pGCL for Isabelle

David Cock

August 16, 2018
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 Maximalitl 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
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:

datatype coin = Heads | Tails

record coins =
    a :: coin
    b :: coin

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

lemma
    Apply \((\lambda s. s(\{ a := Heads \}))\) = \((a := (\lambda s. Heads))\)
    ⟨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 Apply \((\lambda s. s([a := b \ s]))\), which updates \(a\) with the current value of \(b\). If we wish to formally
establish that the previous statement is correct i.e. that in the final state, 

\( a \) really will have whatever value \( b \) had in the initial state, we must first 

introduce the assertion language.

### 2.1.2 Assertion and Annotation

Assertions in pGCL are real-valued functions of the state, which are often 

interpreted as a probability distribution over possible outcomes. These func-

tions are termed *expectations*, for reasons which shortly be clear. Initially, 

however, we need only consider *standard* expectations: those derived from 

a binary predicate. A predicate \( P :: \text{bool} \) is embedded as \( « P » :: \text{real} \), such that 

\[ \begin{align*} 
P \mathbin{\rightarrow} « P » \mathbin{\rightarrow} 1 & \quad \land \quad \neg P \mathbin{\rightarrow} « P » \mathbin{\rightarrow} 0. 
\end{align*} \]

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 \( \forall s. P s \mathbin{\rightarrow} Q s \) is substituted with the equivalent 

expectation entailment \( « P » \models « Q » \), \[ \begin{align*} 
?P \models ?Q \iff ?P \models ?Q \mathbin{\rightarrow} ?Q \mathbin{\rightarrow} ?Q. 
\end{align*} \]

Thus a valid specification of \( \text{Apply} (\lambda s. s(a := b s)) \) is:

\[ \begin{align*} 
\forall x. « \lambda s. b s = x » \models \text{wp} (a := b) « \lambda s. a s = x » 
\end{align*} \]

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:

\[ \begin{align*} 
« \lambda s.:\text{real}. s \not= 0 » \models \text{wp} (\text{Apply} ((\ast) 2) ;; \text{Apply} (\lambda s. s / s)) « \lambda s. s = 1 » \end{align*} \]

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 \( y \) with probability 

\( (1::'a) - p \). The probability may depend on the state, and is therefore of
2.1. LANGUAGE PRIMITIVES

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

**definition**

\`flip-a :: real ⇒ coins prog

**where**

\`flip-a p = a ::= (\l. Heads) (\ls. p) ⊕ a ::= (\l. Tails)

**lemma**

(\ls. 1/2) = wp (flip-a (1/2)) «\ls. 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**

\`\ls. 1/3 ⊢ wp (flip-a (2/3) | flip-a (1/3)) «\ls. 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 ::= (\l. Heads) (\ls. p) ⊕ b ::= (\l. Tails)

**lemma**

(\ls. 1/4) = wp (flip-a (1/2) ; flip-b (1/2)) «\ls. 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 \( \text{Puniform-in} : \)
\[ x \in S \implies \text{Puniform } S x = 1 / \text{ card } S \]
\( \langle \text{proof} \rangle \)

lemma \( \text{Puniform-out} : \)
\[ x \notin S \implies \text{Puniform } S x = 0 \]
\( \langle \text{proof} \rangle \)

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

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

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

The expectations of independent variables add:

lemma \( (\lambda s. 7) = \text{ wp } ((\text{ bind } v \text{ at } (\lambda s. \text{ Puniform } \{1..6\} v) \text{ in red } := (\lambda s. v)) ;; \text{ (bind } v \text{ at } (\lambda s. \text{ Puniform } \{1..6\} v) \text{ in blue } := (\lambda s. v))) \text{ (\lambda s. red } s + \text{ blue } s) \)
\( \langle \text{proof} \rangle \)

debug

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

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

**definition** \textit{inv-count} :: \textit{int} \Rightarrow \textit{bool}

**where**

\[ \textit{inv-count} = (\lambda x. \ 0 \leq x) \]

Read \textit{wp-inv} \textit{G} \textit{body} \textit{I} as: \textit{I} is an invariant of the loop \(\mu x. \textit{body} ;; x < G \oplus \textit{Skip}\), or \(\langle G \rangle \& \& \textit{I} \triangleright \textit{wp body I}\).

**lemma** \textit{wp-inv-count}:

\[ \textit{wp-inv} (\lambda x. \ 0 < x) \ (\text{Apply} (\lambda s. s - 1)) \langle \textit{inv-count} \rangle \]

(proof)

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

**lemma** \textit{term-countdown}:

\[ \langle \textit{inv-count} \rangle \triangleright \textit{wp countdown} (\lambda s. 1) \]

(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** \textit{coin} = \textit{bool}

**definition** \textit{Heads} = \textit{True}

**definition** \textit{Tails} = \textit{False}

**definition** \textit{flip} :: \textit{coin prog}

**where**

\[ \textit{flip} = \text{Apply} (\lambda -. \textit{Heads}) (\lambda s. 1/2) \oplus \text{Apply} (\lambda -. \textit{Tails}) \]

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

**definition** \textit{wait-for-heads} :: \textit{coin prog}

**where**

\[ \textit{wait-for-heads} = \text{do} ((\neq) \textit{Heads}) \rightarrow \textit{flip} \textit{od} \]

Nonetheless, we can show termination.

