:
\[ \text{well-def reveal} \]
2.3. THE MONTY HALL PROBLEM

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

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

Annotations

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

hide-prize chooses a valid door.

lemma wp-hide-prize:
  (λs. 1) ⊢ wp hide-prize «inv-prize»
⟨proof⟩

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

lemma wp-make-guess:
  (λs. 2/3 * «λg. inv-prize g» s) ⊢
  wp make-guess «λg. guess g ≠ prize g ∧ inv-prize g ∧ inv-guess g»
⟨proof⟩

lemma last-one:
  assumes a ≠ b and a ∈ {1::nat,2,3} and b ∈ {1,2,3}
  shows ∃!c. {1,2,3} − {b,a} = {c}
⟨proof⟩

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

lemma wp-reveal:
  «λg. guess g ≠ prize g ∧ inv-prize g ∧ inv-guess g» ⊢
  wp reveal «λg. guess g ≠ prize g ∧
  clue g ≠ prize g ∧
  clue g ≠ guess g ∧
  inv-prize g ∧ inv-guess g ∧ inv-clue g»
(is ?X ⊬ wp reveal ?Y)
⟨proof⟩

Showing that the three doors are all district is a largeish first-order problem, for which sledgehammer gives us a reasonable script.

lemma distinct-game:
  [ guess g ≠ prize g; clue g ≠ prize g; clue g ≠ guess g;
  inv-prize g; inv-guess g; inv-clue g ] →
  {1, 2, 3} = {guess g, prize g, clue g}
⟨proof⟩
Given the invariants, switching from the wrong guess gives the right one.

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

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

\[
\vdash \vdash \text{wp switch-guess} \ \langle \text{player-wins} \rangle
\]

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

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

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

**Using the VCG**

**Lemmas** \( \text{scaled-hide} = \text{wp-scale}[OF \ \text{wp-hide-prize, simplified}] \)

**Declare** \( \text{scaled-hide}[\text{pwp}] \ \text{wp-make-guess}[\text{pwp}] \ \text{wp-reveal}[\text{pwp}] \ \text{wp-switch-guess}[\text{pwp}] \)

**Declare** \( \text{wd-hide-prize}[\text{wd}] \ \text{wd-make-guess}[\text{wd}] \ \text{wd-reveal}[\text{wd}] \ \text{wd-switch-guess}[\text{wd}] \)

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

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

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

**End**
Chapter 3

Semantic Structures

3.1 Expectations

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

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

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

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

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

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

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

### 3.1.1 Bounded Functions

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

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

lemma bounded-byI[intro]:

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

lemma bounded-byI2[intro]:

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

lemma bounded-byD[dest]:

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

lemma bounded-byD2[dest]:

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

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

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

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

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

lemma bounded-bdd-above[intro]:

assumes $bP; \text{ bounded } P$
shows $\text{bdd-above } (\text{range } P)$

The least upper bound has the usual properties:
lemma bound-of-least[intro]:
  assumes bP: bounded-by b P
  shows bound-of P ≤ b
  ⟨proof⟩

lemma bounded-by-bound-of[intro]:
  fixes P::('a ⇒ real)
  assumes bP: bounded P
  shows bounded-by (bound-of P) P
  ⟨proof⟩

lemma bound-of-greater[intro]:
  bounded P =⇒ P x ≤ bound-of P
  ⟨proof⟩

lemma bounded-by-mono:
  [ bounded-by a P; a ≤ b ] =⇒ bounded-by b P
  ⟨proof⟩

lemma bounded-by-imp-bounded[intro]:
  bounded-by b P =⇒ bounded P
  ⟨proof⟩

This is occasionally easier to apply:

lemma bounded-by-bound-of-alt:
  [ bounded P; bound-of P = a ] =⇒ bounded-by a P
  ⟨proof⟩

lemma bounded-const[simp]:
  bounded (λx. c)
  ⟨proof⟩

lemma bounded-by-const[intro]:
  c ≤ b =⇒ bounded-by b (λx. c)
  ⟨proof⟩

lemma bounded-by-mono-alt[intro]:
  [ bounded-by b Q; P ≤ Q ] =⇒ bounded-by b P
  ⟨proof⟩

lemma bound-of-const[simp, intro]:
  bound-of (λx. c) = (c::real)
  ⟨proof⟩

lemma bound-of-leI:
  assumes ∃x. P x ≤ (c::real)
  shows bound-of P ≤ c
  ⟨proof⟩
CHAPTER 3. SEMANTIC STRUCTURES

**Lemma** bound-of-mono[intro]:

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

⟨proof⟩

**Lemma** bounded-by-o[intro,simp]:

\[\forall b. \text{bounded-by } b \ P \implies \text{bounded-by } b \ (P \circ f)\]

⟨proof⟩

**Lemma** le-bound-of[intro]:

\[\forall x. \text{bounded } f \implies f \, x \leq \text{bound-of } f\]

⟨proof⟩

### 3.1.2 Non-Negative Functions.

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

**Definition**

\[\text{nneg} :: \text{'(a }\Rightarrow\text{'b;\{zero,order\}) }\Rightarrow\text{ bool}\]

**Where**

\[\text{nneg } P \longleftrightarrow (\forall x. 0 \leq P \, x)\]

**Lemma** nnegI[intro]:

\[\[
\forall x. 0 \leq P \, x \implies \text{nneg } P
\]

⟨proof⟩

**Lemma** nnegI2[intro]:

\[\lambda x. 0 \leq P \implies \text{nneg } P\]

⟨proof⟩

**Lemma** nnegD[dest]:

\[\text{nneg } P \implies 0 \leq P \, x\]

⟨proof⟩

**Lemma** nnegD2[dest]:

\[\text{nneg } P \implies (\lambda x. 0) \leq P\]

⟨proof⟩

**Lemma** nneg-bdd-below[intro]:

\[\text{nneg } P \implies \text{bdd-below } (\text{range } P)\]

⟨proof⟩

**Lemma** nneg-const[iff]:

\[\text{nneg } (\lambda x. c) \longleftrightarrow 0 \leq c\]

⟨proof⟩

**Lemma** nneg-o[intro,simp]:

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

⟨proof⟩
3.1. EXPECTATIONS

\textbf{Lemma} \texttt{nneg-bound-nneg[intro]}:
\[
\begin{align*}
\text{bounded } P, \text{ nneg } P \quad \Rightarrow \quad 0 \leq \text{bound-of } P \\
\text{(proof)}
\end{align*}
\]

\textbf{Lemma} \texttt{nneg-bounded-by-nneg[dest]}:
\[
\begin{align*}
\text{bounded-by } b \times P, \text{ nneg } P \quad \Rightarrow \quad 0 \leq (b :: \text{real}) \\
\text{(proof)}
\end{align*}
\]

\textbf{Lemma} \texttt{bounded-by-nneg[dest]}:
\[
\begin{align*}
\text{fixes } P :: \text{'}s \Rightarrow \text{real} \\
\text{shows } \text{bounded-by } b \times P, \text{ nneg } P \quad \Rightarrow \quad 0 \leq b \\
\text{(proof)}
\end{align*}
\]

3.1.3 Sound Expectations

\textbf{Definition} \texttt{sound :: ('}s \Rightarrow \text{real}) \Rightarrow \text{bool}
\textbf{where} \texttt{sound } P \equiv \text{bounded } P \land \text{nneg } P

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

\textbf{Lemma} \texttt{soundI}:
\[
\begin{align*}
\text{bounded } P, \text{ nneg } P \quad \Rightarrow \quad \text{sound } P \\
\text{(proof)}
\end{align*}
\]

\textbf{Lemma} \texttt{soundI2[intro]}:
\[
\begin{align*}
\text{bounded-by } b \times P, \text{ nneg } P \quad \Rightarrow \quad \text{sound } P \\
\text{(proof)}
\end{align*}
\]

\textbf{Lemma} \texttt{sound-bounded[dest]}:
\[
\begin{align*}
\text{sound } P \Rightarrow \text{bounded } P \\
\text{(proof)}
\end{align*}
\]

\textbf{Lemma} \texttt{sound-nneg[dest]}:
\[
\begin{align*}
\text{sound } P \Rightarrow \text{nneg } P \\
\text{(proof)}
\end{align*}
\]

\textbf{Lemma} \texttt{bound-of-sound[intro]}:
\[
\begin{align*}
\text{assumes } sP :: \text{sound } P \\
\text{shows } 0 \leq \text{bound-of } P \\
\text{(proof)}
\end{align*}
\]

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

\textbf{Lemma} \texttt{sound-sum[simp,intro]}:
\[
\begin{align*}
\text{assumes } sP :: \text{sound } P \text{ and } sQ :: \text{sound } Q \\
\text{shows } \text{sound } (\lambda s. P s + Q s) \\
\text{(proof)}
\end{align*}
\]
lemma mult-sound:
  assumes \( sP: \text{sound } P \) and \( sQ: \text{sound } Q \)
  shows \( \text{sound } (\lambda s. P s * Q s) \)
  ⟨proof⟩

lemma div-sound:
  assumes \( sP: \text{sound } P \) and \( \text{cpos: } 0 < c \)
  shows \( \text{sound } (\lambda s. P s / c) \)
  ⟨proof⟩

lemma tminus-sound:
  assumes \( sP: \text{sound } P \) and \( \text{nnc: } 0 \leq c \)
  shows \( \text{sound } (\lambda s. P s \ominus c) \)
  ⟨proof⟩

lemma const-sound:
  \( 0 \leq c \Rightarrow \text{sound } (\lambda s. c) \)
  ⟨proof⟩

lemma sound-o[\{intro,simp\}):
  \( \text{sound } P \Rightarrow \text{sound } (P o f) \)
  ⟨proof⟩

lemma sc-bounded-by[\{intro,simp\}]:
  \[ \text{sound } P; 0 \leq c \Rightarrow \text{bounded-by } (c * \text{bound-of } P) (\lambda x. c * P x) \]
  ⟨proof⟩

lemma sc-bounded[\{intro,simp\}]:
  assumes \( sP: \text{sound } P \) and \( \text{pos: } 0 \leq c \)
  shows \( \text{bounded } (\lambda x. c * P x) \)
  ⟨proof⟩

lemma sc-bound[\{simp\}]:
  assumes \( sP: \text{sound } P \)
  and \( \text{nnc: } 0 \leq c \)
  shows \( c * \text{bound-of } P = \text{bound-of } (\lambda x. c * P x) \)
  ⟨proof⟩

lemma sc-sound:
  \[ \text{sound } P; 0 \leq c \Rightarrow \text{sound } (\lambda s. c * P s) \]
  ⟨proof⟩

lemma bounded-by-mult:
  assumes \( sP: \text{sound } P \) and \( bP: \text{bounded-by } a P \)
  and \( sQ: \text{sound } Q \) and \( bQ: \text{bounded-by } b Q \)
  shows \( \text{bounded-by } (a * b) (\lambda s. P s * Q s) \)
  ⟨proof⟩
3.1. EXPECTATIONS

lemma bounded-by-add:
  fixes P :: 's ⇒ real and Q
  assumes bP: bounded-by a P
     and bQ: bounded-by b Q
  shows bounded-by (a + b) (λs. P s + Q s)
  ⟨proof⟩

lemma sound-unit[intro!, simp]:
  sound (λs. 1)
  ⟨proof⟩

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 (λs. P s * Q s)
  ⟨proof⟩

lemma sum-sound:
  assumes sP: ∀x∈S. sound (P x)
  shows sound (λs. ∑x∈S. P x s)
  ⟨proof⟩

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
  ⟨proof⟩

lemma unitaryI2:
  [ nneg P; bounded-by 1 P ] ⇒ unitary P
  ⟨proof⟩

lemma unitary-sound[dest]:
  unitary P ⇒ sound P
  ⟨proof⟩

lemma unitary-bound[dest]:
  unitary P ⇒ bounded-by 1 P
  ⟨proof⟩

3.1.5 Standard Expectations

definition
CHAPTER 3. SEMANTIC STRUCTURES

\[ embed\text{-}bool :: ('s ⇒ bool) ⇒ 's ⇒ real (« - » 1000) \]

where
\[ « P » ≡ (\lambda x. \text{if } P x \text{ then } 1 \text{ else } 0) \]

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

\textbf{lemma embed-bool-nneg}[simp,intro]:
\[ \text{nneg « P »} \]
\(<proof>\)

\textbf{lemma embed-bool-bounded-by-1}[simp,intro]:
\[ \text{bounded-by } 1 « P » \]
\(<proof>\)

\textbf{lemma embed-bool-bounded}[simp,intro]:
\[ \text{bounded } « P » \]
\(<proof>\)

Standard expectations have a number of convenient properties, which mostly follow from boolean algebra.

\textbf{lemma embed-bool-idem}:
\[ « P » s * « P » s = « P » s \]
\(<proof>\)

\textbf{lemma eval-embed-true}[simp]:
\[ P s ⇒ « P » s = 1 \]
\(<proof>\)

\textbf{lemma eval-embed-false}[simp]:
\[ \neg P s ⇒ « P » s = 0 \]
\(<proof>\)

\textbf{lemma embed-ge-0}[simp,intro]:
\[ 0 \leq « G » s \]
\(<proof>\)

\textbf{lemma embed-le-1}[simp,intro]:
\[ « G » s \leq 1 \]
\(<proof>\)

\textbf{lemma embed-le-1-alt}[simp,intro]:
\[ 0 \leq 1 - « G » s \]
\(<proof>\)

\textbf{lemma expect-1-I}:
\[ P x ⇒ 1 \leq « P » x \]
\(<proof>\)
3.1. EXPECTATIONS

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

\[\text{sound} \ «P»\]

(proof)

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

\[«P» o f = «P o f»\]

(proof)

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

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

where negate P = (λs. ¬P s)

**Lemma** negateI:

\[¬P s \implies N P s\]

(proof)

**Lemma** embed-split:

\[f s = «P» s + «N P» s * f s\]

(proof)

**Lemma** negate-embed:

\[«N P» s = 1 - «P» s\]

(proof)

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

\[P s \implies «N P» s = 0\]

(proof)

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

\[¬P s \implies «N P» s = 1\]

(proof)

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

\[N \ Not = (λx. x)\]

(proof)

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

\[N (N P) = P\]

(proof)

**Lemma** embed-bool-cancel:

\[«G» s * «N G» s = 0\]

(proof)
3.1.6 Entailment

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

**abbreviation** entails :: (′s ⇒ real) ⇒ (′s ⇒ real) ⇒ bool (⋅ ⊢ ⋅ 50)

**where** P ⊢ Q ≡ P ≤ Q

**lemma** entailsI[intro]:

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

**⟨proof⟩**

**lemma** entailsD[dest]:

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

**⟨proof⟩**

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

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

**⟨proof⟩**

**lemma** entails-trans[trans]:

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

**⟨proof⟩**

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

**lemma** implies-entails:

\[ \forall s. P s \Rightarrow Q s \implies \langle P \rangle \vdash \langle Q \rangle \]

**⟨proof⟩**

**lemma** entails-implies:

\[ \forall s. \langle P \rangle \vdash \langle Q \rangle; P s \]

**⟨proof⟩**

3.1.7 Expectation Conjunction

**definition**

pconj :: real ⇒ real ⇒ real (infixl .& 71)

**where**

p .& q ≡ p + q ⊖ 1

**definition**

exp-conj :: (′s ⇒ real) ⇒ (′s ⇒ real) ⇒ (′s ⇒ real) (infixl .& 71)

**where** a .& b ≡ λs. (a . s .& b s)

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

lemma pconj-lzero[intro,simp]:
  \( b \leq 1 \implies 0 \land b = 0 \)
  ⟨proof⟩

lemma pconj-rzero[intro,simp]:
  \( b \leq 1 \implies b \land 0 = 0 \)
  ⟨proof⟩

lemma pconj-lone[intro,simp]:
  \( 0 \leq b \implies 1 \land b = b \)
  ⟨proof⟩

lemma pconj-rone[intro,simp]:
  \( 0 \leq b \implies b \land 1 = b \)
  ⟨proof⟩

lemma pconj-bconj:
  \( «a» \land «b» = «\lambda s. a \land b» \land s \)
  ⟨proof⟩

lemma pconj-comm[ac-simps]:
  \( a \land b = b \land a \)
  ⟨proof⟩

lemma pconj-assoc:
  \[ [ \theta \leq a; a \leq 1; 0 \leq b; b \leq 1; 0 \leq c; c \leq 1 ] \implies a \land (b \land c) = (a \land b) \land c \]
  ⟨proof⟩

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

lemma pconj-nneg[intro,simp]:
  \( \theta \leq a \land b \)
  ⟨proof⟩

lemma min-pconj:
  \( (\min a b) \land (\min c d) \leq \min (a \land c) (b \land d) \)
  ⟨proof⟩

lemma pconj-less-one[simp]:
  \( a + b < 1 \implies a \land b = 0 \)
  ⟨proof⟩

lemma pconj-ge-one[simp]:
  \( 1 \leq a + b \implies a \land b = a + b - 1 \)
  ⟨proof⟩
lemma \texttt{pconj-idem[simp]}:

\begin{quote}
\[ «P» \land «P» = «P» \]
\end{quote}

(\texttt{proof})

\subsection*{3.1.8 Rules Involving Conjunction.}

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

(\texttt{proof})

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

(\texttt{proof})

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

(\texttt{proof})

\textbf{lemma} \texttt{exp-conj-bounded-by[intro,simp]}:\newline
\begin{quote}
\begin{align*}
\text{assumes } & bP; \text{ bounded-by } 1 P \\
\text{and } & bQ; \text{ bounded-by } 1 Q \\
\text{shows } & \text{ bounded-by } 1 (P \land Q)
\end{align*}
\end{quote}

(\texttt{proof})

\textbf{lemma} \texttt{exp-conj-o-distrib[simp]}:\newline
\[ (P \land Q) o f = (P o f) \land (Q o f) \]

(\texttt{proof})

\textbf{lemma} \texttt{exp-conj-assoc}:\newline
\begin{quote}
\begin{align*}
\text{assumes unitary } & P \text{ and unitary } Q \text{ and unitary } R \\
\text{shows } & P \land (Q \land R) = (P \land Q) \land R
\end{align*}
\end{quote}

(\texttt{proof})

\textbf{lemma} \texttt{exp-conj-top-left[simp]}:\newline
\begin{quote}
\[ \text{sound } P \implies «\lambda - \text{ True} » \land P = P \]
\end{quote}

(\texttt{proof})

\textbf{lemma} \texttt{exp-conj-top-right[simp]}:\newline
\begin{quote}
\[ \text{sound } P \implies P \land «\lambda - \text{ True} » = P \]
\end{quote}

(\texttt{proof})

\textbf{lemma} \texttt{exp-conj-idem[simp]}:\newline
\[ «P» \land «P» = «P» \]

(\texttt{proof})

\textbf{lemma} \texttt{exp-conj-nneg[intro,simp]}:\newline
\[ (\lambda s. 0) \leq P \land Q \]

(\texttt{proof})
3.1. EXPECTATIONS

**Lemma** `exp-conj-sound[intro,simp]`:
- **Assumes**: `s-P: sound P` and `s-Q: sound Q`
- **Shows**: `sound (P && Q)`

**Lemma** `exp-conj-rzero[simp]`:
- **Bounded-by 1 P**: `P => P && (\lambda s. 0) = (\lambda s. 0)`

**Lemma** `exp-conj-1-right[simp]`:
- **Assumes**: `nn: nneg A`
- **Shows**: `A && (\lambda -. 1) = A`

**Lemma** `exp-conj-std-split`:
- **«\lambda s. P s && Q s» = «P» && «Q»**

### 3.1.9 Rules Involving Entailment and Conjunction Together

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

**Lemma** `entails-frame`:
- **Assumes**: `ePR: P ⊢ ⊢ R` and `eQS: Q ⊢ ⊢ S`
- **Shows**: `P && Q ⊢ ⊢ R && S`

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

**Lemma** `pentails-cases`:
- **Assumes**: `PQe: \lambda x. P x ⊢ Q x`
- **And exhaust**: `\lambda s. \exists x. P (x s) s = 1`
- **And framed**: `\lambda x. P x && R ⊢ Q x && S`
- **And sR: sound R and sS: sound S`
- **Shows**: `R ⊢ ⊢ S`

**Lemma** `unitary-bot[iff]`:
- **Unitary (\lambda s. 0::real)**

**Lemma** `unitary-top[iff]`:
- **Unitary (\lambda s. 1::real)**
lemma unitary-embed iff:
unitary « P »
(proof)

lemma unitary-const iff:
[ 0 ≤ c; c ≤ 1 ] ⇒ unitary (λs. c)
(proof)

lemma unitary-mult:
assumes uA: unitary A and uB: unitary B
shows unitary (λs. A * B s)
(proof)

lemma exp-conj-unitary:
[ unitary P; unitary Q ] ⇒ unitary (P && Q)
(proof)

lemma unitary-comp simp:
unitary P ⇒ unitary (P o f)
(proof)

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

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

end

3.2 Expectation Transformers

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

Transformers are functions from expectations to expectations i.e. ('s ⇒ real) ⇒ 's ⇒ real.
The set of healthy transformers is the universe into which we place our semantic interpretation of pGCL programs. In its standard presentation, the healthiness condition for pGCL programs is sublinearity, for demonic programs, and superlinearity for angelic programs. We extract a minimal core property, consisting of monotonicity, feasibility and scaling to form our healthiness property, which holds across all programs. The additional components of sublinearity are broken out separately, and shown later. The two reasons for this are firstly to avoid the effort of establishing sub-(super-)linearity globally, and to allow us to define primitives whose sublinearity,
3.2. EXPECTATION TRANSFORMERS

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 (c) is no higher than 0.5. If it instead takes the edge to state a, we again see a self loop, and thus in theory an infinite trace. In this case, however, every time the automaton reaches state a, with probability 0.5 + 0.3 = 0.8, it transitions to a terminating state. An infinite trace of transitions \( a \rightarrow a \rightarrow \ldots \) thus has probability 0, and the automaton
terminates with probability 1. We formalise such probabilistic termination arguments in Section 4.11.