**lemma** \textit{wait-for-heads-term}:

\[ \lambda s. 1 \triangleright \textit{wp wait-for-heads} (\lambda s. 1) \]

(proof)

end
2.3 The Monty Hall Problem

theory Monty imports ../pGCL begin

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

The setting is a game show: There is a prize hidden behind one of three doors, and the contestant is invited to choose one. Once the guess is made, the host then 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 ($\lambda s. 1/3$)$\oplus$ 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\} - \{prize\ s,\ 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\} - \{clue\ s,\ guess\ s\}$. guess-behind $d$

The complete game, either with or without switching guesses.

definition monty :: bool $\Rightarrow$ game prog
where
  monty switch $\equiv$ hide-prize $;$
  make-guess $;$
  reveal $;$
  (if switch then switch-guess else Skip)

2.3.3 A Brute Force Solution

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

lemma eval-win[simp]:
  $p = g \implies \langle \text{player-wins} \rangle (s| prize := p, guess := g, clue := c |) = 1$
CHAPTER 2. INTRODUCTION TO PGCL

lemma `eval-loss`[simp]:
\[ p \neq g \implies \text{«player-wins»} (s\| \text{prize} := p, \text{guess} := g, \text{clue} := c \|) = 0 \]

⟨proof⟩

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

lemma `wp-monty-noswitch`:
\[ (\lambda s. \frac{1}{3}) = \text{wp} (\text{monty False}) \text{«player-wins»} \]
⟨proof⟩

lemma `swap-upd`:
\[ s\| \text{prize} := p, \text{clue} := c, \text{guess} := g \| = \]
\[ s\| \text{prize} := p, \text{guess} := g, \text{clue} := c \| \]
⟨proof⟩

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

lemma `wp-monty-switch-brute-force`:
\[ (\lambda s. \frac{2}{3}) = \text{wp} (\text{monty True}) \text{«player-wins»} \]
⟨proof⟩

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`:
\[ \text{well-def hide-prize} \]
⟨proof⟩

lemma `wd-make-guess`:
\[ \text{well-def make-guess} \]
⟨proof⟩

lemma `wd-reveal`:
\[ \text{well-def reveal} \]
⟨proof⟩
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** wp-switch-guess:

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

\[ \vdash \vdash \text{wp switch-guess } \text{«player-wins»} \]

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

**Lemma** wp-monty-switch-modular:

\[ (\lambda s. 2/3) \vdash \vdash \text{wp (monty True) «player-wins»} \]

Using the VCG

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


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

**Lemma** wp-monty-switch-vcg:

\[ (\lambda s. 2/3) \vdash \vdash \text{wp (monty True) «player-wins»} \]

end
Chapter 3

Semantic Structures

3.1 Expectations

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

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

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

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

![Figure 3.1: A probabilistic automaton](image)

13
c is 3.0. The expected value from state \(a\) is the weighted sum of these, or \(0.7 \times 2.0 + 0.3 \times 3.0 = 2.3\).

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

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

### 3.1.1 Bounded Functions

**definition** `bounded-by :: real \to (\alpha \Rightarrow real) \Rightarrow bool`

**where** `bounded-by b P \equiv \forall x. P \ x \leq b`

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

**lemma** `bounded-byI[intro]`:

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

**proof**

**lemma** `bounded-byI2[intro]`:

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

**proof**

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

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

**proof**

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

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

**proof**

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

**definition** `bounded :: (\alpha \Rightarrow real) \Rightarrow bool`

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

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

**definition** `bound-of :: (\alpha \Rightarrow real) \Rightarrow real`

**where** `bound-of \ P \equiv \text{Sup} \ (P \ \cdot \ \text{UNIV})`

**lemma** `bounded-bdd-above[intro]`:

**assumes** `bP; \ \text{bounded} \ P`

**shows** `bdd-above \ (\text{range} \ P)`

**proof**

The least upper bound has the usual properties:
3.1. EXPECTATIONS

lemma bound-of-leq1[intro]:
  assumes \( P \leq (c :: \text{real}) \)
  shows \( \text{bound-of } P \leq c \)
  ⟨proof⟩

lemma bound-of-leq-alt[intro]:
  \( \text{bounded } b \text{ Q } \Rightarrow \text{bounded-by } b \text{ Q } \)
  ⟨proof⟩

lemma bound-of-leq2[intro]:
  \( \text{bounded } (\lambda x. c) \)
  ⟨proof⟩

lemma bound-of-leq3[intro]:
  \( \text{bounded-by } b \text{ Q } \Rightarrow \text{bounded-by } b \text{ (\lambda x. c) } \)
  ⟨proof⟩

lemma bound-of-leq4[intro]:
  \( \text{bounded-by } b \text{ Q } \Rightarrow \text{bounded-by } b \text{ Q } \)
  ⟨proof⟩

lemma bound-of-leq5[intro]:
  \( \text{bound-of } (\lambda x. c) = (c :: \text{real}) \)
  ⟨proof⟩

lemma bound-of-leq6[intro]:
  \( \forall x. \text{P } x \leq (c :: \text{real}) \)
  shows \( \text{bound-of } P \leq c \)
  ⟨proof⟩

lemma bound-of-leq7[intro]:
  \( \text{assumes } \forall x. \text{P } x \leq (c :: \text{real}) \)
  shows \( \text{bound-of } P \leq c \)
  ⟨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**