Having reached $a$, the automaton will proceed to $b$ with probability $0.5 \times \left(\frac{1}{0.5 + 0.3}\right) = 0.625$, 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, $P_{\text{prior}}(c) = 0.3125 \times P_{\text{post}}(b) + 0.1875 \times P_{\text{post}}(c)$. The end result is the same as for nondeterminism: a sublinear transformation (the weights sum to less than one). The two outcomes are thus unified in the semantic interpretation, although as we will establish in Section 4.6, the two have slightly different algebraic properties.

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

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

Figure 3.3: A diverging automaton.
3.2. EXPECTATION TRANSFORMERS

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**

\( le\text{-}trans :: 's\ trans \Rightarrow 's\ trans \Rightarrow \text{bool} \)

**where**

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

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

**definition**

\( le\text{-}utrans :: 's\ trans \Rightarrow 's\ trans \Rightarrow \text{bool} \)

**where**

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

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

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

\( ⟨\text{proof}⟩ \)

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

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

\( ⟨\text{proof}⟩ \)

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

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

\( ⟨\text{proof}⟩ \)

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

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

\( ⟨\text{proof}⟩ \)

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

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

\( ⟨\text{proof}⟩ \)

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

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

\( ⟨\text{proof}⟩ \)

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

\( le\text{-}trans\ x\ x \)

\( ⟨\text{proof}⟩ \)

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

\( le\text{-}utrans\ x\ x \)

\( ⟨\text{proof}⟩ \)
**lemma** le-trans-le-utrans[dest]:
le-trans t u ⇒ le-utrans t u
(proof)

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

Transformer equivalence is induced by comparison:

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

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

**lemma** equiv-transI[intro]:
[ \[ \forall P. sound P ⇒ t P = u P \] ] ⇒ equiv-trans t u
(proof)

**lemma** equiv-utransI[intro]:
[ \[ \forall P. sound P ⇒ t P = u P \] ] ⇒ equiv-utrans t u
(proof)

**lemma** equiv-transD[dest]:
[ equiv-trans t u; sound P ] ⇒ t P = u P
(proof)

**lemma** equiv-utransD[dest]:
[ equiv-utrans t u; unitary P ] ⇒ t P = u P
(proof)

**lemma** equiv-trans-refl[iff]:
equiv-trans t t
(proof)

**lemma** equiv-utrans-refl[iff]:
equiv-utrans t t
(proof)

**lemma** le-trans-antisym:
[ le-trans x y; le-trans y x ] ⇒ equiv-trans x y
(proof)

**lemma** le-utrans-antisym:
3.2. EXPECTATION TRANSFORMERS

\[
[ \text{le-utrans } x \ y; \text{le-utrans } y \ x ] \implies \text{equiv-utrans } x \ y \\
(\text{proof})
\]

**Lemma equiv-trans-comm [ac-simps]:**

\[
\text{equiv-trans } t \ u \iff \text{equiv-trans } u \ t \\
(\text{proof})
\]

**Lemma equiv-utrans-comm [ac-simps]:**

\[
\text{equiv-utrans } t \ u \iff \text{equiv-utrans } u \ t \\
(\text{proof})
\]

**Lemma equiv-imp-le [intro]:**

\[
\text{equiv-trans } t \ u \implies \text{le-trans } t \ u \\
(\text{proof})
\]

**Lemma equiv-uimp-le [intro]:**

\[
\text{equiv-utrans } t \ u \implies \text{le-utrans } t \ u \\
(\text{proof})
\]

**Lemma equiv-imp-le-alt:**

\[
\text{equiv-trans } t \ u \implies \text{le-trans } u \ t \\
(\text{proof})
\]

**Lemma equiv-uimp-le-alt:**

\[
\text{equiv-utrans } t \ u \implies \text{le-utrans } u \ t \\
(\text{proof})
\]

**Lemma le-trans-equiv-rsp [simp]:**

\[
[ \text{equiv-trans } t \ u; \text{le-trans } u \ v ] \implies \text{le-trans } t \ v \\
(\text{proof})
\]

**Lemma le-utrans-equiv-rsp [simp]:**

\[
[ \text{equiv-utrans } t \ u; \text{le-utrans } u \ v ] \implies \text{le-utrans } t \ v \\
(\text{proof})
\]

**Lemma le-trans-equiv-rsp-right [simp]:**

\[
\text{equiv-trans } t \ u \implies \text{le-trans } v \ t \iff \text{le-trans } v \ u \\
(\text{proof})
\]

**Lemma le-utrans-equiv-rsp-right [simp]:**

\[
\text{equiv-utrans } t \ u \implies \text{le-utrans } v \ t \iff \text{le-utrans } v \ u
\]
CHAPTER 3. SEMANTIC STRUCTURES

〈proof〉

lemma le-trans-equiv-trans[trans]:
[ le-trans t u; equiv-trans u v ] \implies le-trans t v
〈proof〉

lemma le-utrans-equiv-utrans[trans]:
[ le-utrans t u; equiv-utrans u v ] \implies le-utrans t v
〈proof〉

lemma equiv-trans-trans[trans]:
assumes xy: equiv-trans x y
and yz: equiv-trans y z
shows equiv-trans x z
〈proof〉

lemma equiv-utrans-trans[trans]:
assumes xy: equiv-utrans x y
and yz: equiv-utrans y z
shows equiv-utrans x z
〈proof〉

lemma equiv-trans-equiv-utrans[dest]:
equiv-trans t u \implies equiv-utrans t u
〈proof〉

3.2.2 Healthy Transformers

Feasibility

definition feasible :: (('a \Rightarrow) \Rightarrow ('a \Rightarrow) \Rightarrow 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]:
[ \land b P. \text{bounded-by } b P; \text{nneg } P ] \implies \text{bounded-by } b (t P);
\land b P. \text{bounded-by } b P; \text{nneg } P \implies \text{nneg } (t P) ] \implies \text{feasible } t
〈proof〉

lemma feasible-boundedD[dest]:
[ feasible t; \text{bounded-by } b P; \text{nneg } P ] \implies \text{bounded-by } b (t P)
〈proof〉

lemma feasible-nnegD[dest]:
3.2. EXPECTATION TRANSFORMERS

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

\text{lemma feasible-sound[dest]:}
\[
\text{feasible } t; \text{ sound } P \implies \text{sound } (t \text{ P })
\]

\text{lemma feasible-pr-0[simp]:}

\text{assumes ft: feasible t}
\text{shows } t (\lambda x. 0) = (\lambda x. 0)

\text{lemma feasible-id:}
\text{feasible } (\lambda x. x)

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

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

\text{lemma feasible-fixes-bot:}
\text{assumes ft: feasible t}
\text{shows } t (\lambda s. 0) = (\lambda s. 0)

\text{lemma feasible-unitaryD[dest]:}
\text{assumes ft: feasible } t \text{ and } uP: \text{unitary } P
\text{shows unitary } (t \text{ P })

\textbf{Monotonicity}

\textbf{definition}
\text{mono-trans :: (('s \Rightarrow \text{real}) \Rightarrow ('s \Rightarrow \text{real})) \Rightarrow \text{bool}}

\textbf{where}
\text{mono-trans } t \equiv \forall P Q. \text{ (sound } P \wedge \text{ sound } Q \wedge 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 \textit{weakest precondition} of \( R \) under \( t \).

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

**Lemma mono-transI[ intro]**:
\[
\begin{align*}
&\forall P. Q. [\text{sound } P; \text{sound } Q; \ P \leq Q] \implies t P \leq t Q \\
\implies &\text{mono-trans } t
\end{align*}
\]

**Lemma mono-transD[ dest]**:
\[
\begin{align*}
&\forall \ P. Q. [\text{sound } P; \text{sound } Q; \ P \leq Q] \implies t P \leq t Q \\
\implies &\text{scaling } t
\end{align*}
\]

**Scaling**

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

**Definition**

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

**Where**

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

The scaling and feasibility properties together allow us to treat transformers as a complete lattice, when operating on bounded expectations. The action of a transformer on such a bounded expectation is completely determined by its action on unitary expectations (those bounded by 1): \( t P s = \text{bound-of } P * t (\lambda s. P s / \text{bound-of } P) s \). 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 scalingI[ intro]**:
\[
\begin{align*}
&\forall P \ c \ x. [\text{sound } P; \ 0 \leq c] \implies c * t P x = t (\lambda x. c * P x) x \\
\implies &\text{scaling } t
\end{align*}
\]

**Lemma scalingD[ dest]**:
\[
\begin{align*}
&\forall \text{ scaling } t; \text{ sound } P; \ 0 \leq c \implies c * t P x = t (\lambda x. c * P x) x
\end{align*}
\]

**Lemma right-scalingD**:  
**Assumes**  
\[s t : \text{ scaling } t\]  
\[s P : \text{ sound } P\]  
\[\text{and } nnc: 0 \leq c\]  
**Shows**  
\[t P s * c = t (\lambda s. P s * c) s\]

**Healthiness**

Healthy transformers are feasible and monotonic, and respect scaling
3.2. EXPECTATION TRANSFORMERS

**definition**

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

**where**

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

**lemma** healthy\text{-}\text{I[intro]}:

\[
\left[ \text{feasible } t; \text{mono-trans } t; \text{scaling } t \right] \Rightarrow \text{healthy } t
\]

**lemmas** healthy\text{-}parts = healthy\text{-}I[\text{OF feasibleI mono-transI scalingI}]

**lemma** healthy-mon\text{O}\text{-D[dest]}:

\[
\text{healthy } t \Rightarrow \text{mono-trans } t
\]

**lemmas** healthy-mon\text{O}\text{-D2} = mono-transD[\text{OF healthy-monO}]

**lemma** healthy-feasible\text{-D[dest]}:

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

**lemma** healthy-scaling\text{-D[dest]}:

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

**lemma** healthy\text{-}bounded\text{-byD[intro]}:

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

**lemma** healthy\text{-}bounded\text{-byD2}:

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

**lemma** healthy\text{-}boundedD[dest,simp]:

\[
\left[ \text{healthy } t; \text{sound } P \right] \Rightarrow \text{bounded } (t P)
\]

**lemma** healthy\text{-}nnegD[dest,simp]:

\[
\left[ \text{healthy } t; \text{sound } P \right] \Rightarrow \text{nneg } (t P)
\]

**lemma** healthy\text{-}nnegD2[dest,simp]:

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

**lemma** healthy\text{-}sound[intro]:

\[
\left[ \text{healthy } t; \text{sound } P \right] \Rightarrow \text{sound } (t P)
\]
CHAPTER 3. SEMANTIC STRUCTURES

**Lemma** healthy-unitary[intro]:
\[
\text{healthy } t; \text{ unitary } P \implies \text{unitary } (t \ P)
\]
(proof)

**Lemma** healthy-id[simp,intro]:
healthy id
(proof)

**Lemmas** healthy-fixes-bot = feasible-fixes-bot[OF healthy-feasibleD]

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

**Lemma** le-trans-bot[intro,simp]:
healthy t \implies le-trans \((\lambda P \ s. \ 0)\) t
(proof)

**Lemma** le-trans-top[intro,simp]:
healthy t \implies le-trans t \((\lambda P \ s. \ \text{bound-of } P)\)
(proof)

**Lemma** healthy-pr-bot[simp]:
healthy t \implies t \((\lambda s. \ 0) = (\lambda s. \ 0)\)
(proof)

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

**Lemma** healthy-equivI:
fixes t::\((s \Rightarrow \text{real}) \Rightarrow s \Rightarrow \text{real}\) and u
assumes equiv: equiv-trans t u
and healthy: healthy t
shows healthy u
(proof)

**Lemma** healthy-equiv:
equiv-trans t u \implies healthy t \iff healthy u
(proof)

**Lemma** healthy-scale:
fixes t::\((s \Rightarrow \text{real}) \Rightarrow s \Rightarrow \text{real}\)
assumes ht: healthy t and nc: \(0 \leq c\) and bc: \(c \leq 1\)
shows healthy \((\lambda P \ s. \ c \ast t \ P \ s)\)
(proof)

**Lemma** healthy-top iff:
healthy \((\lambda P \ s. \ \text{bound-of } P)\)
(proof)

**Lemma** healthy-bot iff:
healthy \((\lambda P \ s. \ 0)\)
(proof)

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

\textbf{definition}

\begin{align*}
nearly-healthy :: (('s \Rightarrow \text{real}) \Rightarrow ('s \Rightarrow \text{real})) &\Rightarrow \text{bool} \\
\text{where} &
nearly-healthy t \leftrightarrow (\forall P. \text{unitary } P \rightarrow \text{unitary } (t P)) \land \\
& (\forall P Q. \text{unitary } P \rightarrow \text{unitary } Q \rightarrow P \vdash Q \rightarrow t P \vdash t Q)
\end{align*}

\textbf{lemma nearly-healthyI[\text{intro}]}:
\begin{align*}
[ & \bigwedge P. \text{unitary } P \implies \text{unitary } (t P); \\
& \bigwedge P Q. \big[ \text{unitary } P; \text{unitary } Q; P \vdash Q \big] \implies t P \vdash t Q \big] \implies nearly-healthy t
\end{align*}

\textbf{lemma nearly-healthy-monoD[\text{dest}]}:
\begin{align*}
[ & nearly-healthy t; P \vdash Q; \text{unitary } P; \text{unitary } Q \big] \implies t P \vdash t Q
\end{align*}

\textbf{lemma nearly-healthy-unitaryD[\text{dest}]}:
\begin{align*}
[ & nearly-healthy t; \text{unitary } P \big] \implies \text{unitary } (t P)
\end{align*}

\textbf{lemma healthy-nearly-healthy[\text{dest}]}:
\begin{align*}
\text{assumes } kt: \text{healthy } t \\
\text{shows } nearly-healthy t
\end{align*}

\textbf{lemmas nearly-healthy-id[iff]} =
\begin{align*}
\text{healthy-nearly-healthy}[\text{OF healthy-id, unfolded id-def}]
\end{align*}

### 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 \textit{quasi-linear} combination of sound expectations is greater than the same combination applied to the transformation of the expectations themselves. The term \( x \ominus y \) represents \textit{truncated subtraction} i.e. \( \max (x - y) (0::'a) \) (see Section 4.13.1).

\textbf{definition sublinear ::}
\begin{align*}
(('s \Rightarrow \text{real}) \Rightarrow ('s \Rightarrow \text{real})) &\Rightarrow \text{bool} \\
\text{where} &
sublinear t \leftrightarrow (\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)
\end{align*}

\textbf{lemma sublinearI[\text{intro}]}:
CHAPTER 3. SEMANTIC STRUCTURES

Figure 3.4: A graphical depiction of sub-additivity as convexity.

\[ P \]
\[ tP \]
\[ Q = tP \cap uP \]
\[ Q(x) \]
\[ x \]
\[ y \]
\[ Q(y) \]
\[ Q(x+Q(y)) \]
\[ Q(x+Q(y)/2) \]

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
3.2. EXPECTATION TRANSFORMERS

discrete and multi-dimensional, but on the reals we have a convenient geometrical intuition. The lines $tP$ and $uP$ represent the effect of two healthy transformers (again affine). Neither monotonicity nor scaling are represented, but both are feasible: Both lines are bounded above by the greatest value of $P$.

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

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

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

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

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

Can be rewritten, using scaling, to:

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

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

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

**lemma sub-addI**: \[ \left[ \bigwedge P Q s. \right. \left[ \text{sound } P; \text{sound } Q \right] \Rightarrow t P s + t Q s \leq t (\lambda s’. P s’ + Q s’) s \] \Rightarrow sub-add t \]

(proof)

**lemma sub-addI2**: \[ \left[ \bigwedge P Q. \left[ \\text{sound } P; \text{sound } Q \right] \Rightarrow \lambda s. t P s + t Q s \vdash t (\lambda s. P s + Q s) \right] \Rightarrow sub-add t \]

(proof)

**lemma sub-addD**: \[ \left[ \text{sub-add } t; \text{sound } P; \text{sound } Q \right] \Rightarrow t P s + t Q s \leq t (\lambda s’. P s’ + Q s’) s \]

(proof)
lemma equiv-sub-add:
fixes t::('s ⇒ real) ⇒ 's ⇒ real
assumes eq: equiv-trans t u
and sa: sub-add t
shows sub-add u
⟨proof⟩

Sublinearity and feasibility imply sub-additivity.

lemma sublinear-subadd:
fixes t::('s ⇒ real) ⇒ 's ⇒ real
assumes slt: sublinear t
and ft: feasible t
shows sub-add t
⟨proof⟩

A few properties following from sub-additivity:

lemma standard-negate:
assumes ht: healthy t
and sat: sub-add t
shows t «P» s + t «¬P» s ≤ 1
⟨proof⟩

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 (∑y∈S. t (P y) x) ≤ t (∑y∈S. P y x)
⟨proof⟩

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 (∑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⟩

Sub-distributivity

definition sub-distrib ::
(('s ⇒ real) ⇒ ('s ⇒ real)) ⇒ bool
where
sub-distrib t ←→ (∀P s. sound P → t P s ⊕ 1 ≤ t (λs'. P s' ⊕ 1) s)

lemma sub-distribI[intro]:
[ ∀P s. sound P → t P s ⊕ 1 ≤ t (λs'. P s' ⊕ 1) s ] → sub-distrib t
⟨proof⟩
3.2. EXPECTATION TRANSFORMERS

lemma `sub-distrib12`:
\[
[ \forall P. \text{sound } P \implies \lambda s. t \ P \odot \ 1 \vdash t (\lambda s. P \ s \odot 1) ] \implies \text{sub-distrib } t
\]
(proof)

lemma `sub-distribD[dest]`:
\[
[ \text{sub-distrib } t; \text{sound } P ] \implies t \ P \odot 1 \leq t (\lambda s'. P \ s' \odot 1) \ s
\]
(proof)

lemma `equiv-sub-distrib`:
fixes \( t \) :: \((\sigma \Rightarrow \text{real}) \Rightarrow \sigma \Rightarrow \text{real}\)
assumes \( \text{eq} \): \( \text{equiv-trans } t \ u \)
and \( \text{sd} \): \( \text{sub-distrib } t \)
shows `sub-distrib u`
(proof)

Sublinearity implies sub-distributivity:

lemma `sublinear-sub-distrib`:
fixes \( t \) :: \((\sigma \Rightarrow \text{real}) \Rightarrow \sigma \Rightarrow \text{real}\)
assumes \( \text{slt} \): \( \text{sublinear } t \)
shows `sub-distrib t`
(proof)

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

lemma `sd-sa-sublinear`:
fixes \( t \) :: \((\sigma \Rightarrow \text{real}) \Rightarrow \sigma \Rightarrow \text{real}\)
assumes \( \text{sd} \): \( \text{sub-distrib } t \) and \( \text{sat} \): \( \text{sub-add } t \) and \( \text{ht} \): \( \text{healthy } t \)
shows `sublinear t`
(proof)

Sub-conjunctivity

definition `sub-conj` :: \((\sigma \Rightarrow \text{real}) \Rightarrow \sigma \Rightarrow \text{real}) \Rightarrow \text{bool}\)
where
\[
\text{sub-conj } t \equiv \forall P \ Q. (\text{sound } P \land \text{sound } Q) \implies t \ P \land t \ Q \vdash t (P \land Q)
\]

lemma `sub-conjI[intro]`:
\[
[ \forall P \ Q. [ \text{sound } P; \text{sound } Q ] \implies t \ P \land t \ Q \vdash t (P \land Q) ] \implies \text{sub-conj } t
\]
(proof)

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

lemma `sub-conj-wp-twice`:
\textbf{CHAPTER 3. SEMANTIC STRUCTURES}

\texttt{fixes } f :: \texttt{(}\texttt{'}s \Rightarrow \texttt{'}s \Rightarrow \texttt{real}) \\
\texttt{assumes } \forall s. \texttt{sub-conj } (f\ s) \\
\texttt{shows } \texttt{sub-conj } (\lambda P\ s.\ f\ s\ P\ s) \\
\langle \text{proof} \rangle

Sublinearity implies sub-conjunctivity:

\textbf{lemma} \texttt{sublinear-sub-conj}:
\texttt{fixes } t :: (\texttt{'}s \Rightarrow \texttt{real}) \Rightarrow \texttt{'}s \Rightarrow \texttt{real} \\
\texttt{assumes } \texttt{slt: sublinear } t \\
\texttt{shows } \texttt{sub-conj } t \\
\langle \text{proof} \rangle

\textbf{Sublinearity under equivalence}

Sublinearity is preserved by equivalence.

\textbf{lemma} \texttt{equiv-sublinear}:
\texttt{[ [ equiv-trans } t\ u;\ \texttt{sublinear } t;\ \texttt{healthy } t \texttt{ ] } \Rightarrow \texttt{sublinear } u \\
\langle \text{proof} \rangle

\textbf{3.2.4 Determinism}

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

\textbf{Additivity}

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

\textbf{definition} \texttt{additive} :: (\texttt{'}a \Rightarrow \texttt{real}) \Rightarrow \texttt{'}a \Rightarrow \texttt{real} \Rightarrow \texttt{bool} \\
\texttt{where} \\
\texttt{additive } t \equiv \forall P\ Q.\ (\texttt{sound } P \land \texttt{sound } Q) \longrightarrow \texttt{t } (\lambda s.\ P\ s + Q\ s) = (\lambda s.\ t\ P\ s + t\ Q\ s)

\textbf{lemma} \texttt{additiveD}:
\texttt{[ additive } t;\ \texttt{sound } P;\ \texttt{sound } Q \texttt{ ] } \Rightarrow \texttt{t } (\lambda s.\ P\ s + Q\ s) = (\lambda s.\ t\ P\ s + t\ Q\ s) \\
\langle \text{proof} \rangle

\textbf{lemma} \texttt{additiveI[intro]}:
\texttt{[ } \land\ P\ Q\ s.\ [\ \texttt{sound } P;\ \texttt{sound } Q \texttt{ ] } \Rightarrow \texttt{t } (\lambda s.\ P\ s + Q\ s)\ s = t\ P\ s + t\ Q\ s \texttt{ ] } \Rightarrow \texttt{additive } t \\
\langle \text{proof} \rangle

Additivity is strictly stronger than sub-additivity.