**3.1.2 Non-Negative Functions.**

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

**Definition**
\[\text{nneg} :: \langle 'a \Rightarrow 'b : \{\text{zero,order}\} \rangle \Rightarrow \text{bool}\]

**Where**
\[\text{nneg } P \iff (\forall x. 0 \leq P x)\]

**Lemma** nnegI[intro]:
\[\forall x. 0 \leq P x \implies \text{nneg } P\]

**Proof**

**Lemma** nnegI2[intro]:
\[(\lambda s. 0) \leq P \implies \text{nneg } P\]

**Proof**

**Lemma** nnegD[dest]:
\[\text{nneg } P \implies 0 \leq P x\]

**Proof**

**Lemma** nnegD2[dest]:
\[\text{nneg } P \implies (\lambda s. 0) \leq P\]

**Proof**

**Lemma** nneg-bdd-below[intro]:
\[\text{nneg } P \implies \text{bdd-below } (\text{range } P)\]

**Proof**

**Lemma** nneg-const[iff]:
\[\text{nneg } (\lambda x. c) \iff 0 \leq c\]

**Proof**

**Lemma** nneg-o[intro,simp]:
\[\text{nneg } P \implies \text{nneg } (P \circ f)\]

**Proof**
3.1. EXPECTATIONS

**Lemma** nneg-bound-nneg[intro]:
\[
\text{[ bounded } P; \text{ nneg } P \implies 0 \leq \text{bound-of } P
\]
(proof)

**Lemma** nneg-bounded-by-nneg[dest]:
\[
\text{[ bounded-by } b \text{ } P; \text{ nneg } P \implies 0 \leq (b::\text{real})
\]
(proof)

**Lemma** bounded-by-nneg[dest]: fixes \( P::s \Rightarrow \text{real} \)
shows \[
\text{[ bounded-by } b \text{ } P; \text{ nneg } P \implies 0 \leq b
\]
(proof)

3.1.3 Sound Expectations

definition sound :: (\( s \Rightarrow \text{real} \)) \Rightarrow \text{bool}
where sound \( P \equiv \text{bounded } P \land \text{nneg } P 

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

**Lemma** soundI:
\[
\text{[ bounded } P; \text{ nneg } P \implies \text{sound } P
\]
(proof)

**Lemma** soundI2[intro]:
\[
\text{[ bounded-by } b \text{ } P; \text{ nneg } P \implies \text{sound } P
\]
(proof)

**Lemma** sound-bounded[dest]:
sound \( P \implies \text{bounded } P
\)
(proof)

**Lemma** sound-nneg[dest]:
sound \( P \implies \text{nneg } P
\)
(proof)

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

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

**Lemma** sound-sum[simp,intro]:
assumes \( sP: \text{sound } P \text{ and } sQ: \text{sound } Q 
\)
shows sound (\( \lambda s. \text{P } s + Q \text{ } s \))
(proof)


\textbf{lemma} \textit{mult-sound}:  
\begin{description}  
\item[assumes] \(sP\) \text{ sound } \(P\) \text{ and } \(sQ\) \text{ sound } \(Q\)  
\item[shows] \text{ sound } (\lambda s. P s \ast Q s)  
\end{description}  
\textit{(proof)}

\textbf{lemma} \textit{div-sound}:  
\begin{description}  
\item[assumes] \(sP\) \text{ sound } \(P\) \text{ and } \text{cpos } \(\theta < c\)  
\item[shows] \text{ sound } (\lambda s. P s / c)  
\end{description}  
\textit{(proof)}

\textbf{lemma} \textit{tminus-sound}:  
\begin{description}  
\item[assumes] \(sP\) \text{ sound } \(P\) \text{ and } \text{nnc } \(\theta \leq c\)  
\item[shows] \text{ sound } (\lambda s. P s \ominus c)  
\end{description}  
\textit{(proof)}

\textbf{lemma} \textit{const-sound}:  
\begin{description}  
\item[0 \leq c \Rightarrow ] \text{ sound } (\lambda s. c)  
\end{description}  
\textit{(proof)}

\textbf{lemma} \textit{sound-o\{intro,simp\}}:  
\begin{description}  
\item[sound P \Rightarrow ] \text{ sound } (P o f)  
\end{description}  
\textit{(proof)}

\textbf{lemma} \textit{sc-bounded-by\{intro,simp\}}:  
\begin{description}  
\item[sound P; 0 \leq c \] \Rightarrow \text{ bounded-by } (c \ast \text{bound-of } P) (\lambda x. c \ast P x)  
\end{description}  
\textit{(proof)}

\textbf{lemma} \textit{sc-bounded\{intro,simp\}}:  
\begin{description}  
\item[assumes] \(sP\) \text{ sound } \(P\) \text{ and } \text{pos } \(\theta \leq c\)  
\item[shows] \text{ bounded } (\lambda x. c \ast P x)  
\end{description}  
\textit{(proof)}

\textbf{lemma} \textit{sc-bound\{simp\}}:  
\begin{description}  
\item[assumes] \(sP\) \text{ sound } \(P\) \text{ and } \text{nnc } \(\theta \leq c\)  
\item[shows] c \ast \text{bound-of } P = \text{bound-of } (\lambda x. c \ast P x)  
\end{description}  
\textit{(proof)}

\textbf{lemma} \textit{sc-sound}:  
\begin{description}  
\item[sound P; 0 \leq c \] \Rightarrow \text{ sound } (\lambda s. c \ast P s)  
\end{description}  
\textit{(proof)}

\textbf{lemma} \textit{bounded-by-mult}:  
\begin{description}  
\item[assumes] \(sP\) \text{ sound } \(P\) \text{ and } \text{bP} \text{ bounded-by } a \text{ P} \text{ and } \text{sQ} \text{ sound } \(Q\) \text{ and } \text{bQ} \text{ bounded-by } b \text{ Q}  
\item[shows] \text{ bounded-by } (a \ast b) (\lambda s. P s \ast Q s)  
\end{description}  
\textit{(proof)}
3.1. EXPECTATIONS

lemma bounded-by-add:
  fixes $P :: 's \Rightarrow \mathbb{R}$ and $Q$
  assumes $bP$: bounded-by $a P$
                   and $bQ$: bounded-by $b Q$
  shows bounded-by $(a + b) (\lambda s. P s + Q s)$
⟨proof⟩

lemma sound-unit[intro!, simp]:
  sound $(\lambda s. 1)$
⟨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 $(\lambda s. P s * Q s)$
⟨proof⟩

lemma sum-sound:
  assumes $sP$: $\forall x \in S. \text{sound } (P x)$
  shows sound $(\lambda s. \sum x \in S. P x s)$
⟨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 $P \iff$ sound $P \land$ 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
embed-bool :: ('s ⇒ bool) ⇒ 's ⇒ real ("-" 1000)

where

«P» ≡ (λs. if P s then 1 else 0)

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

lemma embed-bool-nneg:
  nneg «P»  ⟨proof⟩

lemma embed-bool-bounded-by-1:
  bounded-by 1 «P»  ⟨proof⟩

lemma embed-bool-bounded:
  bounded «P»  ⟨proof⟩

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

lemma embed-bool-idem:
  «P» s * «P» s = «P» s  ⟨proof⟩

lemma eval-embed-true:
  P s ⇒ «P» s = 1  ⟨proof⟩

lemma eval-embed-false:
  ¬P s ⇒ «P» s = 0  ⟨proof⟩

lemma embed-ge-0:
  0 ≤ «G» s  ⟨proof⟩

lemma embed-le-1:
  «G» s ≤ 1  ⟨proof⟩

lemma embed-le-1-alt:
  0 ≤ 1 - «G» s  ⟨proof⟩

lemma expect-1-I:
  P x ⇒ 1 ≤ «P» x  ⟨proof⟩
3.1. EXPECTATIONS

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

\texttt{sound} «P»

\langle\text{proof}\rangle

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

«P» o f = «P o f»

\langle\text{proof}\rangle

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

\langle\text{proof}\rangle

**lemma** embed-split:

f s = «P» s * f s + «N P» s * f s

\langle\text{proof}\rangle

**lemma** negate-embed:

«N P» s = 1 - «P» s

\langle\text{proof}\rangle

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

P s \implies «N P» s = 0

\langle\text{proof}\rangle

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

¬P s \implies «N P» s = 1

\langle\text{proof}\rangle

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

N Not = (λx. x)

\langle\text{proof}\rangle

**lemma** negate-negate[simp]:

N (N P) = P

\langle\text{proof}\rangle

**lemma** embed-bool-cancel:

«G» s * «N G» s = 0

\langle\text{proof}\rangle
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]`:

\[ \forall s. P s ≤ Q s \implies P ⊢ Q \]

**(proof)**

**lemma** `entailsD[dest]`:

\[ P ⊢ Q \implies P s ≤ Q s \]

**(proof)**

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

\[ P = Q \implies P ⊢ Q \]

**(proof)**

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

\[ [ P ⊢ Q; Q ⊢ R ] \implies P ⊢ R \]

**(proof)**

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

**lemma** `implies-entails`:

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

**(proof)**

**lemma** `entails-implies`:

\[ \forall s. \langle P \rangle ⊢ \langle Q \rangle; P s \implies Q s \]

**(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** \( a & \& b ≡ \lambda s. (a s .& b s) \)

Expectation conjunction likewise generalises (boolean) predicate conjunction. We show that the expected properties are preserved, and instantiate both the classical reasoner, and the simplifier (in the case of associativity and commutativity).
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-b conj:
\[ «a» s \land «b» s = «\lambda s. a s \land b s» s \]
⟨proof⟩

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

**lemma** pconj-assoc:
\[ \begin{array}{l}
\{ \theta \leq a; a \leq 1; \theta \leq b; b \leq 1; \theta \leq c; c \leq 1 \} \implies \\
a \land (b \land c) = (a \land b) \land c
\end{array} \]
⟨proof⟩

**lemma** pconj-mono:
\[ \begin{array}{l}
\{ a \leq b; c \leq d \} \implies a \land c \leq b \land d
\end{array} \]
⟨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 pconj-idem[simp]:
«P» s .& «P» s = «P» s
⟨proof⟩

3.1.8 Rules Involving Conjunction.

lemma exp-conj-mono-left:
\( P \vdash Q \implies P & R \vdash Q & R \)
⟨proof⟩

lemma exp-conj-mono-right:
\( Q \vdash R \implies P & Q \vdash P & R \)
⟨proof⟩

lemma exp-conj-comm[ac-simps]:
a & b = b & a
⟨proof⟩

lemma exp-conj-bounded-by[intro,simp]:
assumes \( bP \): bounded-by 1 P
and \( bQ \): bounded-by 1 Q
shows bounded-by 1 (P & Q)
⟨proof⟩