\textbf{lemma} \texttt{additive-sub-add}:
\texttt{additive } t \Rightarrow \texttt{sub-add } t
3.2. EXPECTATION TRANSFORMERS

The additivity property extends to finite summation.

**Lemma** additive-sum:
- **Fines** $S :: 's 	ext{ set}$
- **Assumes** additive: additive $t$
  - and healthy: healthy $t$
  - and finite: finite $S$
  - and $s P z$: $\forall z. \text{sound} (P z)$
- **Shows** $t (\lambda x. \sum_{y \in S} P y x) = (\lambda x. \sum_{y \in S} t (P y) x)$

**Proof**

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:
- **Fines** $t :: (\forall 's :: \text{finite} \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real}$
- **Assumes** additive: additive $t$
  - and $ht$: healthy $t$
  - and $s P$: sound $P$
- **Shows** $t P x = (\sum_{y \in \text{UNIV}} P y \cdot t (\lambda z. z = y) x)$

**Proof**

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

**Lemma** additive-guard-split:
- **Fines** $t :: (\forall 's :: \text{finite} \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real}$
- **Assumes** additive: additive $t$
  - and $ht$: healthy $t$
  - and $s P$: sound $P$
- **Shows** $t P x = (\sum_{y \in \{s. G s\}} P y \cdot t (\lambda z. z = y) x) + (\sum_{y \in \{s. \neg G s\}} P y \cdot t (\lambda z. z = y) x)$

**Proof**

**Maximality**

**Definition**

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

**Where**

$maximal t \equiv \forall c. 0 \leq c \rightarrow t (\lambda. c) = (\lambda. c)$

**Lemma** maximalI[intro]:

$\forall c. 0 \leq c \Rightarrow t (\lambda. c) = (\lambda. c) \implies maximal t$

**Proof**

**Lemma** maximalD[dest]:

$\forall maximal t; 0 \leq c \implies t (\lambda. c) = (\lambda. c)$
A transformer that is both additive and maximal is deterministic:

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

**where**

\[ \text{determ } t \equiv \text{additive } t \land \text{maximal } t \]

**lemma** `determI[intro]`:

\[
\begin{array}{l}
[ \text{additive } t; \text{maximal } t ] \implies \text{determ } t
\end{array}
\]

**proof**

**lemma** `determ-additiveD[intro]`:

\[
\text{determ } t \implies \text{additive } t
\]

**proof**

**lemma** `determ-maximalD[intro]`:

\[
\text{determ } t \implies \text{maximal } t
\]

**proof**

For a fully-deterministic transformer, a transformed standard expectation, and its transformed negation are complementary.

**lemma** `determ-negate`:

\[
\text{assumes } \text{determ: determ } t
\]

\[
\text{shows } t \llbracket P \rrbracket s + t \llbracket \neg P \rrbracket s = 1
\]

**proof**

### 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`:

\[
\text{assumes } \begin{array}{l}
wp1: P \vdash t R \\
and wp2: Q \vdash t S \\
and sc: \text{sub-conj } t \\
and sR: \text{sound } R \\
and sS: \text{sound } S
\end{array}
\]

\[
\text{shows } P \land Q \vdash t (R \land S)
\]

**proof**

These allow mismatched results to be composed

**lemma** `entails-strengthen-post`:

\[
\begin{array}{l}
[ P \vdash t Q; \text{healthy } t; \text{sound } R; Q \vdash R; \text{sound } Q ] \implies P \vdash t R
\end{array}
\]

**proof**
3.2. EXPECTATION TRANSFORMERS

lemma entails-weaken-pre:
\[ [ Q \vdash t R; P \vdash Q ] \implies P \vdash t R \]
(proof)

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

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:
\begin{itemize}
  \item assumes \( h: \text{healthy } t \) and \( wP: \bigwedge s. P s \implies 1 \leq t \langle Q \rangle s \)
  \item shows \( \langle P \rangle \vdash t \langle Q \rangle \)
\end{itemize}
(proof)

The other direction works too.

lemma fold-premise:
\begin{itemize}
  \item assumes \( ht: \text{healthy } t \) and \( wp: \langle P \rangle \vdash t \langle Q \rangle \)
  \item shows \( \forall s. P s \implies 1 \leq t \langle Q \rangle s \)
\end{itemize}
(proof)

Predicate conjunction behaves as expected:

lemma conj-post:
\[ [ P \vdash t \langle \lambda s. Q s \land R s \rangle; \text{healthy } t ] \implies P \vdash t \langle Q \rangle \]
(proof)

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

lemma entails-pconj-assumption:
\begin{itemize}
  \item assumes \( f: \text{feasible } t \) and \( wP: \bigwedge s. P s \implies Q s \leq t R s \)
  \item and \( uQ: \text{unitary } Q \) and \( uR: \text{unitary } R \)
  \item shows \( \langle P \rangle \land \langle Q \rangle \vdash t R \)
\end{itemize}
(proof)

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 λ-. 0::'b), but as we do not insist on any global bound on expectations (and work directly in HOL’s real type, rather than extending it with a point at infinity), there is no top element. We solve the problem by defining the least (greatest) fixed point, restricted to an internally-bounded set, allowing us to substitute this bound for the top element. This works as long as the set contains at least one fixed point, which appears as an extra assumption in all the theorems.

This also works semantically, thanks to the definition of healthiness. Given a healthy transformer parameterised by some sound expectation: t. Imagine that we wish to find the least fixed point of t P. In practice, t is generally doubly healthy, that is ∀ P. sound P → healthy (t P) and ∀ Q. sound Q → healthy (λP. t P Q). Thus by feasibility, t P Q must be bounded by bound-of P. Thus, as by definition x ≤ t P x for any fixed point, all must lie in the set of sound expectations bounded above by λ-. bound-of P.

definition Inf-exp :: 's expect set ⇒ 's expect
where Inf-exp S = (λs. Inf {f s | f ∈ S})

lemma Inf-exp-lower:
[ P ∈ S; ∀ P ∈ S. nneg P ] ⇒ Inf-exp S ≤ P
⟨proof⟩

lemma Inf-exp-greatest:
[ S ≠ {}; ∀ P ∈ S. Q ≤ P ] ⇒ Q ≤ Inf-exp S
⟨proof⟩

definition Sup-exp :: 's expect set ⇒ 's expect
where Sup-exp S = (if S = {} then λs. 0 else (λs. Sup {f s | f ∈ S}))

lemma Sup-exp-upper:
[ P ∈ S; ∀ P ∈ S. bounded-by b P ] ⇒ P ≤ Sup-exp S
⟨proof⟩

lemma Sup-exp-least:
[ ∀ P ∈ S. P ≤ Q; nneg Q ] ⇒ Sup-exp S ≤ Q
3.3. **INDUCTION**

⟨proof⟩

**lemma** `Sup-exp-sound`:
- **assumes** `sS`: \( \forall P. P \in S \Rightarrow \text{sound } P \)
- **and** `bS`: \( \forall P. P \in S \Rightarrow \text{bounded-by } b P \)
- **shows** `\text{sound } (\text{Sup-exp } S)`

⟨proof⟩

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

**lemma** `lfp-exp-lowerbound`:
- \( \left[ t P \leq P; \text{sound } P \right] \Rightarrow \text{lfp-exp } t \leq P \)

⟨proof⟩

**lemma** `lfp-exp-greatest`:
- \( \left[ \left[ \forall P. \left[ t P \leq P; \text{sound } P \right] \Rightarrow Q \leq P; \text{sound } Q; t R \vdash R; \text{sound } R \right] \Rightarrow Q \leq \right] \text{lfp-exp } t \)

⟨proof⟩

**lemma** `feasible-lfp-exp-sound`:
- \( \text{feasible } t \Rightarrow \text{sound } (\text{lfp-exp } t) \)

⟨proof⟩

**lemma** `lfp-exp-sound`:
- **assumes** `fR`: \( t R \vdash R \)  
- **and** `sR`: \( \text{sound } R \)
- **shows** `\text{sound } (\text{lfp-exp } t)`

⟨proof⟩

**lemma** `lfp-exp-bound`:
- \( \left( \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) \right) \Rightarrow \text{bounded-by } t (\text{lfp-exp } t) \)

⟨proof⟩

**lemma** `lfp-exp-unitary`:
- \( \left( \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) \right) \Rightarrow \text{unitary } (\text{lfp-exp } t) \)

⟨proof⟩

**lemma** `lfp-exp-lemma2`:
- **fixes** `t`::'s trans  
- **assumes** `st`: \( \forall P. \text{sound } P \Rightarrow \text{sound } (t P) \)  
- **and** `mt`: \( \text{mono-trans } t \)  
- **and** `fR`: \( t R \vdash R \)  
- **and** `sR`: \( \text{sound } R \)
- **shows** `t (\text{lfp-exp } t) \leq \text{lfp-exp } t`

⟨proof⟩

**lemma** `lfp-exp-lemma3`:
- **assumes** `st`: \( \forall P. \text{sound } P \Rightarrow \text{sound } (t P) \)  
- **and** `mt`: \( \text{mono-trans } t \)  
- **and** `fR`: \( t R \vdash R \)  
- **and** `sR`: \( \text{sound } R \)
shows \( \text{lfp-exp } t \leq t \left( \text{lfp-exp } t \right) \)

\( \langle \text{proof} \rangle \)

**lemma lfp-exp-unfold:**
- assumes \( \text{nt: } \forall P. \text{ sound } P \implies \text{ sound } (t \ P) \)
- and \( \text{mt: mono-trans } t \)
- and \( \text{flt: } t \ R \vdash R \text{ and } \text{sR: sound } R \)
- shows \( \text{lfp-exp } t = t \left( \text{lfp-exp } t \right) \)

\( \langle \text{proof} \rangle \)

**definition gfp-exp :: 's trans ⇒ 's expect**

where

\( \text{gfp-exp } t = \sup\text{-exp} \left\{ P. \text{ unitary } P \land P \leq t \ P \right\} \)

**lemma gfp-exp-upperbound:**
- \( \left[ P \leq t \ P; \text{ unitary } P \right] \implies P \leq \text{gfp-exp } t \)

\( \langle \text{proof} \rangle \)

**lemma gfp-exp-least:**
- \( \left[ \forall P. \left[ P \leq t \ P; \text{ unitary } P \right] \implies P \leq Q; \text{ unitary } Q \right] \implies \text{gfp-exp } t \leq Q \)

\( \langle \text{proof} \rangle \)

**lemma gfp-exp-bound:**
- \( (\forall P. \text{ unitary } P \implies \text{ unitary } (t \ P)) \implies \text{ bounded-by } 1 \text{ (gfp-exp } t) \)

\( \langle \text{proof} \rangle \)

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

\( \langle \text{proof} \rangle \)

**lemma gfp-exp-unitary:**
- \( (\forall P. \text{ unitary } P \implies \text{ unitary } (t \ P)) \implies \text{ unitary } (\text{gfp-exp } t) \)

\( \langle \text{proof} \rangle \)

**lemma gfp-exp-lemma2:**
- assumes \( \text{ft: } \forall P. \text{ unitary } P \implies \text{ unitary } (t \ P) \)
- and \( \text{mt: } \forall P. \forall Q. \left[ \text{ unitary } P; \text{ unitary } Q; P \vdash Q \right] \implies t \ P \vdash t \ Q \)
- shows \( \text{gfp-exp } t \leq t \left( \text{gfp-exp } t \right) \)

\( \langle \text{proof} \rangle \)

**lemma gfp-exp-lemma3:**
- assumes \( \text{ft: } \forall P. \text{ unitary } P \implies \text{ unitary } (t \ P) \)
- and \( \text{mt: } \forall P. \forall Q. \left[ \text{ unitary } P; \text{ unitary } Q; P \vdash Q \right] \implies t \ P \vdash t \ Q \)
- shows \( t \left( \text{gfp-exp } t \right) \leq \text{gfp-exp } t \)

\( \langle \text{proof} \rangle \)

**lemma gfp-exp-unfold:**
- \( (\forall P. \text{ unitary } P \implies \text{ unitary } (t \ P)) \implies (\forall P. \forall Q. \left[ \text{ unitary } P; \text{ unitary } Q; P \vdash Q \right] \implies t \ P \vdash t \ Q) \implies \text{gfp-exp } t = t \left( \text{gfp-exp } t \right) \)

\( \langle \text{proof} \rangle \)
3.3. **INDUCTION**

3.3.2 **The Lattice of Transformers**

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

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

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

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

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

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

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

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

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

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

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

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

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

\[\text{nneg } P; \text{bounded-by } b P \left] \Rightarrow t P \vdash \text{Sup-trans } S P\right.
\]

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

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

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

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

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

\[\text{nneg } P; \text{bounded-by } b P; \forall P. \left[ \text{nneg } P; \text{bounded-by } b P \right] \Rightarrow \text{nneg } (u P) \left] \Rightarrow \text{Sup-trans } S P \vdash u P\right.
\]

**lemma** \( \text{feasible-Sup-trans} : \)
CHAPTER 3. SEMANTIC STRUCTURES

fixes $S$::’s trans set
assumes $fS$:: $\forall t \in S. \text{feasible } t$
shows feasible (Sup-trans $S$)
(proof)

definition lfp-trans :: (’s trans $\Rightarrow$ ’s trans) $\Rightarrow$ ’s trans
where lfp-trans $T = \text{Inf-trans } \{ t. (\forall P. \text{sound } P \Rightarrow \text{sound } (t P)) \land \text{le-trans } (T t) t \}$

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

lemma lfp-trans-greatest:
[ $\bigwedge P. \text{sound } P \Rightarrow \text{sound } (t P)$ ] $\Rightarrow$ le-trans $u$ $t$
(proof)

lemma lfp-trans-sound:
fixes $P Q$::’s expect
assumes $sP$:: sound $P$
and $fv$: le-trans (T $v$) $v$
and $sv$: $\bigwedge P. \text{sound } P \Rightarrow \text{sound } (v P)$
shows sound (lfp-trans $T$ $P$)
(proof)

lemma lfp-trans-unitary:
fixes $P Q$::’s expect
assumes $uP$:: unitary $P$
and $fv$: le-trans (T $v$) $v$
and $sv$: $\bigwedge P. \text{sound } P \Rightarrow \text{sound } (v P)$
and $fT$: le-trans (T ($\lambda P s. \text{bound-of } P$)) ($\lambda P s. \text{bound-of } P$)
shows unitary (lfp-trans $T$ $P$)
(proof)

lemma lfp-trans-lemma2:
fixes $v$::’s trans
assumes mono: $\bigwedge t u. [ \text{le-trans } t u; \bigwedge P. \text{sound } P \Rightarrow \text{sound } (t P); \bigwedge P. \text{sound } P \Rightarrow \text{sound } (u P) ] \Rightarrow \text{le-trans } (T t) (T u)$
and $nT$: $\bigwedge t P. [ \bigwedge Q. \text{sound } Q \Rightarrow \text{sound } (t Q); \text{sound } P ] \Rightarrow \text{sound } (T t P)$
and $fv$: le-trans (T $v$) $v$
and $sv$: $\bigwedge P. \text{sound } P \Rightarrow \text{sound } (v P)$
shows le-trans (T (lfp-trans $T$)) (lfp-trans $T$)
(proof)

lemma lfp-trans-lemma3:
fixes $v$::’s trans
assumes mono: \( \land t u. [ \land \text{le-trans } \text{t } \text{u}; \land \text{P. sound } \text{P } \implies \text{sound } (\text{t } \text{P}); \land \text{P. sound } \text{P } \implies \text{sound } (\text{u } \text{P}) ] \implies \text{le-trans } (\text{T } \text{t}) (\text{T } \text{u}) \)
and sT: \( \land t P. [ \land Q. \text{sound } \text{Q } \implies \text{sound } (\text{t } \text{Q}); \land \text{P. sound } \text{P } \implies \text{sound } (\text{u } \text{P}) ] \implies \text{le-trans } (\text{T } \text{t } \text{P}) \)
and fv: le-trans (T v) v
and sv: \( \land P. \text{sound } \text{P } \implies \text{sound } (\text{v } \text{P}) \)
shows le-trans (lfp-trans T) (T (lfp-trans T))

\langle proof \rangle

lemma lfp-trans-unfold:
fixes P::'s expect
assumes mono: \( \land t u. [ \land \text{le-trans } \text{t } \text{u}; \land \text{P. sound } \text{P } \implies \text{sound } (\text{t } \text{P}); \land \text{P. sound } \text{P } \implies \text{sound } (\text{u } \text{P}) ] \implies \text{le-trans } (\text{T } \text{t}) (\text{T } \text{u}) \)
and sT: \( \land t P. [ \land Q. \text{sound } \text{Q } \implies \text{sound } (\text{t } \text{Q}); \land \text{P. sound } \text{P } \implies \text{sound } (\text{u } \text{P}) ] \implies \text{le-trans } (\text{T } \text{t } \text{P}) \)
and fv: le-trans (T v) v
and sv: \( \land P. \text{sound } \text{P } \implies \text{sound } (\text{v } \text{P}) \)
shows equiv-trans (lfp-trans T) (T (lfp-trans T))
\langle proof \rangle

definition gfp-trans :: ('s trans ⇒ 's trans) ⇒ 's trans
where gfp-trans T = Sup-trans { t. (\forall P. \text{unitary } \text{P } \implies \text{unitary } (\text{t } \text{P})) \land \text{le-utrans } \text{t } (\text{T } \text{t})}]

lemma gfp-trans-upperbound:
[ \land \text{le-utrans } \text{t } (\text{T } \text{t}); \land \text{P. unitary } \text{P } \implies \text{unitary } (\text{t } \text{P}) ] \implies \text{le-utrans } \text{t } (\text{gfp-trans } T)
\langle proof \rangle

lemma gfp-trans-least:
[ \land t. [ \land \text{le-utrans } \text{t } (\text{T } \text{t}); \land \text{P. unitary } \text{P } \implies \text{unitary } (\text{t } \text{P}) ] \implies \text{le-utrans } \text{t } \text{u}; \land \text{P. unitary } \text{P } \implies \text{unitary } (\text{u } \text{P}) ] \implies \text{le-utrans } (\text{gfp-trans } T) \text{ u}
\langle proof \rangle

lemma gfp-trans-unitary:
fixes P::'s expect
assumes uP: unitary P
shows unitary (gfp-trans T P)
\langle proof \rangle

lemma gfp-trans-lemma2:
assumes mono: \( \land t u. [ \land \text{le-utrans } \text{t } \text{u}; \land \text{P. unitary } \text{P } \implies \text{unitary } (\text{t } \text{P}); \land \text{P. unitary } \text{P } \implies \text{unitary } (\text{u } \text{P}) ] \implies \text{le-utrans } (\text{T } \text{t}) (\text{T } \text{u}) \)
and hT: \( \land t P. [ \land Q. \text{unitary } \text{Q } \implies \text{unitary } (\text{t } \text{Q}); \land \text{P. unitary } \text{P } \implies \text{unitary } (\text{T } \text{t } \text{P}) ] \implies \text{unitary } (\text{T } \text{t } \text{P}) \)
shows le-utrans (gfp-trans T) (T (gfp-trans T))
\langle proof \rangle
CHAPTER 3. SEMANTIC STRUCTURES

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.

\[ \text{lemma } \text{gfp-trans-lemma3:} \]
\[ \begin{align*}
& \text{assumes mono: } \forall t, u. \ [ \ \text{le-utrans } t u; \ \forall P. \ \text{unitary } P \Rightarrow \text{unitary } (u P) ; \\
& \quad \forall P. \ \text{unitary } P \Rightarrow \text{unitary } (u P) ] \Rightarrow \text{le-utrans } (T t) (T u) \\
& \text{and } hT: \ [ \forall t P. [ \forall Q. \ \text{unitary } Q \Rightarrow \text{unitary } (t Q); \ \text{unitary } P ] \Rightarrow \text{unitary } (T t P) \\
& \text{shows le-utrans } (T (\text{gfp-trans } T)) (\text{gfp-trans } T) \\
& \quad \text{⟨proof⟩} \\
\end{align*} \]

\[ \text{lemma } \text{gfp-trans-unfold:} \]
\[ \begin{align*}
& \text{assumes mono: } \forall t, u. \ [ \ \text{le-utrans } t u; \ \forall P. \ \text{unitary } P \Rightarrow \text{unitary } (t P) ; \\
& \quad \forall P. \ \text{unitary } P \Rightarrow \text{unitary } (u P) ] \Rightarrow \text{le-utrans } (T t) (T u) \\
& \text{and } hT: \ [ \forall t P. [ \forall Q. \ \text{unitary } Q \Rightarrow \text{unitary } (t Q); \ \text{unitary } P ] \Rightarrow \text{unitary } (T t P) \\
& \text{shows equiv-utrans } (\text{gfp-trans } T) (T (\text{gfp-trans } T)) \\
& \quad \text{⟨proof⟩} \\
\end{align*} \]

\[ \text{3.3.3 Tail Recursion} \]

\[ \begin{align*}
& \text{The least (greatest) fixed point of a tail-recursive expression on transformers is equivalent (given appropriate side conditions) to the least (greatest) fixed point on expectations.} \\
& \text{lemma } \text{gfp-pulldown:} \quad \text{⟨proof⟩} \\
& \quad \text{⟨proof⟩} \\
\end{align*} \]
3.3. INDUCTION

shows \( \text{lfp-trans } T \ P = \text{lfp-exp } (t \ P) \) \((\text{is } X \ P = \text{?Y } P)\)

\[\text{⟨proof}\]\n
definition \( \text{Inf-utrans} :: 's \text{ trans set } \Rightarrow 's \text{ trans} \)
where \( \text{Inf-utrans } S = (\text{if } S = \{\} \text{ then } \lambda \ P \ s \ . \ 1 \text{ else } \text{Inf-trans } S) \)

lemma \( \text{Inf-utrans-lower}: \)
\[\text{⟨proof}\]\n
lemma \( \text{Inf-utrans-greatest}: \)
\[\text{⟨proof}\]\nend
Chapter 4

The pGCL Language

4.1 A Shallow Embedding of pGCL in HOL

theory Embedding imports Misc Induction begin

4.1.1 Core Primitives and Syntax

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

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

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

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

Skip does nothing at all.

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

Apply lifts a state transformer into the space of programs.

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

Seq is sequential composition.

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

PC is probabilistic choice between programs.

definition PC :: 's prog ⇒ ('s ⇒ real) ⇒ 's prog ⇒ 's prog
where PC a P b ≡ λab Q s. P s * a ab Q s + (1 - P s) * b ab Q s
CHAPTER 4. THE PGCL LANGUAGE

$DC$ is demonic choice between programs.

**Definition** $DC :: s prog \Rightarrow s prog \Rightarrow s prog (- \prod - [58,57] 57)$

**Where** $DC a b \equiv \lambda ab Q s \cdot \min (a ab Q s) (b ab Q s)$

$AC$ is angelic choice between programs.

**Definition** $AC :: s prog \Rightarrow s prog \Rightarrow s prog (- \bigcup - [58,57] 57)$

**Where** $AC a b \equiv \lambda ab Q s \cdot \max (a ab Q s) (b ab Q s)$

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

**Definition** $Embed :: s trans \Rightarrow s prog$

**Where** $Embed t = (\lambda ab. t)$

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

**Definition** $Mu :: (s prog \Rightarrow s prog) \Rightarrow s prog$

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

*repeat* expresses finite repetition

**Primrec**

$repeat :: \text{nat} \Rightarrow \text{'a prog} \Rightarrow \text{'a prog}$

**Where**

$repeat 0 p = \text{Skip} \mid$

$repeat (Suc n) p = p \mid; \; repeat n p$

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

**Definition** $SetDC :: (a \Rightarrow s prog) \Rightarrow (s \Rightarrow \text{'a set}) \Rightarrow s prog$

**Where** $SetDC f S \equiv \lambda ab P s \cdot \text{Inf} ((\lambda a. f a ab P s) \cdot S s)$

**Syntax** $\prod x \in S. \; p == \text{CONST SetDC (\%a.} \; p) \; S$

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

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

**Definition**

$SetPC :: (a \Rightarrow s prog) \Rightarrow (s \Rightarrow a \Rightarrow \text{real}) \Rightarrow s prog$

**Where**

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

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

**Definition**

$Bind :: (s \Rightarrow 'a) \Rightarrow (a \Rightarrow \text{'s prog}) \Rightarrow \text{'s prog}$
4.1. A SHALLOW EMBEDDING OF PGCL IN HOL

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⟩

**syntax**
-SetPC :: ident => ('s => 'a => real) => 's prog
  (choose - at - [66,66])
  ⟨ML⟩

**syntax**
-set-dc :: ident => ('s => 'a set) => 's prog (- :∈ - [66,66])
  ⟨ML⟩

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 (%x. CONST UNIV)

**definition** wp :: 's prog => 's trans
**where**
wp pr ≡ pr True
definition
  \text{wlp} :: \mathfrak{p} \text{ prog} \Rightarrow \mathfrak{p} \text{ trans}
where
  \text{wlp} \text{ pr} \equiv \text{pr} \text{ False}

If-Then-Else as a degenerate probabilistic choice.

abbreviation\,(\text{input})
  \text{if-then-else} :: [\mathfrak{s} \Rightarrow \text{bool}, \mathfrak{s} \text{ prog}, \mathfrak{s} \text{ prog}] \Rightarrow \mathfrak{s} \text{ prog}
  (\text{If - Then - Else - 58})
where
  If \, P \, \text{ Then } \, a \, \text{ Else } \, b = a \, \mu \, P \, \oplus \, b

Syntax for loops

abbreviation
  \text{do-while} :: [\mathfrak{s} \Rightarrow \text{bool}, \mathfrak{s} \text{ prog}] \Rightarrow \mathfrak{s} \text{ prog}
  (\text{do - \cdots / / (4 - \cdots / / od})
where
  do-while \, P \, a \equiv \mu \, x. \, \text{If } \, P \, \text{ Then } \, a \, ;; \, x \, \text{ Else } \, \text{Skip}

4.1.2 Unfolding rules for non-recursive primitives

lemma eval-wp-Abort:
  \text{wp} \, \text{Abort} \, P = (\lambda \, s. \, 0)
  (\text{proof})

lemma eval-wlp-Abort:
  \text{wlp} \, \text{Abort} \, P = (\lambda \, s. \, 1)
  (\text{proof})

lemma eval-wp-Skip:
  \text{wp} \, \text{Skip} \, P = P
  (\text{proof})

lemma eval-wlp-Skip:
  \text{wlp} \, \text{Skip} \, P = P
  (\text{proof})

lemma eval-wp-Apply:
  \text{wp} \, (\text{Apply} \, f) \, P = P \, o \, f
  (\text{proof})

lemma eval-wlp-Apply:
  \text{wlp} \, (\text{Apply} \, f) \, P = P \, o \, f
  (\text{proof})

lemma eval-wp-Seq:
  \text{wp} \, (a \, ;; \, b) \, P = (\text{wp} \, a \, o \, \text{wp} \, b) \, P
  (\text{proof})
lemma eval-wlp-Seq:
wlp (a ;; b) P = (wlp a o wlp b) P
(proof)

lemma eval-wp-PC:
wp (a Q\oplus b) P = (\lambda s. Q s \ast wp a P s + (1 - Q s) \ast wp b P s)
(proof)

lemma eval-wlp-PC:
wlp (a Q\oplus b) P = (\lambda s. Q s \ast wlp a P s + (1 - Q s) \ast wlp b P s)
(proof)

lemma eval-wp-DC:
wp (a \bigcap b) P = (\lambda s. \min (wp a P s) (wp b P s))
(proof)

lemma eval-wlp-DC:
wlp (a \bigcap b) P = (\lambda s. \min (wlp a P s) (wlp b P s))
(proof)

lemma eval-wp-AC:
wp (a \bigcup b) P = (\lambda s. \max (wp a P s) (wp b P s))
(proof)

lemma eval-wlp-AC:
wlp (a \bigcup b) P = (\lambda s. \max (wlp a P s) (wlp b P s))
(proof)

lemma eval-wp-Embed:
wp (Embed t) = t
(proof)

lemma eval-wlp-Embed:
wlp (Embed t) = t
(proof)

lemma eval-wp-SetDC:
wp (SetDC p S) R s = Inf ((\lambda a. wp (p a) R s) \cdot S s)
(proof)

lemma eval-wlp-SetDC:
wlp (SetDC p S) R s = Inf ((\lambda a. wlp (p a) R s) \cdot S s)
(proof)

lemma eval-wp-SetPC:
wp (SetPC f p) P = (\lambda s. \sum a \in supp (p s). p s a \ast wp (f a) P s)
(proof)
Lemma eval-wlp-SetPC:
\[
\text{wlp} (\text{SetPC } f p) P = (\lambda s. \sum_{a \in \text{supp} (p s)} p s a * \text{wlp} (f a) P s)
\]
(proof)

Lemma eval-wp-Mu:
\[
\text{wp} (\mu t. T t) = \text{lfp-trans} (\lambda t. \text{wp} (T (\text{Embed } t)))
\]
(proof)

Lemma eval-wlp-Mu:
\[
\text{wlp} (\mu t. T t) = \text{gfp-trans} (\lambda t. \text{wlp} (T (\text{Embed } t)))
\]
(proof)

Lemma eval-wp-Bind:
\[
\text{wp} (\text{Bind } g f) = (\lambda P s. \text{wp} (f (g s)) P s)
\]
(proof)

Lemma eval-wlp-Bind:
\[
\text{wlp} (\text{Bind } g f) = (\lambda P s. \text{wlp} (f (g s)) P s)
\]
(proof)

Use simp add:wp_eval to fully unfold a program fragment


Lemma Skip-Seq:
\[
\text{Skip } ;; \ A = A
\]
(proof)

Lemma Seq-Skip:
\[
\ A ;; \text{Skip} = A
\]
(proof)

Use these as simp rules to clear out Skips

Lemmas \text{skip-simps} = \text{Skip-Seq Seq-Skip}

end

4.2 Healthiness

Theory Healthiness imports Embedding begin
4.2. HEALTHINESS

4.2.1 The Healthiness of the Embedding

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

**Lemma** healthy-wp-Abort:

\[ \text{healthy \ (wp \ Abort)} \]

(proof)

**Lemma** nearly-healthy-wlp-Abort:

\[ \text{nearly-healthy \ (wlp \ Abort)} \]

(proof)

**Lemma** healthy-wp-Skip:

\[ \text{healthy \ (wp \ Skip)} \]

(proof)

**Lemma** nearly-healthy-wlp-Skip:

\[ \text{nearly-healthy \ (wlp \ Skip)} \]

(proof)

**Lemma** healthy-wp-Seq:

\[ \text{fixes \ } t::'s \ prog \text{ and } u \]

\[ \text{assumes \ } ht: \text{healthy \ (wp \ t)} \text{ and } hu: \text{healthy \ (wp \ u)} \]

\[ \text{shows \ healthy \ (wp \ (t ;; u))} \]

(proof)

**Lemma** nearly-healthy-wlp-Seq:

\[ \text{fixes \ } t::'s \ prog \text{ and } u \]

\[ \text{assumes \ } ht: \text{nearly-healthy \ (wlp \ t)} \text{ and } hu: \text{nearly-healthy \ (wlp \ u)} \]

\[ \text{shows \ nearly-healthy \ (wlp \ (t ;; u))} \]

(proof)

**Lemma** healthy-wp-PC:

\[ \text{fixes \ } f::'s \ prog \]

\[ \text{assumes \ } kf: \text{healthy \ (wp \ f)} \text{ and } kg: \text{healthy \ (wp \ g)} \]

\[ \text{and } uP: \text{unitary} \ P \]

\[ \text{shows \ healthy \ (wp \ (f \ p⊕ g))} \]

(proof)

**Lemma** nearly-healthy-wlp-PC:

\[ \text{fixes \ } f::'s \ prog \]

\[ \text{assumes \ } kf: \text{nearly-healthy \ (wlp \ f)} \]

\[ \text{and } kg: \text{nearly-healthy \ (wlp \ g)} \]

\[ \text{and } uP: \text{unitary} \ P \]

\[ \text{shows \ nearly-healthy \ (wlp \ (f \ p⊕ g))} \]

(proof)

**Lemma** healthy-wp-DC:

\[ \text{fixes \ } f::'s \ prog \]
assumes \( hf: \text{healthy} (wp f) \) and \( hg: \text{healthy} (wp g) \)
shows \( \text{healthy} (wp (f \cap g)) \)

\[\langle \text{proof} \rangle\]

**lemma nearly-healthy-wlp-DC:**
fixes \( f::'s \text{ prog} \)
assumes \( hf: \text{nearly-healthy} (wp f) \) and \( hg: \text{nearly-healthy} (wp g) \)
shows \( \text{nearly-healthy} (wp (f \cap g)) \)

\[\langle \text{proof} \rangle\]

**lemma healthy-wp-AC:**
fixes \( f::'s \text{ prog} \)
assumes \( hf: \text{healthy} (wp f) \) and \( hg: \text{healthy} (wp g) \)
shows \( \text{healthy} (wp (f \cup g)) \)

\[\langle \text{proof} \rangle\]

**lemma nearly-healthy-wlp-AC:**
fixes \( f::'s \text{ prog} \)
assumes \( hf: \text{nearly-healthy} (wp f) \) and \( hg: \text{nearly-healthy} (wp g) \)
shows \( \text{nearly-healthy} (wp (f \cup g)) \)

\[\langle \text{proof} \rangle\]

**lemma healthy-wp-Embed:**
\( \text{healthy} \ t = \implies \text{healthy} (wp (\text{Embed} \ t)) \)

\[\langle \text{proof} \rangle\]

**lemma nearly-healthy-wlp-Embed:**
\( \text{nearly-healthy} \ t = \implies \text{nearly-healthy} (wp (\text{Embed} \ t)) \)

\[\langle \text{proof} \rangle\]

**lemma healthy-wp-repeat:**
assumes \( h-a: \text{healthy} (wp a) \)
shows \( \text{healthy} (wp (\text{repeat} \ n \ a)) \) (\( \text{is} \ ?X \ n \))

\[\langle \text{proof} \rangle\]

**lemma nearly-healthy-wlp-repeat:**
assumes \( h-a: \text{nearly-healthy} (wp a) \)
shows \( \text{nearly-healthy} (wp (\text{repeat} \ n \ a)) \) (\( \text{is} \ ?X \ n \))

\[\langle \text{proof} \rangle\]

**lemma healthy-wp-SetDC:**
fixes \( \text{prog}::'b \Rightarrow 'a \text{ prog} \) and \( S::'a \Rightarrow 'b \text{ set} \)
assumes \( \text{healthy: } \forall \ x \ s. \ x \in S \ s = \implies \text{healthy} (wp (\text{prog} \ x)) \)
and \( \text{nonempty: } \forall \ s. \ \exists \ x. \ x \in S \ s \)
shows \( \text{healthy} (wp (\text{SetDC} \ \text{prog} \ S)) \) (\( \text{is} \ \text{healthy} \ ?T \))

\[\langle \text{proof} \rangle\]
lemma nearly-healthy-wlp-SetDC:
  fixes prog::'b ⇒ 'a prog and S::'a ⇒ 'b set
  assumes healthy: ∀x s. x ∈ S s ⇒ nearly-healthy (wlp (prog x))
  and nonempty: ∀s. ∃x. x ∈ S s
  shows nearly-healthy (wlp (SetDC prog S)) (is nearly-healthy ?T)
⟨proof⟩

lemma healthy-wp-SetPC:
  fixes p::'s ⇒ 'a ⇒ real
  and f::'a ⇒ 's prog
  assumes healthy: ∀a s. a ∈ supp (p s) ⇒ healthy (wp (f a))
  and sound: ∀s. sound (p s)
  and sub-dist: ∀s. (∑a∈supp (p s). p s a) ≤ 1
  shows healthy (wp (SetPC f p)) (is healthy ?X)
⟨proof⟩

lemma nearly-healthy-wlp-SetPC:
  fixes p::'s ⇒ 'a ⇒ real
  and f::'a ⇒ 's prog
  assumes healthy: ∀a s. a ∈ supp (p s) ⇒ nearly-healthy (wlp (f a))
  and sound: ∀s. sound (p s)
  and sub-dist: ∀s. (∑a∈supp (p s). p s a) ≤ 1
  shows nearly-healthy (wlp (SetPC f p)) (is nearly-healthy ?X)
⟨proof⟩

lemma healthy-wp-Apply:
  healthy (wp (Apply f))
⟨proof⟩

lemma nearly-healthy-wlp-Apply:
  nearly-healthy (wlp (Apply f))
⟨proof⟩

lemma healthy-wp-Bind:
  fixes f::'s ⇒ 'a
  assumes hsub: ∀s. healthy (wp (p (f s)))
  shows healthy (wp (Bind f p))
⟨proof⟩

lemma nearly-healthy-wlp-Bind:
  fixes f::'s ⇒ 'a
  assumes hsub: ∀s. nearly-healthy (wlp (p (f s)))
  shows nearly-healthy (wlp (Bind f p))
⟨proof⟩

4.2.2 Healthiness for Loops

lemma wp-loop-step-mono:
  fixes t u::'s trans
assumes \( \text{hb}: \text{healthy} (wp \ \text{body}) \)
and \( \text{le}: \text{le-trans} \ t \ u \)
and \( \text{ht}: \bigwedge P. \text{sound} \ P \Rightarrow \text{sound} (t \ P) \)
and \( \text{hu}: \bigwedge P. \text{sound} \ P \Rightarrow \text{sound} (u \ P) \)
shows \( \text{le-trans} (wp (\text{body} :: \text{Embed} t \ « G \oplus \text{Skip})) \)
\( (wp (\text{body} :: \text{Embed} u \ « G \oplus \text{Skip})) \)

\( \langle \text{proof} \rangle \)

\text{lemma} \ wlp\text{-}loop\text{-}step\text{-}mono:
fixes \( t \ u \)·′s trans
assumes \( \text{mb}: \text{nearly-healthy} (wlp \ \text{body}) \)
and \( \text{le}: \text{le-utrans} \ t \ u \)
and \( \text{ht}: \bigwedge P. \text{unitary} \ P \Rightarrow \text{unitary} (t \ P) \)
and \( \text{hu}: \bigwedge P. \text{unitary} \ P \Rightarrow \text{unitary} (u \ P) \)
shows \( \text{le-utrans} (wlp (\text{body} :: \text{Embed} t \ « G \oplus \text{Skip})) \)
\( (wlp (\text{body} :: \text{Embed} u \ « G \oplus \text{Skip})) \)

\( \langle \text{proof} \rangle \)

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

\text{lemma} \ lfp\text{-}loop\text{-}fp:
assumes \( \text{hb}: \text{healthy} (wp \ \text{body}) \)
and \( \text{sP}: \text{sound} \ P \)
shows \( \lambda s. \ « G » s * wp \text{body} (\lambda s. \text{bound-of} \ P) s + \ « \text{N} G » s * P s \vdash \lambda s. \text{bound-of} \ P \)

\( \langle \text{proof} \rangle \)

\text{lemma} \ lfp\text{-}loop\text{-}greatest:
fixes \( P \)·′s expect
assumes \( \text{lb}: \bigwedge R. \lambda s. \ « G » s * wp \text{body} R s + \ « \text{N} G » s * P s \vdash R \Rightarrow \text{sound} R \Rightarrow Q \vdash R \)
and \( \text{hb}: \text{healthy} (wp \ \text{body}) \)
and \( \text{sP}: \text{sound} \ P \)
and \( \text{sQ}: \text{sound} \ Q \)
shows \( Q \vdash lfp\text{-}exp (\lambda Q s. \ « G » s * wp \text{body} Q s + \ « \text{N} G » s * P s) \)

\( \langle \text{proof} \rangle \)

\text{lemma} \ lfp\text{-}loop\text{-}sound:
fixes \( P \)·′s expect
assumes \( \text{hb}: \text{healthy} (wp \ \text{body}) \)
and \( \text{sP}: \text{sound} \ P \)
shows \( \text{sound} (lfp\text{-}exp (\lambda Q s. \ « G » s * wp \text{body} Q s + \ « \text{N} G » s * P s)) \)

\( \langle \text{proof} \rangle \)

\text{lemma} \ wlp\text{-}loop\text{-}step\text{-}unitary:
fixes \( t \ u \)·′s trans
assumes \( \text{hb}: \text{nearly-healthy} (wlp \ \text{body}) \)
and \( \text{ht}: \bigwedge P. \text{unitary} \ P \Rightarrow \text{unitary} (t \ P) \)
and \( \text{uP}: \text{unitary} \ P \)
4.2. HEALTHINESS

shows unitary (wp (body ;; Embed t « G ⊕ Skip) P)
⟨proof⟩

lemma wp-loop-step-sound:
  fixes t u:‘s trans
  assumes hb: healthy (wp body)
  and ht: P. sound P ⟹ sound (t P)
  and sP: sound P
  shows sound (wp (body ;; Embed t « G ⊕ Skip) P)
⟨proof⟩

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

lemma wp-Loop1:
  fixes body :: ‘s prog
  assumes unitary: unitary P
  and healthy: nearly-healthy (wp body)
  shows wp (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⟩

lemma wp-loop-sound:
  assumes sP: sound P
  and hb: healthy (wp body)
  shows sound (wp do G → body od P)
⟨proof⟩

Likewise, we can rewrite strict loops.

lemma wp-Loop1:
  fixes body :: ‘s prog
  assumes sP: sound P
  and hb: healthy (wp body)
  shows wp (do G → body od) P =
    lfp-exp (λQ s. «G» s * wp body Q s + «N G» s * P s)
  (is ?X = lfp-exp (?Y P))
⟨proof⟩

lemma nearly-healthy-wlp-loop:
  fixes body::‘s prog
  assumes hb: nearly-healthy (wp body)
  shows nearly-healthy (wp (do G → body od))
⟨proof⟩

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

lemma healthy-wp-loop:
  fixes body::‘s prog
assumes hb: healthy (wp body)
shows healthy (wp (do G → body od))
(proof)

Use 'simp add:healthy_intros' or 'blast intro:healthy_intros' as appropriate
to discharge healthiness side-conditions for primitive programs automatically.

lemmas healthy-intros =
  healthy-wp-Abort nearly-healthy-wlp-Abort healthy-wp-Skip nearly-healthy-wlp-Skip
  healthy-wp-Seq nearly-healthy-wlp-Seq healthy-wp-PC nearly-healthy-wlp-PC
  healthy-wp-DC nearly-healthy-wlp-DC healthy-wp-AC nearly-healthy-wlp-AC
  healthy-wp-Embed nearly-healthy-wlp-Embed healthy-wp-Apply nearly-healthy-wlp-Apply
  healthy-wp-SetDC nearly-healthy-wlp-SetDC healthy-wp-SetPC nearly-healthy-wlp-SetPC
  healthy-wp-Bind nearly-healthy-wlp-Bind healthy-wp-repeat nearly-healthy-wlp-repeat
  healthy-wp-loop nearly-healthy-wlp-loop

end

4.3 Continuity

theory Continuity imports Healthiness begin

We rely on one additional healthiness property, continuity, which is shown
here separately, as its proof relies, in general, on healthiness. It is only
relevant when a program appears in an inductive context i.e. inside a loop.