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

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

lemma exp-conj-top-left[simp]:
sound P \implies «λ.- True» & P = P
⟨proof⟩

lemma exp-conj-top-right[simp]:
sound P \implies P & «λ.- True» = P
⟨proof⟩

lemma exp-conj-idem[simp]:
«P» & «P» = «P»
⟨proof⟩

lemma exp-conj-nneg[intro,simp]:
(λs. 0) ≤ P & Q
⟨proof⟩
3.1. EXPECTATIONS

**Lemma exp Conj-Sound**: [intro, simp]
- **Assumes**: s-P: sound P
  - and s-Q: sound Q
- **Shows**: sound (P &amp; Q)
  ⟨proof⟩

**Lemma exp Conj-RZero**: [simp]
- **Bounded-by 1**: P ➔ P &amp; (λs. 0) = (λs. 0)
  ⟨proof⟩

**Lemma exp Conj-1-Right**: [simp]
- **Assumes**: nn: nneg A
- **Shows**: A &amp; (λs. 1) = A
  ⟨proof⟩

**Lemma exp Conj-Std-Split**: «λs. P s &amp; Q s» = «P» &amp; «Q»
  ⟨proof⟩

### 3.1.9 Rules Involving Entailment and Conjunction Together

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

**Lemma entails-Frame**: [intro, simp]
- **Assumes**: ePR: P ⊢ ⊢ R
  - and eQS: Q ⊢ ⊢ S
- **Shows**: P &amp; Q ⊢ ⊢ R &amp; S
  ⟨proof⟩

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

**Lemma pentails-Cases**: [intro, simp]
- **Assumes**: PQe: ∃x. P x ⊢ Q x
  - and exhaust: ∃s. ∃x. P (x s) s = 1
  - and framed: ∃x. P x &amp; R ⊢ Q x &amp; S
  - and sR: sound R and sS: sound S
  - and bQ: ∃x. bounded-by 1 (Q x)
- **Shows**: R ⊢ S
  ⟨proof⟩

**Lemma unitary-Bot**: [iff]
- unitary (λs. 0::real)
  ⟨proof⟩

**Lemma unitary-Top**: [iff]
- unitary (λs. 1::real)
  ⟨proof⟩
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 s * 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 (\( e \)) is no higher than 0.5. If it instead takes the edge to state \( a \), we again see a self loop, and thus in theory an infinite trace. In this case, however, every time the automaton reaches state \( a \), with probability 0.5 + 0.3 = 0.8, it transitions to a terminating state. An infinite trace of transitions \( a \to a \to \ldots \) thus has probability 0, and the automaton

![Figure 3.2: A nondeterministic-probabilistic automaton.](image-url)
terminates with probability 1. We formalise such probabilistic termination arguments in Section 4.11.

Having reached \( a \), the automaton will proceed to \( b \) with probability \( 0.5 \times (1/(0.5 + 0.3)) = 0.625 \), and to \( c \) with probability 0.375. As \( a \) is in turn reached half the time, the final probability of ending in \( b \) is 0.3125, and in \( c \), 0.1875, which sum to only 0.5. The remaining probability is that the automaton diverges via \( d \). We view nondeterminism and divergence demonically: we take the least probability of reaching a given final state, and use it to calculate the expectation. Thus for this automaton, \( P_{\text{prior}}(c) = 0.3125 \times P_{\text{post}}(b) + 0.1875 \times P_{\text{post}}(c) \). 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.
### 3.2.1 Comparing Transformers

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

**definition**

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

**where**

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

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

**definition**

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

**where**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

\[
\text{le-utrans } x x
\]
CHAPTER 3. SEMANTIC STRUCTURES

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

\[
\text{le-trans } t \ u \implies \text{le-utrans } t \ u
\]

**Definition**

l-trans :: \( \text{'s trans } \rightarrow \text{'s trans } \rightarrow \text{bool} \)

**Where**

\[
l-trans \ t \ u \iff \text{le-trans } t \ u \land \neg \text{le-trans } u \ t
\]

Transformer equivalence is induced by comparison:

**Definition**

equiv-trans :: \( \text{'s trans } \rightarrow \text{'s trans } \rightarrow \text{bool} \)

**Where**

equiv-trans \ t \ u \iff \text{le-trans } t \ u \land \text{le-trans } u \ t

**Definition**

equiv-utrans :: \( \text{'s trans } \rightarrow \text{'s trans } \rightarrow \text{bool} \)

**Where**

equiv-utrans \ t \ u \iff \text{le-utrans } t \ u \land \text{le-utrans } u \ t

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

\[
\left[ \left[ \forall P. \text{sound } P \implies t \ P = u \ P \right] \right] \implies \text{equiv-trans } t \ u
\]

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

\[
\left[ \left[ \forall P. \text{sound } P \implies t \ P = u \ P \right] \right] \implies \text{equiv-utrans } t \ u
\]

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

\[
\left[ \text{equiv-trans } t \ u ; \text{sound } P \right] \implies t \ P = u \ P
\]

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

\[
\left[ \text{equiv-utrans } t \ u ; \text{unitary } P \right] \implies t \ P = u \ P
\]

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

equiv-trans \ t \ t

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

**Lemma** le-trans-antisym:

\[
\left[ \text{le-trans } x \ y ; \text{le-trans } y \ x \right] \implies \text{equiv-trans } x \ y
\]

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

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

\textbf{lemma} \text{equiv-trans-comm[ac-simps]}:
\[
\text{equiv-trans } t \ u \iff \text{equiv-trans } u \ t
\]
(\text{proof})