A continuous transformer preserves limits (or the suprema of ascending
chains).

definition bd-cts :: 's trans ⇒ bool
where bd-cts t = (∀ M. (∀ i. M i ⊢ M (Suc i)) ∧ sound (M i) →→
  (∃ b. ∀ i. bounded-by b (M i) →→
  t (Sup-exp (range M)) = Sup-exp (range (t o M)))

lemma bd-ctsD:
[ [ bd-cts t; ∃ i. M i ⊢ M (Suc i); ∃ i. sound (M i); ∃ i. bounded-by b (M i) ]] →→
  t (Sup-exp (range M)) = Sup-exp (range (t o M))
(proof)

lemma bd-ctsI:
(∀ b M. (∃ i. M i ⊢ M (Suc i)) ⇒ (∃ i. sound (M i)) ⇒ (∃ i. bounded-by b
  (M i)) ⇒→
  t (Sup-exp (range M)) = Sup-exp (range (t o M)) ⇒ bd-cts t
(proof)

A generalised property for transformers of transformers.

definition bd-cts-tr :: ('s trans ⇒ 's trans) ⇒ bool
where bd-cts-tr T = (∀ M. (∃ i. le-trans (M i) (M (Suc i)) ∧ feasible (M i)) →→
equiv-trans \((T \ (\text{Sup}-\text{trans} \ (M \cdot \text{UNIV}))) \ (\text{Sup}-\text{trans} \ ((T \circ M) \cdot \text{UNIV}))\))

**lemma** bd-cts-trD:
\[
\text{bd-cts-tr } T; \ \bigwedge i. \ \text{le-trans} \ (M \ i) \ (M \ (\text{Suc} \ i)) ; \ \bigwedge i. \ \text{feasible} \ (M \ i) \ \implies \\
equiv-trans \ (T \ (\text{Sup}-\text{trans} \ (M \cdot \text{UNIV}))) \ (\text{Sup}-\text{trans} \ ((T \circ M) \cdot \text{UNIV}))
\]

**lemma** bd-cts-trI:
\[
(\bigwedge M. \ (\bigwedge i. \ \text{le-trans} \ (M \ i) \ (M \ (\text{Suc} \ i))) \ \implies \ (\bigwedge i. \ \text{feasible} \ (M \ i)) \ \implies \\
equiv-trans \ (T \ (\text{Sup}-\text{trans} \ (M \cdot \text{UNIV}))) \ (\text{Sup}-\text{trans} \ ((T \circ M) \cdot \text{UNIV}))
\]
\[\implies \text{bd-cts-tr } T\]

### 4.3.1 Continuity of Primitives

**lemma** cts-wp-Abort:
\[
\text{bd-cts} \ (\wp \ (\text{Abort}::'s \ \text{prog}))
\]

**lemma** cts-wp-Skip:
\[
\text{bd-cts} \ (\wp \ \text{Skip})
\]

**lemma** cts-wp-Apply:
\[
\text{bd-cts} \ (\wp \ (\text{Apply} \ f))
\]

**lemma** cts-wp-Bind:
\[
\text{fixes} \ a::'a \Rightarrow 's \ \text{prog} \\
\text{assumes} \ ca: \ \bigwedge s. \ \text{bd-cts} \ (\wp \ (a \ (f \ s))) \\
\text{shows} \ \text{bd-cts} \ (\wp \ (\text{Bind} \ f \ a))
\]

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:
\[
\text{fixes} \ a \ b::'s \ \text{prog} \\
\text{assumes} \ ca: \ \text{bd-cts} \ (\wp \ a) \\
\quad \text{and} \ cb: \ \text{bd-cts} \ (\wp \ b) \\
\quad \text{and} \ ha: \ \text{healthy} \ (\wp \ a) \\
\quad \text{and} \ hb: \ \text{healthy} \ (\wp \ b) \\
\text{shows} \ \text{bd-cts} \ (\wp \ (a \ \bigcap \ b))
\]

**lemma** cts-wp-Seq:
\[
\text{fixes} \ a \ b::'s \ \text{prog} \\
\text{assumes} \ ca: \ \text{bd-cts} \ (\wp \ a)
\]
\textbf{lemma} \texttt{cts-wp-PC}: \\
\textbf{fixes} \( a \vdash s \) prog  \\
\textbf{assumes} \( ca \): bd-cts (wp \( a \))  \\
\textbf{and} \( cb \): bd-cts (wp \( b \))  \\
\textbf{and} \( ha \): healthy (wp \( a \))  \\
\textbf{and} \( hb \): healthy (wp \( b \))  \\
\textbf{and} \( up \): unitary \( p \)  \\
\textbf{shows} bd-cts (wp (PC a p b))  \\
\textbf{⟨proof⟩}  

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.

\textbf{lemma} \texttt{SetPC-Bind}:  \\
SetPC a p = Bind p (\( \lambda p. \text{SetPC} a (\lambda -. p) \))  \\
\textbf{⟨proof⟩}  

\textbf{lemma} \texttt{SetPC-remove}:  \\
\textbf{assumes} \( nz \): \( p \, x \neq 0 \) and \( n1 \): \( p \, x \neq 1 \)  \\
\textbf{and} \( fsupp \): finite (supp \( p \))  \\
\textbf{shows} \( \text{SetPC} a (\lambda -. p) = PC (a \, x) (\lambda -. p \, x) (\text{SetPC} a (\lambda -. \text{dist-remove} p \, x)) \)  \\
\textbf{⟨proof⟩}  

\textbf{lemma} \texttt{cts-bot}:  \\
bd-cts (\( \lambda(P::'s \, \text{expect}) (s::'s). \, 0::\text{real} \))  \\
\textbf{⟨proof⟩}  

\textbf{lemma} \texttt{wp-SetPC-nil}:  \\
wp (SetPC a (\( \lambda s. a. \, 0 \))) = (\lambda P \, s. \, 0)  \\
\textbf{⟨proof⟩}  

\textbf{lemma} \texttt{SetPC-sgl}:  \\
supp \( p \) = \{x\} \implies \text{SetPC} a (\lambda -. p) = (\lambda a \, b \, P \, s. \, p \, x \star a \, x \, a \, b \, P \, s) \)  \\
\textbf{⟨proof⟩}  

\textbf{lemma} \texttt{bd-cts-scale}:  \\
\textbf{fixes} \( a::'s \) trans  \\
\textbf{assumes} \( ca \): bd-cts \( a \)  \\
\textbf{and} \( ha \): healthy \( a \)  \\
\textbf{and} \( nnc \): \( 0 \leq c \)  \\
\textbf{shows} bd-cts (\( \lambda P \, s. \, c \star a \, P \, s \))  \\
\textbf{⟨proof⟩}
4.3. CONTINUITY

\textbf{lemma} \textit{cts-wp-SetPC-const}:
\begin{itemize}
\item \texttt{fixes} \texttt{a::'a \Rightarrow 's prog}
\item \texttt{assumes} ca: \(\forall x. \ x \in (\text{supp } p) \Rightarrow \text{bd-cts} (wp (a \ x))\)
\item \texttt{and} ha: \(\forall x. \ x \in (\text{supp } p) \Rightarrow \text{healthy} (wp (a \ x))\)
\item \texttt{and} up: \text{unitary } p
\item \texttt{and} sump: \text{sum } p (\text{supp } p) \leq 1
\item \texttt{and} fsupp: \text{finite } (\text{supp } p)
\item \texttt{shows} \text{bd-cts} (wp (SetPC a (\lambda \cdot p)))
\end{itemize}
\texttt{(proof)}

\textbf{lemma} \textit{cts-wp-SetPC}:
\begin{itemize}
\item \texttt{fixes} \texttt{a::'a \Rightarrow 's prog}
\item \texttt{assumes} ca: \(\forall x s. \ x \in (\text{supp } p s) \Rightarrow \text{bd-cts} (wp (a \ x))\)
\item \texttt{and} ha: \(\forall x s. \ x \in (\text{supp } p s) \Rightarrow \text{healthy} (wp (a \ x))\)
\item \texttt{and} up: \text{unitary } (p s)
\item \texttt{and} sump: \text{sum } (p s) (\text{supp } (p s)) \leq 1
\item \texttt{and} fsupp: \text{finite } (\text{supp } (p s))
\item \texttt{shows} \text{bd-cts} (wp (SetPC a p))
\end{itemize}
\texttt{(proof)}

\textbf{lemma} \textit{wp-SetDC-Bind}:
\(\text{SetDC a S} = \text{Bind } S (\lambda s. \text{SetDC a (\lambda \cdot S)})\)
\texttt{(proof)}

\textbf{lemma} \textit{SetDC-finite-insert}:
\begin{itemize}
\item \texttt{assumes} fS: \text{finite } S
\item \texttt{and} neS: \text{S} \neq \{\}
\item \texttt{shows} \text{SetDC a (\lambda \cdot insert } x S) = a x \bigcap \text{SetDC a (\lambda \cdot S)}
\end{itemize}
\texttt{(proof)}

\textbf{lemma} \textit{SetDC-singleton}:
\(\text{SetDC a (\lambda \cdot \{x\})} = a x\)
\texttt{(proof)}

\textbf{lemma} \textit{cts-wp-SetDC-const}:
\begin{itemize}
\item \texttt{fixes} \texttt{a::'a \Rightarrow 's prog}
\item \texttt{assumes} ca: \(\forall x. \ x \in S \Rightarrow \text{bd-cts} (wp (a \ x))\)
\item \texttt{and} ha: \(\forall x. \ x \in S \Rightarrow \text{healthy} (wp (a \ x))\)
\item \texttt{and} fS: \text{finite } S
\item \texttt{and} neS: \text{S} \neq \{\}
\item \texttt{shows} \text{bd-cts} (wp (SetDC a (\lambda \cdot S)))
\end{itemize}
\texttt{(proof)}

\textbf{lemma} \textit{cts-wp-SetDC}:
\begin{itemize}
\item \texttt{fixes} \texttt{a::'a \Rightarrow 's prog}
\item \texttt{assumes} ca: \(\forall x s. \ x \in S s \Rightarrow \text{bd-cts} (wp (a \ x))\)
\item \texttt{and} ha: \(\forall x s. \ x \in S s \Rightarrow \text{healthy} (wp (a \ x))\)
\item \texttt{and} fS: \text{finite } (S s)
\item \texttt{and} neS: \text{S} \text{ s} \neq \{\}
\end{itemize}
shows \( \text{bd-cts} (\text{wp} (\text{SetDC} \ a \ S)) \)

\( \langle \text{proof} \rangle \)

\textbf{lemma} \( \text{cts-wp-repeat} \):

\[ \text{bd-cts} (\text{wp} \ a) \Rightarrow \text{healthy} (\text{wp} \ a) \Rightarrow \text{bd-cts} (\text{wp} (\text{repeat} \ n \ a)) \]

\( \langle \text{proof} \rangle \)

\textbf{lemma} \( \text{cts-wp-Embed} \):

\[ \text{bd-cts} \ t \Rightarrow \text{bd-cts} (\text{wp} (\text{Embed} \ t)) \]

\( \langle \text{proof} \rangle \)

\textbf{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.

\textbf{lemma} \( \text{cts-wp-loopstep} \):

\textit{fixes} \( \text{body} :: \'s \ \text{prog} \)
\textit{assumes} \( \text{hb} : \text{healthy} (\text{wp} \ \text{body}) \)
\textit{and} \( \text{cb} : \text{bd-cts} (\text{wp} \ \text{body}) \)
\textit{shows} \( \text{bd-cts-tr} (\lambda x. \text{wp} (\text{body} ;; \text{Embed} \ x ;; G) ;; \text{Skip}) \) \( \langle \text{proof} \rangle \)

\textbf{end}

\textbf{4.4 Continuity and Induction for Loops}

\textbf{theory} \( \text{LoopInduction} \) \textbf{imports} \( \text{Healthiness} \ \text{Continuity} \) \textbf{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.

\textbf{lemma} \( \text{wp-loop-step-mono-trans} \):

\textit{fixes} \( \text{body} :: \'s \ \text{prog} \)
\textit{assumes} \( \text{sP} : \text{sound} \ P \)
\textit{and} \( \text{hb} : \text{healthy} (\text{wp} \ \text{body}) \)
\textit{shows} \( \text{mono-trans} (\lambda Q s. \text{wp} \ (\text{body} ;; \text{Embed} \ x ;; G) ;; \text{Skip}) \) \( \langle \text{proof} \rangle \)

We can therefore apply the standard fixed-point lemmas to unfold it:

\textbf{lemma} \( \text{lfp-wp-loop-unfold} \):

\textit{fixes} \( \text{body} :: \'s \ \text{prog} \)
\textit{assumes} \( \text{hb} : \text{healthy} (\text{wp} \ \text{body}) \)
\textit{and} \( \text{sP} : \text{sound} \ P \)
shows \( \text{lfp-exp} (\lambda Q \ s. \langle G \rangle s \ast \text{wp body} Q \ s + \langle N \ G \rangle s \ast P \ s) = \\
(\lambda s. \langle G \rangle s \ast \text{wp body} (\text{lfp-exp} (\lambda Q \ s. \langle G \rangle s \ast \text{wp body} Q \ s + \langle N \ G \rangle s \ast P \ s)) s + \\
\langle N \ G \rangle s \ast P \ s) \)

\langle \text{proof} \rangle

lemma wp-loop-step-unitary:
\begin{align*}
&\text{fixes body::'s prog} \\
&\text{assumes hb: healthy (wp body) and uP: unitary P and uQ: unitary Q} \\
&\text{shows unitary (\lambda s. \langle G \rangle s \ast \text{wp body} Q \ s + \langle N \ G \rangle s \ast P \ s)} \\
&\langle \text{proof} \rangle
\end{align*}

From the lattice structure on transformers, we establish a transfinite induction principle for loops. We use this to show a number of properties, particularly subdistributivity, for loops. This proof follows the pattern of lemma lfp_ordinal_induct in HOL/Inductive.

lemma loop-induct:
\begin{align*}
&\text{fixes body::'s prog} \\
&\text{assumes hwp: healthy (wp body) and hwlp: nearly-healthy (wlp body)} \\
&\text{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 \implies \text{unitary } (\text{snd } x \ Q) \right] \implies \\
\quad P \ (\text{Sup-trans } (\text{fst } ' S)) \ (\text{Inf-utrans } (\text{snd } ' S)) \\
&\text{and IH: } \forall t \ u. \left[ \ P t u; \ \text{feasible } t; \ \forall Q. \ \text{unitary } Q \implies \text{unitary } (u Q) \right] \implies \\
\quad P \ (\text{wp } (\text{body } :: \text{Embed } t \ ' G \oplus \text{Skip})); \ (\text{wlp } (\text{body } :: \text{Embed } u \ ' G \oplus \text{Skip})) \\
&\text{and P-equiv: } \forall t \ t' \ u. \left[ \ P t u; \ \text{equiv-trans } t t' \ ; \ \text{equiv-utrans } a u' \right] \implies P t' \ u' \\
&\quad \text{The property must be preserved by equivalence} \\
&\text{shows } P \ (\text{wp } (\text{do } G \rightarrow \text{body od})); \ (\text{wlp } (\text{do } G \rightarrow \text{body od})) \\
&\quad \text{The property can refer to both interpretations simultaneously. The unifier will happily apply the rule to just one or the other, however.} \\
&\langle \text{proof} \rangle
\end{align*}

4.4.1 The Limit of Iterates

The iterates of a loop are its sequence of finite unrollings. We show shortly that this converges on the least fixed point. This is enormously useful, as
we can appeal to various properties of the finite iterates (which will follow
by finite induction), which we can then transfer to the limit.

**definition iterates :: ’s prog ⇒ (’s ⇒ bool) ⇒ nat ⇒ ’s trans**

**where iterates body G i = ((λx. wp (body ;; Embed x « G ⊕ Skip)) ◀ i) (λP s. 0)**

**lemma iterates-0[simp]:**

\[ iterates \text{ body } G \ 0 = (\lambda P \ s. 0) \]

⟨proof⟩

**lemma iterates-Suc[simp]:**

\[ iterates \text{ body } G \ (Suc \ i) = wp (\text{body ;;} \ Embed \ (iterates \text{ body } G \ i) \ « G \oplus \ Skip) \]

⟨proof⟩

All iterates are healthy.

**lemma iterates-healthy:**

\[ healthy \ (wp \text{ body}) ⇒ healthy \ (iterates \text{ body } G \ i) \]

⟨proof⟩

The iterates are an ascending chain.

**lemma iterates-increasing:**

fixes body ::’s prog

assumes hb: healthy (wp body)

shows le-trans (iterates body G i) (iterates body G (Suc i))

⟨proof⟩

**lemma wp-loop-step-bounded:**

fixes t::’s trans and Q::’s expect

assumes nQ: nneg Q

and bQ: bounded-by b Q

and ht: healthy t

and hb: healthy (wp body)

shows bounded-by b (wp (body ;; Embed t « G ⊕ Skip) Q)

⟨proof⟩

This is the key result: The loop is equivalent to the supremum of its iterates.
This proof follows the pattern of lemma continuous_lfp in HOL/Library/Continuity.

**lemma lfp-iterates:**

fixes body::’s prog

assumes hb: healthy (wp body)

and cb: bd-cts (wp body)

shows equiv-trans (wp (do G → body od)) (Sup-trans (range (iterates body G)))

(is equiv-trans ?X ?Y)

⟨proof⟩

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:**
4.5. SUBLINEARITY

fixes body::'s prog
assumes hb: healthy (wp body)
    and cb: bd-cts (wp body)
    and sP: sound P
shows (λi. iterates body G i P s) \rightarrow wp (do G → body od) P s
⟨proof⟩

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⟩

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⟩

lemmas cts-intros =
cts-wp-Abort cts-wp-Skip
cuts-wp-Seq cts-wp-PC
cuts-wp-DC cts-wp-Embed
cuts-wp-Apply cts-wp-SetDC
cuts-wp-SetPC cts-wp-Bind
cuts-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)
⟨proof⟩

lemma sublinear-wp-Abort:
sublinear (wp Abort)
⟨proof⟩

**lemma** sublinear-wp-Apply:
sublinear (wp (Apply f))
⟨proof⟩

**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)
sows sublinear (wp (x ;; y))
⟨proof⟩

**lemma** sublinear-wp-PC:
fixes x::'s prog
assumes slx: sublinear (wp x) and sly: sublinear (wp y)
and uP: unitary P
shows sublinear (wp (x p⊕ y))
⟨proof⟩

**lemma** sublinear-wp-DC:
fixes x::'s prog
assumes slx: sublinear (wp x) and sly: sublinear (wp y)
sows sublinear (wp (x d y))
⟨proof⟩

As for continuity, we insist on a finite support.

**lemma** sublinear-wp-SetPC:
fixes p::'a ⇒ 's prog
assumes slp: ∀s a. a ∈ supp (P s) ⇒ sublinear (wp (p a))
and sum: ∀s. (∑ a∈supp (P s). P s a) ≤ 1
and nnP: ∀s a. 0 ≤ P s a
and fin: ∀s. finite (supp (P s))
sows sublinear (wp (SetPC p P))
⟨proof⟩

**lemma** sublinear-wp-SetDC:
fixes p::'a ⇒ 's prog
assumes slp: ∀s a. a ∈ S s ⇒ sublinear (wp (p a))
and hp: ∀s a. a ∈ S s ⇒ healthy (wp (p a))
and ne: ∀s. S s ≠ {}
sows sublinear (wp (SetDC p S))
⟨proof⟩

**lemma** sublinear-wp-Embed:
sublinear t ⇒ sublinear (wp (Embed t))
⟨proof⟩
4.5. **SUBLINEARITY**

**Lemma sublinear-wp-repeat:**

\[
\left[ \text{sublinear} \ (\text{wp} \ p) ; \ \text{healthy} \ (\text{wp} \ p) \right] \implies \text{sublinear} \ (\text{wp} \ (\text{repeat} \ n \ p))
\]

(proof)

**Lemma sublinear-wp-Bind:**

\[
\left[ \bigwedge s. \ \text{sublinear} \ (\text{wp} \ (a \ (f \ s))) \right] \implies \text{sublinear} \ (\text{wp} \ (\text{Bind} \ f \ a))
\]

(proof)

### 4.5.2 Sublinearity for Loops

We break the proof of sublinearity loops into separate proofs of sub-distributivity and sub-additivity. The first follows by transfinite induction.

**Lemma sub-distrib-wp-loop:**

fixes body::’s prog

assumes sdb: sub-distrib (wp body)
and hb: healthy (wp body)
and nhb: nearly-healthy (wp body)
shows sub-distrib (wp (do G −→ body od))

(proof)

For sub-additivity, we again use the limit-of-iterates characterisation. Firstly, all iterates are sublinear:

**Lemma sublinear-iterates:**

assumes hb: healthy (wp body)
and sb: sublinear (wp body)
shows sublinear (iterates body G i)

(proof)

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 body)
and cb: bd-cts (wp body)
and hwp: healthy (wp body)
shows sub-add (wp (do G −→ body od))

(proof)

**Lemma sublinear-wp-loop:**

fixes body::’s prog

assumes hb: healthy (wp body)
and nhb: nearly-healthy (wp body)
and sb: sublinear (wp body)
and cb: bd-cts (wp body)
shows sublinear (wp (do G −→ body od))

(proof)

**Lemmas sublinear-intros** =
sublinear-wp-Abort
sublinear-wp-Skip
sublinear-wp-Apply
sublinear-wp-Seq
sublinear-wp-PC
sublinear-wp-DC
sublinear-wp-SetPC
sublinear-wp-SetDC
sublinear-wp-Embed
sublinear-wp-repeat
sublinear-wp-Bind
sublinear-wp-loop

end

4.6 Determinism

theory Determinism imports WellDefined begin

We provide a set of lemmas for establishing that appropriately restricted
programs are fully additive, and maximal in the refinement order. This is
particularly useful with data refinement, as it implies correspondence.