\textbf{lemma} \text{equiv-utrans-comm[ac-simps]}:
\[
\text{equiv-utrans } t \ u \iff \text{equiv-utrans } u \ t
\]
(\text{proof})

\textbf{lemma} \text{equiv-imp-le[intro]}:
\[
\text{equiv-trans } t \ u \implies \text{le-trans } t \ u
\]
(\text{proof})

\textbf{lemma} \text{equivu-imp-le[intro]}:
\[
\text{equiv-utrans } t \ u \implies \text{le-utrans } t \ u
\]
(\text{proof})

\textbf{lemma} \text{equiv-imp-le-alt}:
\[
\text{equiv-trans } t \ u \implies \text{le-trans } u \ t
\]
(\text{proof})

\textbf{lemma} \text{equiv-uimp-le-alt}:
\[
\text{equiv-utrans } t \ u \implies \text{le-utrans } u \ t
\]
(\text{proof})

\textbf{lemma} \text{le-trans-equiv-rsp[simp]}:
\[
\text{equiv-trans } t \ u \implies \text{le-trans } u \ v \iff \text{le-trans } v \ u
\]
(\text{proof})

\textbf{lemma} \text{le-utrans-equiv-rsp[simp]}:
\[
\text{equiv-utrans } t \ u \implies \text{le-utrans } u \ v \iff \text{le-utrans } v \ u
\]
(\text{proof})

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

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

\textbf{lemma} \text{le-trans-equiv-rsp-right[simp]}:
\[
\text{equiv-trans } t \ u \implies \text{le-trans } v \ t \iff \text{le-trans } v \ u
\]
(\text{proof})

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


3.2.2 Healthy Transformers

Feasibility

**definition** feasible :: (('a ⇒ real) ⇒ ('a ⇒ real)) ⇒ bool

**where**

feasible t ⇐⇒ (∀ b P. bounded-by b P ∧ nneg P → bound-by b (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]:

∀ b P. [ bounded-by b P; nneg P ] → bound-by b (t P);

∀ b P. [ bound-by b P; nneg P ] → nneg (t P) → feasible t

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

[ feasible t; bound-by b P; nneg P ] → bound-by b (t P)

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

3.2. EXPECTATION TRANSFORMERS

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

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

\text{lemma feasible-pr-0[simp]:}
- \text{fixes } t::(\forall 's \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real}
- \text{assumes } ft: \text{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]:}
\[
[\text{feasible } t; \text{ sound } P; \text{ bounded-by } b \ P] \implies \text{bounded-by } b (t \ P)
\]

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

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

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

\text{Monotonicity}

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

\text{where}
- \text{mono-trans } t \equiv \forall P Q. (\text{sound } P \land \text{sound } Q \land P \leq Q) \implies t P \leq t Q

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

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

**lemma** mono-transI[intro]:

$[\bigwedge P Q. \blacksquare \text{sound } P; \text{sound } Q; P \leq Q \implies t P \leq t Q] \implies \text{mono-trans } t$

**lemma** mono-transD[dest]:

$[\text{mono-trans } t; \text{sound } P; P \leq Q \implies t P \leq t Q]$

**Scaling**

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

**definition** scaling :: \((\forall t. s \Rightarrow \text{real}) \Rightarrow (\forall t. s \Rightarrow \text{real})\) ⇒ bool

**where**

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

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

**lemma** scalingI[intro]:

$[\bigwedge P c x. \blacksquare \text{sound } P; 0 \leq c \implies c \ast t P x = t (\lambda x. c \ast P x) x] \implies \text{scaling } t$

**lemma** scalingD[dest]:

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

**lemma** right-scalingD:

**assumes** st: scaling $t$

**and** sP: sound $P$

**and** nnc: $0 \leq c$

**shows** $t P s \ast c = t (\lambda s. P s \ast c) s$

**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** \( \text{healthyI}[\text{intro}] \):

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

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

**lemma** \( \text{healthy-monoD}[\text{dest}] \):

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

**lemmas** \( \text{healthy-monoD2} = \text{mono-transD}[\text{OF healthy-monoD}] \)

**lemma** \( \text{healthy-feasibleD}[\text{dest}] \):

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

**lemma** \( \text{healthy-scalingD}[\text{dest}] \):

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

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

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

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

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

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

\[ \[[\text{healthy} t; \text{sound} P] \implies \text{bounded} (t P) \]

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

\[ \[[\text{healthy} t; \text{sound} P] \implies \text{nneg} (t P) \]

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

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

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

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

Lemma 3.1: healthy-unitary[intro]:
\[
\left[\text{healthy } t; \text{unitary } P\right] \implies \text{unitary } (t \ P)
\]
(Proof)

Lemma 3.2: healthy-id[simp,intro]:
\text{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]:
\text{healthy } t \implies \text{le-trans } (\lambda P \ s. 0) \ t
(Proof)

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

Lemma healthy-pr-bot[simp]:
\text{healthy } t \implies t (\lambda s. 0) = (\lambda s. 0)
(Proof)

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

Lemma healthy-equiv1:
\text{fixes } t:('s \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real and } u
\text{assumes equiv: equiv-trans } t \ u
\text{and healthy: healthy } t
\text{shows healthy } u
(Proof)

Lemma healthy-equiv:
\text{equiv-trans } t \ u \implies \text{healthy } t \iff \text{healthy } u
(Proof)

Lemma healthy-scale:
\text{fixes } t:('s \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real}
\text{assumes ht: healthy } t \text{ and nc: } 0 \leq c \text{ and bc: } c \leq 1
\text{shows healthy } (\lambda P \ s. c * t \ P \ s)
(Proof)

Lemma healthy-top[iff]:
\text{healthy } (\lambda P \ s. \text{bound-of } P)
(Proof)

Lemma healthy-bot[iff]:
\text{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}

\textit{nearly-healthy} :: \( ('s \Rightarrow \text{real}) \Rightarrow ('s \Rightarrow \text{real}) \Rightarrow \text{bool} \)

\textbf{where}

\( \text{nearly-healthy } t \leftarrow (\forall P. \text{unitary } P \longrightarrow \text{unitary } (t P)) \land \\
(\forall P Q. \text{unitary } P \longrightarrow \text{unitary } Q \longrightarrow P \vdash Q \longrightarrow t P \vdash t Q) \)

\textbf{lemma} \text{nearly-healthyI}[\text{intro}]:

\[ \begin{align*}
\land P. \text{unitary } P \Longrightarrow \text{unitary } (t P); \\
\land P Q. \land \text{unitary } P; \text{unitary } Q; P \vdash Q \Longrightarrow t P \vdash t Q \end{align*} \Longrightarrow \text{nearly-healthy } t
\]

\textbf{lemma} \text{nearly-healthy-monoD}[\text{dest}]:

\[ \begin{align*}
\land \text{nearly-healthy } t; P \vdash Q; \text{unitary } P; \text{unitary } Q \Longrightarrow t P \vdash t Q
\end{align*} \]

\textbf{lemma} \text{nearly-healthy-unitaryD}[\text{dest}]:

\[ \begin{align*}
\land \text{nearly-healthy } t; \text{unitary } P \Longrightarrow \text{unitary } (t P)
\end{align*} \]

\textbf{lemma} \text{healthy-nearly-healthy}[\text{dest}]:

\text{assumes} \text{ht}: \text{healthy } t

\text{shows} \text{nearly-healthy } t

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

\subsection*{3.2.3 Sublinearity}

As already mentioned, the core healthiness property (aside from feasibility and continuity) for transformers is \textit{sublinearity}: The transformation of a quasi-linear combination of sound expectations is greater than the same combination applied to the transformation of the expectations themselves. The term \( x \oplus y \) represents truncated subtraction i.e. \( \max (x - y) (0::'a) \) (see Section \ref{subsection:truncated_subtraction}).

\textbf{definition} \text{sublinear} ::

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

\textbf{where}

\( \text{sublinear } t \leftarrow (\forall a b c P Q s. (\text{sound } P \land \text{sound } Q \land 0 \leq a \wedge 0 \leq b \wedge 0 \leq c) \longrightarrow \\
\quad a * t P s + b * t Q s \ominus c \\
\quad \leq t (\lambda s'. a * P s' + b * Q s' \ominus c) s) \)

\textbf{lemma} \text{sublinearI}[\text{intro}]:
Figure 3.4: A graphical depiction of sub-additivity as convexity.

\[ Q = tP \cap uP \]

\[ Q(x) + Q(y) \]

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

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:

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

Can be rewritten, using scaling, to:

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

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

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

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

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

(proof)

**lemma sub-addI2:**

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

(proof)

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

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

(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 (λx. ∑y∈S. t (P y) x) ≤ t (λx. ∑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:

\[
\forall P. \text{sound } P \rightarrow \lambda s. t \; \mathcal{S} \; I \vdash t \; (\lambda s. P \; \mathcal{S} \; I) \] \rightarrow \text{sub-distrib } t
\]

(proof)

**Lemma** sub-distribD[dest]:

\[
\left[ \text{sub-distrib } t; \text{sound } P \right] \rightarrow t \; \mathcal{S} \; I \leq t \; (\lambda s'. P \; \mathcal{S} \; I) \; s
\]

(proof)

**Lemma** equiv-sub-distrib:

fixes \( t::(\mathcal{S} \Rightarrow \text{real}) \Rightarrow \mathcal{S} \Rightarrow \text{real} \)

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

and sd: sub-distrib \( t \)

shows sub-distrib \( u \)

(proof)

Sublinearity implies sub-distributivity:

**Lemma** sublinear-sub-distrib:

fixes \( t::(\mathcal{S} \Rightarrow \text{real}) \Rightarrow \mathcal{S} \Rightarrow \text{real} \)

assumes slt: 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 \( t::(\mathcal{S} \Rightarrow \text{real}) \Rightarrow \mathcal{S} \Rightarrow \text{real} \)

assumes sdt: sub-distrib \( t \) and sat: sub-add \( t \) and ht: healthy \( t \)

shows sublinear \( t \)

(proof)

### Sub-conjunctivity

**Definition**

\( \text{sub-conj } :: ((\mathcal{S} \Rightarrow \text{real}) \Rightarrow \mathcal{S} \Rightarrow \text{real}) \Rightarrow \text{bool} \)

where

\( \text{sub-conj } t \equiv \forall P \; Q. (\text{sound } P \land \text{sound } Q) \rightarrow t \; P \land t \; Q \vdash t \; (P \land Q) \)

**Lemma** sub-conjI[intro]:

\[
\left[ \forall P \; Q. [ \text{sound } P; \text{sound } Q ] \rightarrow t \; P \land t \; Q \vdash t \; (P \land Q) \right] \rightarrow \text{sub-conj } t
\]

(proof)