4.6.1 Additivity

lemma additive-wp-Abort:
  additive (wp (Abort))
⟨proof⟩

wlp Abort is not additive.

lemma additive-wp-Skip:
  additive (wp (Skip))
⟨proof⟩

lemma additive-wp-Apply:
  additive (wp (Apply f))
⟨proof⟩

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⟩

lemma additive-wp-PC:
  [ additive (wp a); additive (wp b) ] ⇒ additive (wp (a △ b))
4.6. DETERMINISM

(proof)

DC is not additive.

lemma additive-wp-SetPC:
[ \forall x s. x \in \text{supp} (p s) \Rightarrow \text{additive} (wp (a x)); \exists s. \text{finite} (\text{supp} (p s)) ] \Rightarrow
\text{additive} (wp (\text{SetPC} a p))
(proof)

lemma additive-wp-Bind:
[ \forall x. \text{additive} (wp (a (f x))) ] \Rightarrow \text{additive} (wp (\text{Bind} f a))
(proof)

lemma additive-wp-Embed:
[ additive t ] \Rightarrow \text{additive} (wp (\text{Embed} t))
(proof)

lemma additive-wp-repeat:
\text{additive} (wp a) \Rightarrow \text{well-def} a \Rightarrow \text{additive} (wp (\text{repeat} n a))
(proof)

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} (wp \text{Skip})
(proof)

lemma max-wp-Apply:
\text{maximal} (wp (\text{Apply} f))
(proof)

lemma max-wp-Seq:
[ \text{maximal} (wp a); \text{maximal} (wp b) ] \Rightarrow \text{maximal} (wp (a ; b))
(proof)

lemma max-wp-PC:
[ \text{maximal} (wp a); \text{maximal} (wp b) ] \Rightarrow \text{maximal} (wp (a \text{PC} b))
(proof)

lemma max-wp-DC:
[ \text{maximal} (wp a); \text{maximal} (wp b) ] \Rightarrow \text{maximal} (wp (a \text{DC} b))
(proof)
lemma max-wp-SetPC:
\[ \{ \land s. a \in \text{supp} (P s) \implies \text{maximal} (wp (p a)) \} \implies \text{maximal} (wp (\text{SetPC} p P)) \]
⟨proof⟩

lemma max-wp-SetDC:
\begin{align*}
\text{fixes } & p : \forall a \Rightarrow s \text{ prog} \\
\text{assumes } & wp: \land s. a \in S s \implies \text{maximal} (wp (p a)) \\
\text{and } & \text{ne}: \land s. S s \neq \{\} \\
\text{shows } & \text{maximal} (wp (\text{SetDC} p S))
\end{align*}
⟨proof⟩

lemma max-wp-Embed:
\[ \text{maximal} t \implies \text{maximal} (wp (\text{Embed} t)) \]
⟨proof⟩

lemma max-wp-repeat:
\[ \text{maximal} (wp a) \implies \text{maximal} (wp (\text{repeat} n a)) \]
⟨proof⟩

lemma max-wp-Bind:
\begin{align*}
\text{assumes } & ma: \land s. \text{maximal} (wp (a (f s))) \\
\text{shows } & \text{maximal} (wp (\text{Bind} f a))
\end{align*}
⟨proof⟩

lemmas max-intros =
\begin{align*}
\text{max-wp-Skip} & \quad \text{max-wp-Apply} \\
\text{max-wp-Seq} & \quad \text{max-wp-PC} \\
\text{max-wp-DC} & \quad \text{max-wp-SetPC} \\
\text{max-wp-SetDC} & \quad \text{max-wp-Embed} \\
\text{max-wp-Bind} & \quad \text{max-wp-repeat}
\end{align*}

A healthy transformer that terminates is maximal.

lemma healthy-term-max:
\begin{align*}
\text{assumes } & ht: \text{healthy} t \\
\text{and } & \text{trm}: \lambda s. 1 \vdash t (\lambda s. 1) \\
\text{shows } & \text{maximal} t
\end{align*}
⟨proof⟩

4.6.3 Determinism

lemma det-wp-Skip:
\[ \text{deterministic} (wp \text{ Skip}) \]
⟨proof⟩

lemma det-wp-Apply:
\[ \text{deterministic} (wp (\text{Apply} f)) \]
⟨proof⟩
**4.7. WELL-DEFINED PROGRAMS.**

**lemma det-wp-Seq:**
\[ \text{determ} (\text{wp } a) \implies \text{determ} (\text{wp } b) \implies \text{well-def } b \implies \text{determ} (\text{wp } (a ;; b)) \]
\(\langle \text{proof} \rangle\)

**lemma det-wp-PC:**
\[ \text{determ} (\text{wp } a) \implies \text{determ} (\text{wp } b) \implies \text{determ} (\text{wp } (a p\oplus b)) \]
\(\langle \text{proof} \rangle\)

**lemma det-wp-SetPC:**
\[ (\forall x s. x \in \text{supp } (p s)) \implies \text{determ} (\text{wp } (a x)) \implies \\
(\forall s. \text{finite } (\text{supp } (p s))) \implies \\
(\forall s. \text{sum } (p s) (\text{supp } (p s)) = 1) \implies \\
\text{determ} (\text{wp} (\text{SetPC } a p)) \]
\(\langle \text{proof} \rangle\)

**lemma det-wp-Bind:**
\[ (\forall x. \text{determ} (\text{wp } (a (f x)))) \implies \text{determ} (\text{wp } (\text{Bind } f a)) \]
\(\langle \text{proof} \rangle\)

**lemma det-wp-Embed:**
\[ \text{determ } t \implies \text{determ} (\text{wp } (\text{Embed } t)) \]
\(\langle \text{proof} \rangle\)

**lemma det-wp-repeat:**
\[ \text{determ} (\text{wp } a) \implies \text{well-def } a \implies \text{determ} (\text{wp } (\text{repeat } n a)) \]
\(\langle \text{proof} \rangle\)

**lemmas determ-intros =**

- det-wp-Skip
- det-wp-Apply
- det-wp-Seq
- det-wp-PC
- det-wp-SetPC
- det-wp-Bind
- det-wp-Embed
- det-wp-repeat

**end**

**4.7 Well-Defined Programs.**

**theory WellDefined imports**

- Healthiness
- Sublinearity
- LoopInduction

**begin**

The definition of a well-defined program collects the various notions of healthiness and well-behavedness that we have so far established: healthiness of the strict and liberal transformers, continuity and sublinearity of the strict transformers, and two new properties. These are that the strict transformer always lies below the liberal one (i.e. that it is at least as strict,
recalling the standard embedding of a predicate), and that expectation con-
junction is distributed between them in a particular manner, which will be
vital in establishing the loop rules.

4.7.1 Strict Implies Liberal

This establishes the first connection between the strict and liberal interpre-
tations \((wp\text{ and } wlp)\).

**definition**

\[ wp\text{-under-wlp} :: \text{'s prog } \Rightarrow \text{ bool} \]

**where**

\[ wp\text{-under-wlp prog} \equiv \forall P. \text{ unitary } P \implies wp\text{ prog } P \vdash wlp\text{ prog } P \]

**lemma** \(wp\text{-under-wlpI}[\text{intro}]:\)

\[ \left[ \left[ \forall P. \text{ unitary } P \implies wp\text{ prog } P \vdash wlp\text{ prog } P \right] \right] \implies wp\text{-under-wlp prog} \]

**lemma** \(wp\text{-under-wlpD}[\text{dest}]:\)

\[ \left[ wp\text{-under-wlp prog}; \text{ unitary } P \right] \implies wp\text{ prog } P \vdash wlp\text{ prog } P \]

**lemma** \(wp\text{-under-le-trans}:\)

\[ wp\text{-under-wlp a } \Rightarrow le\text{-utrans } (wp\text{ a}) (wlp\text{ a}) \]

**lemma** \(wp\text{-under-wlp-Abort}:\)

\[ wp\text{-under-wlp Abort} \]

**lemma** \(wp\text{-under-wlp-Skip}:\)

\[ wp\text{-under-wlp Skip} \]

**lemma** \(wp\text{-under-wlp-Apply}:\)

\[ wp\text{-under-wlp } (\text{Apply } f) \]

**lemma** \(wp\text{-under-wlp-Seq}:\)

**assumes** \(h\text{-wlp-a}: \text{nearly-healthy } (wlp\text{ a})\)

\[ \text{and } h\text{-wp-b}: \text{ healthy } (wp\text{ b}) \]

\[ \text{and } h\text{-wlp-b}: \text{nearly-healthy } (wlp\text{ b}) \]

\[ \text{and } wp\text{-u-a}: \text{ wp-under-wlp a} \]

\[ \text{and } wp\text{-u-b}: \text{ wp-under-wlp b} \]

**shows** \(wp\text{-under-wlp } (a \text{ ;; } b)\)

**lemma** \(wp\text{-under-wlp-PC}:\)

**assumes** \(h\text{-wp-a}: \text{ healthy } (wp\text{ a})\)
4.7. WELL-DEFINED PROGRAMS.

and h-wlp-a: nearly-healthy (wlp a)
and h-wp-b: healthy (wp b)
and h-wlp-b: nearly-healthy (wlp b)
and wp-u-a: wp-under-wlp a
and wp-u-b: wp-under-wlp b
and uP: unitary P
shows wp-under-wlp (a ⊗ b)
⟨proof⟩

lemma wp-under-wlp-DC:
assumes wp-u-a: wp-under-wlp a
and wp-u-b: wp-under-wlp b
shows wp-under-wlp (a ≔ b)
⟨proof⟩

lemma wp-under-wlp-SetPC:
assumes wp-u-f: \( \forall s \ a \ a \in \supp(P s) \implies \wp-under-wlp(f a) \)
and nP: \( \forall s \ a \ a \in \supp(P s) \implies 0 \leq P s a \)
shows wp-under-wlp (SetPC f P)
⟨proof⟩

lemma wp-under-wlp-SetDC:
assumes wp-u-f: \( \forall s \ a \ a \in S s \implies \wp-under-wlp(f a) \)
and hf: \( \forall s \ a \ a \in S s \implies \healthy(wp(f a)) \)
and nS: \( \forall s \ S s \neq \{\} \)
shows wp-under-wlp (SetDC f S)
⟨proof⟩

lemma wp-under-wlp-Embed:
wp-under-wlp (Embed t)
⟨proof⟩

lemma wp-under-wlp-loop:
fixes body::′s prog
assumes hwp: healthy (wp body)
and hwlp: nearly-healthy (wlp body)
and wp-under: wp-under-wlp body
shows wp-under-wlp (do G → body od)
⟨proof⟩

lemma wp-under-wlp-repeat:
\[ \\begin{array}{c}
\text{healthy (wp a)}; \text{nearly-healthy (wlp a)}; \wp-under-wlp a \\
\implies \wp-under-wlp (\text{repeat n a})
\end{array} \]
⟨proof⟩

lemma wp-under-wlp-Bind:
\[ \\begin{array}{c}
\forall s. \wp-under-wlp (a (f s)) \\
\implies \wp-under-wlp (\text{Bind f a})
\end{array} \]
⟨proof⟩
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 &amp; Q)

lemma sub-distrib-pconjI[intro]:
  \[ [∀ P Q. [ unitary P; unitary Q ] ⇒ wlp prog P && wp prog Q ⊢ wp prog (P &amp; Q) ] \] =⇒
  sub-distrib-pconj prog
⟨proof⟩

lemma sub-distrib-pconjD[dest]:
  ∀ P Q. [ sub-distrib-pconj prog; unitary P; unitary Q ] ⇒
  wlp prog P &amp; wp prog Q ⊢ wp prog (P &amp; Q)
⟨proof⟩

lemma sdp-Abort:
  sub-distrib-pconj Abort
⟨proof⟩

lemma sdp-Skip:
  sub-distrib-pconj Skip
⟨proof⟩

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⟩

lemma sdp-Apply:
  sub-distrib-pconj (Apply f)
⟨proof⟩
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⟩

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 P \oplus b)
⟨proof⟩

lemma sdp-Embed:
   \[ \forall P Q. \forall unitary P: unitary Q \implies t P \&\& t Q \vdash t (P \&\& Q) \] \implies
sub-distrib-pconj (Embed t)
⟨proof⟩

lemma sdp-repeat:
   fixes a :: 's prog
   assumes sdp-a: sub-distrib-pconj a
   and hwp: healthy (wp a) and hwlp: nearly-healthy (wlp a)
   shows sub-distrib-pconj (repeat n a) (\is \ ?X n)
⟨proof⟩

lemma sdp-SetPC:
   fixes p :: 'a \Rightarrow 's prog
   assumes sdp: \(\forall s a. a \in supp (P s) \implies sub-distrib-pconj (p a)\)
   and fin: \(\forall s. finite (supp (P s))\)
   and nnp: \(\forall s a. 0 \leq P s a\)
   and sub: \(\forall s. sum (P s) (supp (P s)) \leq 1\)
   shows sub-distrib-pconj (SetPC p P)
⟨proof⟩

lemma sdp-SetDC:
   fixes p :: 'a \Rightarrow 's prog
   assumes sdp: \(\forall s a. a \in S s \implies sub-distrib-pconj (p a)\)
   and hwp: \(\forall s a. a \in S s \implies healthy (wp (p a))\)
   and hwlp: \(\forall s a. a \in S s \implies nearly-healthy (wlp (p a))\)
   and ne: \(\forall s. S s \neq \{}\)
shows \( \text{sub-distrib-pconj} (\text{SetDC} p S) \)

\(\langle\text{proof}\rangle\)

**lemma** \(\text{sdp-Bind}:\)

\[
\left[ \exists s. \text{sub-distrib-pconj} \left( p \left( f \ s \right) \right) \right] \implies \text{sub-distrib-pconj} \left( \text{Bind} f p \right)
\]

\(\langle\text{proof}\rangle\)

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 body: ‘s prog

assumes \(\text{sdp-body}: \text{sub-distrib-pconj} \text{ body}\)

and \(\text{hwlp}: \text{nearly-healthy} (\text{wlp body})\)

and \(\text{hwp}: \text{healthy} (\text{wp body})\)

shows \(\text{sub-distrib-pconj} \left( \text{do} G \rightarrow \text{body ad} \right)\)

\(\langle\text{proof}\rangle\)

**lemmas** \(\text{sdp-intros} =\)

\(\text{sdp-Abort} \text{ sdp-Skip} \text{ sdp-Apply}\)

\(\text{sdp-Seq} \text{ sdp-DC} \text{ sdp-PC}\)

\(\text{sdp-SetPC} \text{ sdp-SetDC} \text{ sdp-Embed}\)

\(\text{sdp-repeat} \text{ sdp-Bind} \text{ sdp-loop}\)

**4.7.3 The Well-Defined Predicate.**

**definition**

\(\text{well-def} :: \text{’s prog} \Rightarrow \text{bool}\)

where

\(\text{well-def prog} \equiv \text{healthy} (\text{wp prog}) \land \text{nearly-healthy} (\text{wlp prog})\)

\(\land \text{wp-under-wlp prog} \land \text{sub-distrib-pconj prog}\)

\(\land \text{sublinear} (\text{wp prog}) \land \text{bd-cts} (\text{wp prog})\)

**lemma** \(\text{well-defI}[\text{intro}]:\)

\[
\left[ \text{healthy} (\text{wp prog}); \text{nearly-healthy} (\text{wlp prog});
\text{wp-under-wlp prog}; \text{sub-distrib-pconj prog}; \text{sublinear} (\text{wp prog});
\text{bd-cts} (\text{wp prog}) \right] \implies
\text{well-def prog}
\]

\(\langle\text{proof}\rangle\)

**lemma** \(\text{well-def-wp-healthy}[\text{dest}]:\)

\(\text{well-def prog} \implies \text{healthy} (\text{wp prog})\)

\(\langle\text{proof}\rangle\)

**lemma** \(\text{well-def-wlp-nearly-healthy}[\text{dest}]:\)

\(\text{well-def prog} \implies \text{nearly-healthy} (\text{wlp prog})\)

\(\langle\text{proof}\rangle\)

**lemma** \(\text{well-def-wp-under}[\text{dest}]:\)
4.7. WELL-DEFINED PROGRAMS.

well-def prog $\implies$ wp-under-wlp prog 
(proof)

**lemma** well-def-sdp[dest]:
well-def prog $\implies$ sub-distrib-pconj prog 
(proof)

**lemma** well-def-wp-sublinear[dest]:
well-def prog $\implies$ sublinear (wp prog) 
(proof)

**lemma** well-def-wp-cts[dest]:
well-def prog $\implies$ bd-cts (wp prog) 
(proof)

**lemmas** wd-dests =
well-def-wp-healthy well-def-wlp-nearly-healthy
well-def-wp-under well-def-sdp
well-def-wp-sublinear well-def-wp-cts

**lemma** wd-Abort:
well-def Abort 
(proof)

**lemma** wd-Skip:
well-def Skip 
(proof)

**lemma** wd-Apply:
well-def (Apply f) 
(proof)

**lemma** wd-Seq:
[ well-def a; well-def b ] $\implies$ well-def (a ;; b) 
(proof)

**lemma** wd-PC:
[ well-def a; well-def b; unitary P ] $\implies$ well-def (a $\oplus$ b) 
(proof)

**lemma** wd-DC:
[ well-def a; well-def b ] $\implies$ well-def (a $\sqcap$ b) 
(proof)

**lemma** wd-SetDC:
[ $\forall x. x \in S$ $\implies$ well-def (a x); $\forall s. S s \neq \{\}$; $\forall s. finite (S s)$ ] $\implies$ well-def (SetDC a S) 
(proof)
lemma \texttt{wd-SetPC}:\[
\forall x. x \in (\text{supp } (p \ s)) \implies \text{well-def } (a \ x); \ \land s. \text{unitary } (p \ s); \ \land s. \text{finite } (\text{supp } (p \ s)); \\
\land s. \text{sum } (p \ s) (\text{supp } (p \ s)) \leq 1 \implies \text{well-def } (\text{SetPC } a \ p)
\]
⟨proof⟩

lemma \texttt{wd-Embed}:
\textit{fixes} t::\:'s trans
\textit{assumes} \textit{ht}: \textit{healthy } t \ \textit{and} \ st: \textit{sublinear } t \ \textit{and} \ ct: \textit{bd-cts } t
\textit{shows} \textit{well-def } (\textit{Embed } t)
⟨proof⟩

lemma \texttt{wd-repeat}:
\textit{well-def } a \implies \textit{well-def } (\textit{repeat } n \ a)
⟨proof⟩

lemma \texttt{wd-Bind}:
\[ \forall s. \text{well-def } (a \ (f \ s)) \ \implies \text{well-def } (\textit{Bind } f \ a)
\]
⟨proof⟩

lemma \texttt{wd-loop}:
\textit{well-def } body \implies \textit{well-def } (\textit{do } G \rightarrow \textit{body } od)
⟨proof⟩

lemmas \texttt{wd-intros} =
\texttt{wd-Abort} \ \texttt{wd-Skip} \ \texttt{wd-Apply} \\
\texttt{wd-Embed} \ \texttt{wd-Seq} \ \texttt{wd-PC} \\
\texttt{wd-DC} \ \texttt{wd-SetPC} \ \texttt{wd-SetDC} \\
\texttt{wd-Bind} \ \texttt{wd-repeat} \ \texttt{wd-loop}
end

4.8 The Loop Rules

theory \texttt{Loops} imports \texttt{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 \textit{entails} itself, given the loop guard.

\textbf{definition} \texttt{wp-inv} :: \texttt{('s } \Rightarrow \texttt{bool) } \Rightarrow \texttt{('s } \Rightarrow \texttt{real) } \Rightarrow \texttt{bool}
\texttt{where}
\texttt{wp-inv G body I } \leftrightarrow \texttt{(\forall s. } \texttt{«G» s } \ast \texttt{ I s } \leq \texttt{ wp body I s)}
4.8. THE LOOP RULES

Lemma wp-inv1:
\[ \bigwedge I. (\forall s. \langle G \rangle s * I s \leq wp body I s) \Rightarrow wp-inv G body I \]
(proof)

Definition
\[ wp-inv :: (\text{'s} \Rightarrow \text{bool}) \Rightarrow \text{'s prog} \Rightarrow (\text{'s} \Rightarrow \text{real}) \Rightarrow \text{bool} \]
Where
\[ wp-inv G body I \iff (\forall s. \langle G \rangle s * I s \leq wlp body I s) \]

Lemma wp-invD:
\[ wp-inv G body I \Rightarrow \langle \text{proof} \rangle \]

Lemma wp-invI:
\[ \bigwedge I. (\forall s. \langle G \rangle s * I s \leq wlp body I s) \Rightarrow wp-inv G body I \]
(proof)

Lemma wp-invD:
\[ wp-inv G body I \Rightarrow \langle \text{proof} \rangle \]

For standard invariants, the multiplication reduces to conjunction.

Lemma wp-inv-stdD:
\[ \text{assumes inv: wp-inv G body «I»} \]
\[ \text{and hb: healthy (wp body)} \]
\[ \text{shows «G» &«I» \vdash wp body «I»} \]
(proof)

4.8.2 Partial Correctness


Lemma wlp-Loop:
\[ \text{assumes wd: well-def body} \]
\[ \text{and uI: unitary I} \]
\[ \text{and inv: wlp-inv G body I} \]
\[ \text{shows I \leq wlp do G \rightarrow body od (\lambda s. \langle N G \rangle s * I s)} \]
\[ (\text{is I \leq wlp do G \rightarrow body od ?P}) \]
(proof)

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:
\[ \text{assumes wd: well-def body} \]
\[ \text{and inv: wp-inv G body I} \]
\[ \text{and unit: unitary I} \]
\[ \text{shows I & wp (do G \rightarrow body od) (\lambda s. 1) \vdash wp (do G \rightarrow body od) (\lambda s. \langle N G \rangle s * I s)} \]
\[ (\text{is I & & ?T \vdash wp ?loop ?X}) \]
4.8.4 Unfolding

lemma \textit{wp-loop-unfold}:
\begin{align*}
\text{fixes} & \quad \textit{body :: 's prog} \\
\text{assumes} & \quad \textit{sP: sound P} \\
\text{and} & \quad \textit{h: healthy (wp body)} \\
\text{shows} & \quad \textit{wp (do G \rightarrow body od) P} = \\
& \quad (\lambda s. \langle \textit{N} G \rangle s \ast P s + \langle \textit{G} \rangle s \ast \text{wp body (wp (do G \rightarrow body od) P) s}) \\
\end{align*}

(\textit{proof})

lemma \textit{wp-loop-nguard}:
\begin{align*}
[ [ \text{healthy (wp body)}; \text{sound P}; \neg G s ] ] & \Rightarrow \text{wp do G \rightarrow body od P s} = P s \\
\end{align*}

(\textit{proof})

lemma \textit{wp-loop-guard}:
\begin{align*}
[ [ \text{healthy (wp body)}; \text{sound P}; G s ] ] & \Rightarrow \\
& \quad \text{wp do G \rightarrow body od P s} = \text{wp (body ;; do G \rightarrow body od) P s} \\
\end{align*}

(\textit{proof})

end

4.9 The Algebra of pGCL

theory \textit{Algebra imports WellDefined begin}

Programs in pGCL have a rich algebraic structure, largely mirroring that for GCL. We show that programs form a lattice under refinement, with \( a \sqcap b \) and \( a \sqcup b \) as the meet and join operators, respectively. We also take advantage of the algebraic structure to establish a framework for the modular decomposition of proofs.

4.9.1 Program Refinement

Refinement in pGCL relates to refinement in GCL exactly as probabilistic entailment relates to implication. It turns out to have a very similar algebra, the rules of which we establish shortly.

definition \textit{refines :: 's prog \Rightarrow 's prog \Rightarrow bool (infix \sqsubseteq 70)}
where
\begin{align*}
\text{prog} & \sqsubseteq \text{prog'} \equiv \forall P. \text{sound P} \rightarrow \text{wp prog P} \vdash \text{wp prog'} P \\
\end{align*}

lemma \textit{refinesI [intro]}:
\begin{align*}
[ [ \forall P. \text{sound P} \rightarrow \text{wp prog P} \vdash \text{wp prog'} P ] ] & \Rightarrow \text{prog} \sqsubseteq \text{prog'} \\
\end{align*}

(\textit{proof})
4.9. THE ALGEBRA OF PGCL

**Lemma** refinesD[dest]:
\[
\text{[ prog } \sqsubseteq \text{ prog}'; \text{ sound } P ] \implies \text{ wp prog } P \vdash \text{ wp prog} P
\]
\[\text{ (proof)}\]

The equivalence relation below will turn out to be that induced by refinement. It is also the application of `equiv-trans` to the weakest precondition.

**Definition**
pequiv :: 's prog ⇒ 's prog ⇒ bool (infix ≃)

**Where**
\[
\text{prog } \simeq \text{ prog}' \equiv \forall P. \text{ sound } P \implies \text{ wp prog } P = \text{ wp prog} P
\]

**Lemma** pequivI[dest, intro]:
\[
\text{[ } \forall P. \text{ sound } P \implies \text{ wp prog } P = \text{ wp prog} P \text{ ] } \implies \text{ prog } \simeq \text{ prog}'
\]
\[\text{ (proof)}\]

**Lemma** pequivD[dest, simp]:
\[
\text{[ prog } \simeq \text{ prog}', \text{ sound } P \text{ ] } \implies \text{ wp prog } P = \text{ wp prog} P
\]
\[\text{ (proof)}\]

**Lemma** pequiv-equiv-trans:
\[
a \simeq b \iff \text{ equiv-trans (wp a) (wp b)}
\]
\[\text{ (proof)}\]

4.9.2 Simple Identities

The following identities involve only the primitive operations as defined in Section 4.1.1, and refinement as defined above.

**Laws following from the basic arithmetic of the operators separately**

**Lemma** DC-comm[ac-simps]:
\[
a \sqcap b = b \sqcap a
\]
\[\text{ (proof)}\]

**Lemma** DC-assoc[ac-simps]:
\[
a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c
\]
\[\text{ (proof)}\]

**Lemma** DC-idem:
\[
a \sqcap a = a
\]
\[\text{ (proof)}\]

**Lemma** AC-comm[ac-simps]:
\[
a \sqcup b = b \sqcup a
\]
\[\text{ (proof)}\]

**Lemma** AC-assoc[ac-simps]:
\[
\]
\[ a \cup (b \cup c) = (a \cup b) \cup c \]

**Lemma AC-idem:**
\[ a \cup a = a \]

**Lemma PC-quasi-comm:**
\[ a p\oplus b = b (\lambda s. 1 - p s)\oplus a \]

**Lemma AC-idem:**
\[ a p\oplus a = a \]

**Lemma Seq-assoc[ac-simps]:**
\[ A ;; (B ;; C) = A ;; B ;; C \]

**Lemma Abort-refines[intro]:**
well-def \( a \Rightarrow \text{Abort} \subseteq a \)

**Laws relating demonic choice and refinement**

**Lemma left-refines-DC:**
\[ (a \cap b) \subseteq a \]

**Lemma right-refines-DC:**
\[ (a \cap b) \subseteq b \]

**Lemma DC-refines:**
fixes \( a::\text{prog} \) and \( b \) and \( c \)
assumes \( rab: a \subseteq b \) and \( rac: a \subseteq c \)
shows \( a \subseteq (b \cap c) \)

**Lemma DC-mono:**
fixes \( a::\text{prog} \)
assumes \( rab: a \subseteq b \) and \( rcd: c \subseteq d \)
shows \( (a \cap c) \subseteq (b \cap d) \)

**Laws relating angelic choice and refinement**

**Lemma left-refines-AC:**
\[ a \subseteq (a \cup b) \]
4.9. THE ALGEBRA OF PGCL

lemma right-refines-AC:
\[ b \subseteq (a \sqcup b) \]
\(\langle proof\rangle\)

lemma AC-refines:
\[
\text{fixes } a::'s \text{ prog and } b \text{ and } c \\
\text{assumes } rac: a \subseteq c \text{ and } rbc: b \subseteq c \\
\text{shows } (a \sqcup b) \subseteq c
\]
\(\langle proof\rangle\)

lemma AC-mono:
\[
\text{fixes } a::'s \text{ prog} \\
\text{assumes } rab: a \subseteq b \text{ and } rcd: c \subseteq d \\
\text{shows } (a \sqcup c) \subseteq (b \sqcup d)
\]
\(\langle proof\rangle\)

Laws depending on the arithmetic of \( a \oplus b \) and \( a \sqcap b \) together

lemma DC-refines-PC:
\[
\text{assumes } \text{unit}: \text{unitary } p \\
\text{shows } (a \sqcap b) \subseteq (a \oplus b)
\]
\(\langle proof\rangle\)

Laws depending on the arithmetic of \( a \oplus b \) and \( a \sqcup b \) together

lemma PC-refines-AC:
\[
\text{assumes } \text{unit}: \text{unitary } p \\
\text{shows } (a \oplus b) \subseteq (a \sqcup b)
\]
\(\langle proof\rangle\)

Laws depending on the arithmetic of \( a \sqcup b \) and \( a \sqcap b \) together

lemma DC-refines-AC:
\[
(a \sqcap b) \subseteq (a \sqcup b)
\]
\(\langle proof\rangle\)

Laws Involving Refinement and Equivalence

lemma pr-trans[trans]:
\[
\text{fixes } A::'a \text{ prog} \\
\text{assumes } prAB: A \subseteq B \\
\quad \text{and } prBC: B \subseteq C \\
\text{shows } A \subseteq C
\]
\(\langle proof\rangle\)

lemma pequiv-refl[intro,simp]:
\[
a \equiv a
\]
\(\langle proof\rangle\)
LEMMA pequiv-comm[ac-simps]:
\[ a \simeq b \iff b \simeq a \]
(proof)

LEMMA pequiv-pr[dest]:
\[ a \simeq b \implies a \sqsubseteq b \]
(proof)

LEMMA pequiv-trans[intro,trans]:
\[ \begin{cases} a \simeq b; b \simeq c \end{cases} \implies a \simeq c \]
(proof)

LEMMA pequiv-pr-trans[intro,trans]:
\[ \begin{cases} a \simeq b; b \sqsubseteq c \end{cases} \implies a \sqsubseteq c \]
(proof)

LEMMA pr-pequiv-trans[intro,trans]:
\[ \begin{cases} a \sqsubseteq b; b \simeq c \end{cases} \implies a \simeq c \]
(proof)

Refinement induces equivalence by antisymmetry:

LEMMA pequiv-antisym:
\[ \begin{cases} a \sqsubseteq b; b \sqsubseteq a \end{cases} \implies a \simeq b \]
(proof)

LEMMA pequiv-DC:
\[ \begin{cases} a \simeq c; b \simeq d \end{cases} \implies (a \sqcap b) \simeq (c \sqcap d) \]
(proof)

LEMMA pequiv-AC:
\[ \begin{cases} a \simeq c; b \simeq d \end{cases} \implies (a \sqcup b) \simeq (c \sqcup d) \]
(proof)

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)
and wa: well-def a
and wb: well-def b
and dr: a \sqsubseteq b
shows a \simeq b

Proof by contradiction.
(proof)
4.9. THE ALGEBRA OF PGCL

4.9.4 The Algebraic Structure of Refinement

Well-defined programs form a half-bounded semilattice under refinement, where *Abort* is bottom, and \( a \sqcap b \) is \( \inf \). There is no unique top element, but all fully-deterministic programs are maximal.

The type that we construct here is not especially useful, but serves as a convenient way to express this result.

\[
\text{quotient-type } 's \text{ program } = \\
's \text{ prog } / \text{ partial } : \lambda a b.\ a \simeq b \land \text{well-def } a \land \text{well-def } b
\]

\[
\text{instantiation program :: (type) semilattice-inf begin} \\
\text{lift-definition} \\
\text{less-eq-program :: 'a program } \Rightarrow \ 'a \text{ program } \Rightarrow \text{ bool is refines} \\
\text{(proof)}
\]

\[
\text{lift-definition} \\
\text{less-program :: 'a program } \Rightarrow \ 'a \text{ program } \Rightarrow \text{ bool is } \lambda a b.\ a \sqsubseteq b \land \neg b \sqsubseteq a \\
\text{(proof)}
\]

\[
\text{lift-definition} \\
\text{inf-program :: 'a program } \Rightarrow \ 'a \text{ program } \Rightarrow \ 'a \text{ program is } \text{DC} \\
\text{(proof)}
\]

\[
\text{instance (proof)}
\text{end}
\]

\[
\text{instantiation program :: (type) bot begin} \\
\text{lift-definition} \\
\text{bot-program :: 'a program is Abort} \\
\text{(proof)}
\]

\[
\text{instance (proof)}
\text{end}
\]

\[
\text{lemma eq-det: } \land a b.'s \text{ prog. } a \simeq b; \text{ determ } (wp \ a) ] \Rightarrow \text{ determ } (wp \ b) \\
\text{ (proof)}
\]

\[
\text{lift-definition} \\
\text{pdeterm :: 's program } \Rightarrow \text{ bool is } \lambda a.\ \text{determ } (wp \ a) \\
\text{(proof)}
\]

\[
\text{lemma determ-maximal:} \\
[ \text{pdeterm } a; a \leq x ] \Rightarrow a = x \\
\text{(proof)}
\]
4.9.5 Data Refinement

A projective data refinement construction for pGCL. By projective, we mean that the abstract state is always a function ($\phi$) of the concrete state. Refinement may be predicated ($G$) on the state.

**definition**

$\text{drefines} :: (\mathbf{b} \Rightarrow \mathbf{a}) \Rightarrow \mathbf{a} \propto \mathbf{b} \propto \mathbf{bool}$

**where**

$\text{drefines} \ G \ A \ B \equiv \forall P Q. (\text{unitary} \ P \land \text{unitary} \ Q \land (P \vdash \wp A \ Q)) \rightarrow (\langle G \rangle \land \langle P \circ \phi \rangle) \vdash \wp B \ (Q \circ \phi))$

**lemma** $\text{drefinesD}[^{\text{dest}}]$:

\[
\begin{align*}
\text{assumes} & \ : \text{drefines} \ \varphi \ G \ A \ B \\
& \text{and} \ uP : \text{unitary} \ P \\
& \text{and} \ uQ : \text{unitary} \ Q \\
& \text{and} \ wpA : P \vdash \wp A \ Q \\
& \text{and} \ G : G \ s \\
\text{shows} & \ (P \circ \phi) \ s \leq \wp B \ (Q \circ \phi) \ s
\end{align*}
\]

\langle proof \rangle

We can alternatively use $G$ as an assumption:

**lemma** $\text{drefinesD2}$:

\[
\begin{align*}
\text{assumes} & \ : \text{drefines} \ \varphi \ G \ A \ B \\
& \text{and} \ G : G \ s \\
& \text{and} \ uQ : \text{unitary} \ Q \\
& \text{and} \ wa : \text{well-def} \ a \\
\text{shows} & \ \wp a \ Q \ (\varphi \ s) \leq \wp b \ (Q \circ \phi) \ s
\end{align*}
\]

\langle proof \rangle

This additional form is sometimes useful:

**lemma** $\text{drefinesD3}$:

\[
\begin{align*}
\text{assumes} & \ : \text{drefines} \ \varphi \ G \ a \ b \\
& \text{and} \ G : G \ s \\
& \text{and} \ uQ : \text{unitary} \ Q \\
& \text{and} \ wa : \text{well-def} \ a \\
\text{shows} & \ \wp a \ Q \ (\varphi \ s) \leq \wp b \ (Q \circ \phi) \ s
\end{align*}
\]

\langle proof \rangle

**lemma** $\text{drefinesI}[^{\text{intro}}]$:

\[
\begin{align*}
\text{assumes} & \ : \text{drefines} \ \varphi \ G \ A \ B \\
& \text{and} \ G : G \ s \\
\text{shows} & \ drefines \ \varphi \ G \ A \ B
\end{align*}
\]

\langle proof \rangle

Use $G$ as an assumption, when showing refinement:

**lemma** $\text{drefinesI}2$:

\[
\begin{align*}
\text{fixes} & \ : A :: \mathbf{a} \ \text{prog} \\
& \text{and} \ B :: \mathbf{b} \ \text{prog} \\
& \text{and} \ \varphi :: \mathbf{b} \Rightarrow \mathbf{a} \\
& \text{and} \ G :: \mathbf{b} \Rightarrow \mathbf{bool} \\
\text{assumes} & \ : \text{well-def} \ B
\end{align*}
\]
and withAs:
\[ P Q s. \begin{array}{l}
(\text{\textit{unitary } P; \text{\textit{unitary } Q;}})
\end{array}
\]
\[ G s; P \vdash wp A Q \] \[ \implies (P o \varphi) s \leq wp B (Q o \varphi) s \]

shows \textit{drefines } \varphi \ G A B

(proof)

\textbf{lemma dr-strengthen-guard:}

\textbf{fixes} a::'s prog \textbf{and} b::'t prog

\textbf{assumes} fg: \[ \forall s. F s \implies G s \]

\textbf{and} drab: \textit{drefines } \varphi \ G a b

\textbf{shows} \textit{drefines } \varphi \ F a b

(proof)

Probabilistic correspondence, \textit{pcorres}, is equality on distribution transformers, modulo a guard. It is the analogue, for data refinement, of program equivalence for program refinement.

\textbf{definition}

\textit{pcorres} :: (\[ \forall s. F s \implies \] G s) \rightarrow (wp A Q o \varphi) = (wp B (Q o \varphi))

\textbf{lemma pcorresI:}

\[ \forall Q. \text{\textit{unitary } Q \implies (wp A Q o \varphi) = wp B (Q o \varphi)} \]

\textbf{lemma pcorresD:}

\[ \forall Q. \text{\textit{unitary } Q \implies wp A Q o \varphi = wp B (Q o \varphi)} \]

(proof)

Often easier to use, as it allows one to assume the precondition.

\textbf{lemma pcorresI2}[intro]:

\textbf{fixes} A::'a prog \textbf{and} B::'b prog

\textbf{assumes} withG: \[ \forall s. \text{\textit{unitary } Q; G s} \] \[ \implies wp A Q (\varphi s) = wp B (Q o \varphi) s \]

\textbf{and} wA: well-def A

\textbf{and} wB: well-def B

\textbf{shows} \textit{pcorres } \varphi \ G A B

(proof)

\textbf{lemma pcorresD2:}

\[ \forall Q. \text{\textit{unitary } Q \implies wp A Q (\varphi s) = wp B (Q o \varphi) s} \]

(proof)
4.9.6 The Algebra of Data Refinement

Program refinement implies a trivial data refinement:

**lemma** *refines-drefines*:

```plaintext
fixes \( a : 's \) prog
assumes \( \text{rab: } a \subseteq b \) and \( \text{wb: well-def } b \)
shows drefines \((\lambda s. s) G a b\)
```

**proof**

Data refinement is transitive:

**lemma** *dr-trans*[trans]:

```plaintext
fixes \( A : 'a \) prog and \( B : 'b \) prog and \( C : 'c \) prog
assumes \( \text{drAB: } \text{drefines } \varphi G A B \)
and \( \text{drBC: } \text{drefines } \varphi' G' B C \)
and \( \text{Gimp: } \forall s. G' s \Rightarrow G (\varphi' s) \)
shows drefines \((\varphi \circ \varphi') G' A C\)
```

**proof**

Data refinement composes with program refinement:

**lemma** *pr-dr-trans*[trans]:

```plaintext
assumes \( \text{prAB: } A \subseteq B \)
and \( \text{drBC: } \text{drefines } \varphi G B C \)
shows drefines \((\varphi \circ \varphi') G A C\)
```

**proof**

**lemma** *dr-pr-trans*[trans]:

```plaintext
assumes \( \text{drAB: } \text{drefines } \varphi G A B \)
assumes \( \text{prBC: } B \sqsubseteq C \)
shows drefines \((\varphi \circ \varphi') G A C\)
```

**proof**

If the projection \( \varphi \) commutes with the transformer, then data refinement is reflexive:

**lemma** *dr-refl*:

```plaintext
assumes \( \text{wa: well-def } a \)
and \( \text{comm: } \forall Q. \text{unitary } Q \Rightarrow wp a Q o \varphi \vdash wp a (Q o \varphi) \)
shows drefines \(\varphi G a a\)
```

**proof**

Correspondence implies data refinement

**lemma** *pcorres-drefine*:

```plaintext
assumes \( \text{corres: pcorres } \varphi G A C \)
and \( \text{wC: well-def } C \)
shows drefines \(\varphi G A C\)
```

**proof**

Any *data* refinement of a deterministic program is correspondence. This is the analogous result to that relating program refinement and equivalence.
4.9. THE ALGEBRA OF PGCL

**Lemma** \( \text{drefines-determ} \):

- **Fixes** \( a::'a \text{ prog and } b::'b \text{ prog} \)
- **Assumes** \( da: \text{ determ (wp a)} \) and \( wa: \text{ well-def a} \)
- and \( wb: \text{ well-def b} \) and \( dr: \text{ drefines } \varphi \ G \ a \ b \)
- **Shows** \( p\text{corres } \varphi \ G \ a \ b \)

The proof follows exactly the same form as that for program refinement: Assuming that correspondence doesn’t hold, we show that \( \text{wp} \ b \) is not feasible, and thus not healthy, contradicting the assumption.

\[ \langle \text{proof} \rangle \]

### 4.9.7 Structural Rules for Correspondence

**Lemma** \( \text{p\text{corres-Skip}} \):

\( \text{p\text{corres } \varphi \ G \ \text{Skip} \ \text{Skip}} \)

\[ \langle \text{proof} \rangle \]

Correspondence composes over sequential composition.

**Lemma** \( \text{p\text{corres-Seq}} \):

- **Fixes** \( A::'b \text{ prog and } B::'c \text{ prog} \) and \( C::'b \text{ prog and } D::'c \text{ prog} \) and \( \varphi::'c \Rightarrow 'b \)
- **Assumes** \( \text{p\text{cAB: p\text{corres } \varphi \ G A B}} \) and \( \text{p\text{cCD: p\text{corres } \varphi \ H C D}} \) and \( \text{wA: \text{ well-def A and } wB: \text{ well-def B}} \) and \( \text{wC: \text{ well-def C and } wD: \text{ well-def D}} \) and \( p\text{3p2: } \bigwedge Q. \text{ unitary } Q \implies \lvert I \rvert \&\& \text{wp B Q} = \text{wp B} \ (\lvert H \rvert \&\& Q) \) and \( p\text{1p3: } \bigwedge s. \ G \ s \implies I \ s \)
- **Shows** \( \text{p\text{corres } \varphi \ G (A;;C) (B;;D}} \)

\[ \langle \text{proof} \rangle \]

### 4.9.8 Structural Rules for Data Refinement

**Lemma** \( \text{dr-Skip} \):

- **Fixes** \( \varphi::'c \Rightarrow 'b \)
- **Shows** \( \text{d\text{refines } \varphi \ G \ \text{Skip} \ \text{Skip}} \)

\[ \langle \text{proof} \rangle \]

**Lemma** \( \text{dr-Abort} \):

- **Fixes** \( \varphi::'c \Rightarrow 'b \)
- **Shows** \( \text{d\text{refines } \varphi \ G \ \text{Abort} \ \text{Abort}} \)

\[ \langle \text{proof} \rangle \]

**Lemma** \( \text{dr-Apply} \):

- **Fixes** \( \varphi::'c \Rightarrow 'b \)
- **Assumes** \( \text{commutes: } f \circ \varphi = \varphi \circ g \)
- **Shows** \( \text{d\text{refines } \varphi \ G \ \{\text{Apply } f\} \ \{\text{Apply } g\}} \)
lemma \textit{dr-Seq}:  
assumes \textit{drAB}: \textit{drefines }\varphi \ P \ A \ B  
and \textit{drBC}: \textit{drefines }\varphi \ Q \ C \ D  
and wpB: \ «P» \not\vdash \ wp \ B \ «Q»  
and wB: \ well-def B  
and wC: \ well-def C  
and wD: \ well-def D  
shows \textit{drefines }\varphi \ P \ (A ;; C) \ (B ;; D)  
⟨\textit{proof}\rangle

lemma \textit{dr-repeat}:  
fixes \varphi :: \ 'a \Rightarrow \ 'b  
assumes \textit{dr-ab}: \textit{drefines }\varphi \ G \ a \ b  
and Gpr: \ «G» \not\vdash \ wp \ b \ «G»  
and wa: \ well-def a  
and wb: \ well-def b  
shows \textit{drefines }\varphi \ G \ (\text{repeat } n \ a) \ (\text{repeat } n \ b) \ (\text{is } ?X \ n)  
⟨\textit{proof}\rangle

end

4.10 Structured Reasoning

theory \textit{StructuredReasoning} imports \textit{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 \textit{wp-Abort}:  
(\lambda s. 0) \not\vdash \ wp \ Abort \ Q  
⟨\textit{proof}\rangle

lemma \textit{wlp-Abort}:  
(\lambda s. 1) \not\vdash \ wlp \ Abort \ Q  
⟨\textit{proof}\rangle

lemma \textit{wp-Skip}:  
P \not\vdash \ wp \ Skip \ P  
⟨\textit{proof}\rangle

lemma \textit{wlp-Skip}:  
P \not\vdash \ wlp \ Skip \ P
4.10.  STRUCTURED REASONING

⟨proof⟩

**Lemma** **wp-Apply:**

\[ Q \circ f \vdash wp (Apply f) \ Q \]

⟨proof⟩

**Lemma** **wlp-Apply:**

\[ Q \circ f \vdash wlp (Apply f) \ Q \]

⟨proof⟩

**Lemma** **wp-Seq:**

assumes

\begin{align*}
\text{ent-a: } & P \vdash \neg wp a Q \\
\text{and ent-b: } & Q \vdash \neg wp b R \\
\text{and wa: } & \text{well-def } a \\
\text{and wb: } & \text{well-def } b \\
\text{and s-Q: } & \text{sound } Q \\
\text{and s-R: } & \text{sound } R
\end{align*}

shows

\[ P \vdash \neg wp (a ;; b) R \]

⟨proof⟩

**Lemma** **wlp-Seq:**

assumes

\begin{align*}
\text{ent-a: } & P \vdash \neg wlp a Q \\
\text{and ent-b: } & Q \vdash \neg wlp b R \\
\text{and wa: } & \text{well-def } a \\
\text{and wb: } & \text{well-def } b \\
\text{and u-Q: } & \text{unitary } Q \\
\text{and u-R: } & \text{unitary } R
\end{align*}

shows

\[ P \vdash \neg wlp (a ;; b) R \]

⟨proof⟩

**Lemma** **wp-PC:**

\[ (\lambda s. P s \star wp a Q s + (1 - P s) \star wp b Q s) \vdash wp (a \oplus b) Q \]

⟨proof⟩

**Lemma** **wlp-PC:**

\[ (\lambda s. P s \star wlp a Q s + (1 - P s) \star wlp b Q s) \vdash wlp (a \oplus b) Q \]

⟨proof⟩

A simpler rule for when the probability does not depend on the state.

**Lemma** **PC-fixed:**

assumes

\begin{align*}
\text{wp: } & P \vdash a \ ab R \\
\text{and wpb: } & Q \vdash b \ ab R \\
\text{and np: } & 0 \leq p \text{ and bp: } p \leq 1
\end{align*}

shows

\[ (\lambda s. p \star P s + (1 - p) \star Q s) \vdash (a \ (\lambda s. p) \oplus b) \ ab R \]

⟨proof⟩

**Lemma** **wp-PC-fixed:**

\[
[ P \vdash wp a R; Q \vdash wp b R; 0 \leq p; p \leq 1 ] \implies (\lambda s. p \star P s + (1 - p) \star Q s) \vdash wp (a \ (\lambda s. p) \oplus b) R
\]
chapter 4. The PGCL language

proof

lemma wlp-PC-fixed:
\[
\begin{align*}
& \text{fixes } a::\text{prog and } b \\
& \text{assumes } \text{wpa: } \text{P } \vdash \text{wlp a R} \\
& \quad \text{and } \text{wpb: } \text{Q } \vdash \text{wlp b R} \\
& \text{shows } (\lambda s. \text{min (P s) (Q s)}) \vdash \text{wlp } (\lambda s. p) \oplus b \text{ R} \\
\end{align*}
\]
(proof)

lemma wp-DC:
\[
(\lambda s. \text{min (wp a Q s) (wp b Q s)}) \vdash \text{wp (a } \bigcap b \text{) Q}
\]
(proof)

lemma wlp-DC:
\[
\begin{align*}
& \text{fixes } a::\text{prog and } b \\
& \text{assumes } \text{wpa: } \text{P } \vdash \text{wp a R} \\
& \quad \text{and } \text{wpb: } \text{Q } \vdash \text{wp b R} \\
& \text{shows } (\lambda s. \text{min (P s) (Q s)}) \vdash \text{wp (a } \bigcap b \text{) R} \\
\end{align*}
\]
(proof)

Combining annotations for both branches:

lemma DC-split:
\[
\begin{align*}
& \text{fixes } a::\text{prog and } b \\
& \text{assumes } \text{wpa: } \text{P } \vdash \text{wp a R} \\
& \quad \text{and } \text{wpb: } \text{Q } \vdash \text{wp b R} \\
& \text{shows } (\lambda s. \text{min (P s) (Q s)}) \vdash \text{wp (a } \bigcap b \text{) R} \\
\end{align*}
\]
(proof)

lemma wp-DC-split:
\[
\begin{align*}
& \text{fixes } a::\text{prog and } b \\
& \text{assumes } \text{wpa: } \text{P } \vdash \text{wp a R} \\
& \quad \text{and } \text{wpb: } \text{Q } \vdash \text{wp b R} \\
& \text{shows } (\lambda s. \text{min (P s) (Q s)}) \vdash \text{wp (a } \bigcap b \text{) R} \\
\end{align*}
\]
(proof)

lemma wlp-DC-split:
\[
\begin{align*}
& \text{fixes } a::\text{prog and } b \\
& \text{assumes } \text{wpa: } \text{P } \vdash \text{wlp a R} \\
& \quad \text{and } \text{wpb: } \text{Q } \vdash \text{wlp b R} \\
& \text{shows } (\lambda s. \text{min (P s) (Q s)}) \vdash \text{wlp (a } \bigcap b \text{) R} \\
\end{align*}
\]
(proof)

lemma wp-DC-split-same:
\[
\begin{align*}
& \text{fixes } a::\text{prog and } b \\
& \text{assumes } \text{wpa: } \text{P } \vdash \text{wlp a R} \\
& \quad \text{and } \text{wpb: } \text{Q } \vdash \text{wlp b R} \\
& \text{shows } (\lambda s. \text{min (P s) (Q s)}) \vdash \text{wlp (a } \bigcap b \text{) R} \\
\end{align*}
\]
(proof)

lemma SetPC-split:
\[
\begin{align*}
& \text{fixes } f::x \Rightarrow y \text{ prog and } p::y \Rightarrow x \Rightarrow \text{real} \\
& \text{assumes } \text{rec: } \bigwedge x s. x \in \text{supp (p s)} \Rightarrow P x \vdash f x \text{ ab Q} \\
& \quad \text{and } \text{nnp: } \bigwedge x. \text{nneg (p s)} \\
& \text{shows } (\lambda s. \sum x \in \text{supp (p s)}. p s x \ast P x s) \vdash \text{SetPC f p ab Q} \\
\end{align*}
\]
(proof)
4.10. STRUCTURED REASONING

**Lemma wp-SetPC-split:**
\[ \forall s, x \in \text{supp}(p) \implies \forall s. x \in \text{supp}(p) \implies P \vdash \wp(f) \quad \forall s, n\neg(p) \implies (\lambda s. \sum x \in \text{supp}(p). p x * P x) \vdash \wp(\text{SetPC } f p) \]

(proof)

**Lemma wlp-SetPC-split:**
\[ \forall s, x \in \text{supp}(p) \implies \forall s. x \in \text{supp}(p) \implies P \vdash \wlp(f) \quad \forall s, n\neg(p) \implies (\lambda s. \sum x \in \text{supp}(p). p x * P x) \vdash \wlp(\text{SetPC } f p) \]

(proof)

**Lemma wp-SetDC-split:**
\[ \forall s, x \in S \implies \forall s. x \in S \implies P \vdash \wp(f) \quad \forall s, S \neq \{\} \implies (\lambda s. \text{Inf}((\lambda x. P x s) \cdot S)) \vdash \wp(\text{SetDC } f S) \]

(proof)

**Lemma wlp-SetDC-split:**
\[ \forall s, x \in S \implies \forall s. x \in S \implies P \vdash \wlp(f) \quad \forall s, S \neq \{\} \implies (\lambda s. \text{Inf}((\lambda x. P x s) \cdot S)) \vdash \wlp(\text{SetDC } f S) \]

(proof)

**Lemma wp-SetDC:**
assumes \( wp: \forall s, x \in S \implies P \vdash \wp(f) \)
and \( nc: \forall s, S \neq \{\} \)
shows \( (\lambda s. \text{Inf}((\lambda x. P x s) \cdot S)) \vdash \wp(\text{SetDC } f S) \)

(proof)

**Lemma wlp-SetDC:**
assumes \( wp: \forall s, x \in S \implies P \vdash \wlp(f) \)
and \( nc: \forall s, S \neq \{\} \)
and \( sP: \forall x, \text{sound}(P) \)
shows \( (\lambda s. \text{Inf}((\lambda x. P x s) \cdot S)) \vdash \wlp(\text{SetDC } f S) \)

(proof)

**Lemma wp-Embed:**
\( P \vdash t Q \implies P \vdash \wp(\text{Embed } t) \)

(proof)

**Lemma wlp-Embed:**
\( P \vdash t Q \implies P \vdash \wlp(\text{Embed } t) \)

(proof)

**Lemma wp-Bind:**
\[ [ \forall s. P s \leq \wp(a(f)) \implies P \vdash (\text{Bind } f a) ] \]

(proof)

**Lemma wlp-Bind:**
\[ [ \forall s. P s \leq \wlp(a(f)) \implies P \vdash (\text{Bind } f a) ] \]

(proof)
lemma wp-repeat:
\[
\begin{align*}
P \vdash \wp a \ Q; & \quad Q \vdash \wp (\text{repeat } n \ a) \ R; \\
\text{well-def } a; & \quad \text{sound } Q; \quad \text{sound } R \quad \Rightarrow \quad P \vdash \wp (\text{repeat } (\text{Suc } n) \ a) \ R
\end{align*}
\]
\langle proof \rangle

lemma wlp-repeat:
\[
\begin{align*}
P \vdash \wlp a \ Q; & \quad Q \vdash \wlp (\text{repeat } n \ a) \ R; \\
\text{well-def } a; & \quad \text{unitary } Q; \quad \text{unitary } R \quad \Rightarrow \quad P \vdash \wlp (\text{repeat } (\text{Suc } n) \ a) \ R
\end{align*}
\]
\langle proof \rangle

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=
\[
\begin{align*}
\text{entails-strengthen-post}[\text{where } t=\wp a \ \text{for } a]
\end{align*}
\]

lemma wlp-strengthen-post:
\[
\begin{align*}
P \vdash \wlp a \ Q \Rightarrow \text{nearly-healthy } (\wlp a) \Rightarrow \text{unitary } R \Rightarrow Q \vdash R \Rightarrow \text{unitary } Q \\
\Rightarrow \quad P \vdash \wlp a \ R
\end{align*}
\]
\langle proof \rangle

lemmas wp-weaken-pre=
\[
\begin{align*}
\text{entails-weaken-pre}[\text{where } t=\wp a \ \text{for } a]
\end{align*}
\]

lemmas wlp-weaken-pre=
\[
\begin{align*}
\text{entails-weaken-pre}[\text{where } t=\wlp a \ \text{for } a]
\end{align*}
\]

lemmas wp-scale=
\[
\begin{align*}
\text{entails-scale}[\text{where } t=\wp a \ \text{for } a, \ \text{OF - well-def-\wp-healthy}]
\end{align*}
\]

4.10.2 Algebraic Decomposition

Refinement is a powerful tool for decomposition, belied by the simplicity of the rule. This is an axiomatic formulation of refinement (all annotations of the \(a\) are annotations of \(b\)), rather than an operational version (all traces of \(b\) are traces of \(a\)).

lemma wp-refines:
\[
\begin{align*}
[ a \sqsubseteq b; & \quad P \vdash \wp a \ Q; \quad \text{sound } Q \quad \Rightarrow \quad P \vdash \wp b \ Q
\end{align*}
\]
\langle proof \rangle

lemmas wp-drefines = drefinesD
4.10.3 Hoare triples

The Hoare triple, or validity predicate, is logically equivalent to the weakest-precondition entailment form. The benefit is that it allows us to define transitivity rules for computational (also/finally) reasoning.

**definition**

\[
\text{wp-valid} :: (\{a \Rightarrow \text{real}\} \Rightarrow (\{a \Rightarrow \text{real}\} \Rightarrow \text{bool} (\{\cdot\} - \{\cdot\}) p)
\]

**where**

\[
\text{wp-valid} P \text{ prog } Q \equiv P \vdash \text{ wp prog } Q
\]

**lemma** \text{wp-validI}:

\[
P \vdash \text{ wp prog } Q \implies \{P\} \text{ prog } \{Q\} p
\]

\langle proof \rangle

**lemma** \text{wp-validD}:

\[
\{P\} \text{ prog } \{Q\} p \implies P \vdash \text{ wp prog } Q
\]

\langle proof \rangle

**lemma** \text{valid-Seq}:

\[
\begin{bmatrix}
\{P\} a \{Q\} p; \{Q\} b \{R\} p; \text{ well-def } a; \text{ well-def } b; \text{ sound } Q; \text{ sound } R
\end{bmatrix} \implies
\{P\} a \{R\} p
\]

\langle proof \rangle

We make it available to the computational reasoner:

**declare** \text{valid-Seq[trans]}

end

4.11 Loop Termination

**theory** \text{Termination} \text{ 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** \text{maximal-Seq-term}:

\text{fixes } r::'s \text{ prog and } s::'s \text{ prog}

\text{assumes } mr: \text{ maximal } (\text{ wp } r)

\text{and } ws: \text{ well-def } s

\text{and } ts: (\lambda s. 1) \vdash \text{ wp } s (\lambda s. 1)

\text{shows } (\lambda s. 1) \vdash \text{ wp } (r ;; s) (\lambda s. 1)

\langle proof \rangle
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** \( \langle N \ R \rangle \vdash \text{wp G \rightarrow body od (\lambda s. 1)} \)

\langle proof \rangle

### 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: \( \forall s. m s = 0 \rightarrow \neg G s \)  
  and variant: \( \forall n. \langle \lambda s. m s = Suc n \rangle \vdash \text{wp body } \langle \lambda s. m s = n \rangle \)  
  **shows** \( \lambda s. 1 \vdash \text{wp G \rightarrow body od (\lambda s. 1)} \)

\langle proof \rangle

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** wb: well-def body  
  and guard: \( \forall s. m s = 0 \rightarrow \neg G s \)  
  and variant: \( \forall n. \langle \lambda s. m s = Suc n \rangle \& \langle I \rangle \vdash \text{wp body } \langle \lambda s. m s = n \rangle \)  
  and inv: \( \text{wp-inv G body } \langle I \rangle \)  
  **shows** \( \langle I \rangle \vdash \text{wp G \rightarrow body od (\lambda s. 1)} \)

\langle proof \rangle

### 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: \( \langle \lambda s. p \rangle \vdash \text{wp body } \langle N \ R \rangle \)  
  and nzp: \( 0 < p \)  
  — The body is maximal i.e. it terminates absolutely.  
  and mb: \( \text{maximal (wp body)} \)  
  **shows** \( \langle \lambda s. 1 \rangle \vdash \text{wp G \rightarrow body od (\lambda s. 1)} \)

\langle proof \rangle
4.12 Automated Reasoning

theory Automation imports StructuredReasoning
begin

This theory serves as a container for automated reasoning tactics for pGCL, implemented in ML. At present, there is a basic verification condition generator (VCG).

named-theorems wd
  theorems to automatically establish well-definedness
named-theorems pwp-core
  core probabilistic wp rules, for evaluating primitive terms
named-theorems pwp
  user-supplied probabilistic wp rules
named-theorems wlp
  user-supplied probabilistic wlp rules

⟨ML⟩

declare wd-intros[wd]

lemmas core-wp-rules =
  wp-Skip     wlp-Skip
  wp-Abort    wlp-Abort
  wp-Apply    wlp-Apply
  wp-Seq      wlp-Seq
  wp-DC-split wlp-DC-split
  wp-PC-fixed wlp-PC-fixed
  wp-SetDC    wlp-SetDC
  wp-SetPC-split wlp-SetPC-split

declare core-wp-rules[pwp-core]

end
4.13 Miscellaneous Mathematics

theory Misc
imports "HOL-Analysis.Multivariate-Analysis"
begin

lemma sum-UNIV:
  fixes S :: 'a::finite set
  assumes complete: \( \forall x. x \notin S \Rightarrow f x = 0 \)
  shows \( \sum f S = \sum f \text{UNIV} \)
⟨proof⟩

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⟩

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⟩

lemma mult-div-mono-left:
  fixes c :: real
  assumes nnc: \( 0 \leq c \) and nzc: \( c \neq 0 \)
  and inv: \( a \leq \text{inverse} c \cdot b \)
  shows \( c \cdot a \leq b \)
⟨proof⟩

lemma mult-div-mono-right:
  fixes c :: real
  assumes nnc: \( 0 \leq c \) and nzc: \( c \neq 0 \)
  and inv: \( \text{inverse} c \cdot a \leq b \)
  shows \( a \leq c \cdot b \)
⟨proof⟩

end
lemma min-distrib:
  fixes c::real
  assumes nnc: \( 0 \leq c \)
  shows \( c \cdot \min a b = \min (c \cdot a) (c \cdot b) \)
⟨proof⟩

lemma finite-set-least:
  fixes S::'a:linorder set
  assumes finite: finite S
      and ne: \( S \neq \{\} \)
  shows \( \exists x \in S. \forall y \in S. x \leq y \)
⟨proof⟩

lemma cSup-add:
  fixes c::real
  assumes ne: \( S \neq \{\} \)
      and bS: \( \forall x. x \in S \implies x \leq b \)
  shows \( \Sup S + c = \Sup \{x + c \mid x \in S\} \)
⟨proof⟩

lemma cSup-mult:
  fixes c::real
  assumes ne: \( S \neq \{\} \)
      and bS: \( \forall x. x \in S \implies x \leq b \)
      and nnc: \( 0 \leq c \)
  shows \( c \cdot \Sup S = \Sup \{c \cdot x \mid x \in S\} \)
⟨proof⟩

lemma closure-contains-Sup:
  fixes S :: real set
  assumes neS: \( S \neq \{\} \) and bS: \( \forall x \in S. x \leq B \)
  shows \( \Sup S \in \text{closure } S \)
⟨proof⟩

lemma tendsto-min:
  fixes x y::real
  assumes ta: \( a \longrightarrow x \)
      and tb: \( b \longrightarrow y \)
  shows \( (\lambda i. \min (a i) (b i)) \longrightarrow \min x y \)
⟨proof⟩

definition supp :: (\'s \Rightarrow \'real) \Rightarrow \'s set
where \( \text{supp } f = \{x. f x \neq 0\} \)

definition dist-remove :: (\'s \Rightarrow \'real) \Rightarrow \'s \Rightarrow \'s \Rightarrow \real
where \( \text{dist-remove } p x = (\lambda y. \text{if } y=x \text{ then } 0 \text{ else } p y / (1 - p x)) \)

lemma supp-dist-remove:
4.13. MISCELLANEOUS MATHEMATICS

\[ p \ x \neq 0 \implies p \ x \neq 1 \implies \text{supp} \ (\text{dist-remove} \ p \ x) = \text{supp} \ p - \{x\} \]

⟨proof⟩

**Lemma supp-empty:**
\[ \text{supp} \ f = \{\} \implies f \ x = 0 \]
⟨proof⟩

**Lemma nsupp-zero:**
\[ x \notin \text{supp} \ f \implies f \ x = 0 \]
⟨proof⟩

**Lemma sum-supp:**
\[ \text{fixes} \ f :: \alpha :: \text{finite} \Rightarrow \text{real} \]
\[ \text{shows} \ \sum f (\text{supp} f) = \sum f \ \text{UNIV} \]
⟨proof⟩

### 4.13.1 Truncated Subtraction

definition
\[ \text{tminus} :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real} \ (\text{infixl} \odot 60) \]

where
\[ x \odot y = \max (x - y) \ 0 \]

**Lemma minus-le-tminus[intro,simp]:**
\[ a - b \leq a \odot b \]
⟨proof⟩

**Lemma tminus-cancel-1:**
\[ 0 \leq a \implies a + 1 \odot 1 = a \]
⟨proof⟩

**Lemma tminus-zero-imp-le:**
\[ x \odot y \leq 0 \implies x \leq y \]
⟨proof⟩

**Lemma tminus-zero[simp]:**
\[ 0 \leq x \implies x \odot 0 = x \]
⟨proof⟩

**Lemma tminus-left-mono:**
\[ a \leq b \implies a \odot c \leq b \odot c \]
⟨proof⟩

**Lemma tminus-less:**
\[ [0 \leq a; 0 \leq b] \implies a \odot b \leq a \]
⟨proof⟩

**Lemma tminus-left-distrib:**
\[ \text{assumes} \ \text{nna} : 0 \leq a \]
shows $a * (b ⊕ c) = a * b ⊕ a * c$
⟨proof⟩

lemma tminus-le[simp]:
$b ≤ a \implies a ⊕ b = a - b$
⟨proof⟩

lemma tminus-le-alt[simp]:
$a ≤ b \implies a ⊕ b = 0$
⟨proof⟩

lemma tminus-nle[simp]:
$¬b ≤ a \implies a ⊕ b = 0$
⟨proof⟩

lemma tminus-add-mono:
$(a+b) ⊕ (c+d) ≤ (a⊕c) + (b⊕d)$
⟨proof⟩

lemma tminus-sum-mono:
assumes $fS$: finite $S$
shows $\text{sum } f S ⊕ \text{sum } g S ≤ \text{sum } (λx. f x ⊕ g x) S$
(is $?X S$)
⟨proof⟩

lemma tminus-nneg[simp,intro]:
$0 ≤ a ⊕ b$
⟨proof⟩

lemma tminus-right-antimono:
assumes $clb$: $c ≤ b$
shows $a ⊕ b ≤ a ⊕ c$
⟨proof⟩

lemma min-tminus-distrib:
$\text{min } a b ⊕ c = \text{min } (a ⊕ c) (b ⊕ c)$
⟨proof⟩

end
Bibliography