**Lemma** sub-conjD[dest]:

\[
\left[ \text{sub-conj } t; \text{sound } P; \text{sound } Q \right] \rightarrow t \; P \land t \; Q \vdash t \; (P \land Q)
\]

(proof)

**Lemma** sub-conj-wp-twice:
fixes \( f : s \Rightarrow ( ( s \Rightarrow \text{real} ) \Rightarrow \text{real} ) \)
assumes all: \( \forall s . \text{sub-conj} ( f s ) \)
shows \( \text{sub-conj} ( \lambda P s . f s P s ) \)

Sublinearity implies sub-conjunctivity:

\begin{verbatim}
lemma sublinear-sub-conj:
fixes t:(\text{\textquotesingle}s \Rightarrow \text{real}) \Rightarrow \text{\textquotesingle}s \Rightarrow \text{real}
assumes slt: sublinear t
shows sub-conj t
\end{verbatim}

Sublinearity under equivalence

Sublinearity is preserved by equivalence.

\begin{verbatim}
lemma equiv-sublinear:
[ [ equiv-trans t u; sublinear t; healthy t ] ] \Rightarrow sublinear u
\end{verbatim}

3.2.4 Determinism

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

Additivity

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

\begin{verbatim}
definition additive :: (\text{\textquotesingle}a \Rightarrow \text{real} \Rightarrow \text{\textquotesingle}a \Rightarrow \text{real} ) \Rightarrow \text{bool}
where
additive t \equiv \forall P Q . (\text{sound} P \land \text{sound} Q ) \rightarrow
t (\lambda s . P s + Q s ) = (\lambda s . t P s + t Q s )
lemma additiveD:
[ additive t; sound P; sound Q ] \Rightarrow t (\lambda s . P s + Q s ) = (\lambda s . t P s + t Q s )
\end{verbatim}

Additivity is strictly stronger than sub-additivity.
3.2. EXPECTATION TRANSFORMERS

The additivity property extends to finite summation.

**Lemma** additive-sum:

*Fixes* $S::'s set

*Assumes* additive: additive $t$

*and* healthy: healthy $t$

*and* finite: finite $S$

*and* $sPz$: $\forall z. sound (P z)$

*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:

*Fixes* $t::('s::finite ⇒ real) ⇒ 's ⇒ real

*Assumes* additive: additive $t$

*and* ht: healthy $t$

*and* $sP$: sound $P$

*Shows* $t P x = (\sum y \in UNIV. P y \ast t \ » \lambda z. z = y » x)$

*Proof*

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

**Lemma** additive-guard-split:

*Fixes* $t::('s::finite ⇒ real) ⇒ 's ⇒ real

*Assumes* additive: additive $t$

*and* ht: healthy $t$

*and* $sP$: sound $P$

*Shows* $t P x = (\sum y \in \{s. G s\}. P y \ast t \ » \lambda z. z = y » x) + (\sum y \in \{s. \neg G s\}. P y \ast t \ » \lambda z. z = y » x)$

*Proof*

**Maximality**

**Definition**

$maximal :: (('a ⇒ real) ⇒ 'a ⇒ real) ⇒ bool$

*Where*

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

**Lemma** maximalI[intro]:

$[ \forall c. 0 \leq c \rightarrow t (\lambda -. c) = (\lambda -. c) ] \rightarrow maximal t$

*Proof*

**Lemma** maximalD[dest]:

$[ maximal t; 0 \leq c ] \Rightarrow t (\lambda -. c) = (\lambda -. c)$
A transformer that is both additive and maximal is deterministic:

\begin{definition}
\textit{determ} :: ((\textit{a} \Rightarrow \textit{real}) \Rightarrow \textit{a} \Rightarrow \textit{real}) \Rightarrow \textit{bool}
\end{definition}

\begin{proof}
\textit{determ} \texttt{t} \equiv \textit{additive} \texttt{t} \land \textit{maximal} \texttt{t}
\end{proof}

\begin{lemma} \textit{determI}[\textit{intro}]:
\begin{assumes}
\textit{additive} \texttt{t}; \textit{maximal} \texttt{t}
\end{assumes}
\Rightarrow \textit{determ} \texttt{t}
\end{lemma}

\begin{proof}
\end{proof}

\begin{lemma} \textit{determ-additiveD}[\textit{intro}]:
\textit{determ} \texttt{t} \Rightarrow \textit{additive} \texttt{t}
\end{lemma}

\begin{proof}
\end{proof}

\begin{lemma} \textit{determ-maximalD}[\textit{intro}]:
\textit{determ} \texttt{t} \Rightarrow \textit{maximal} \texttt{t}
\end{lemma}

\begin{proof}
\end{proof}

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

\begin{lemma} \textit{determ-negate}:
\begin{assumes}
\textit{determ}:
\texttt{determ} \texttt{t}
\end{assumes}
\begin{shows}
\texttt{t} \texttt{«P»} \texttt{s} + \texttt{t} \texttt{«N»} \texttt{P} \texttt{s} = 1
\end{shows}
\end{lemma}

\begin{proof}
\end{proof}

\section{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.

\begin{lemma} \textit{entails-combine}:
\begin{assumes}
wp1: \texttt{P} \Downarrow \texttt{t} \texttt{R}
\text{ and wp2: } \texttt{Q} \Downarrow \texttt{t} \texttt{S}
\text{ and sc: } \textit{sub-conj} \texttt{t}
\text{ and sR: } \textit{sound} \texttt{R}
\text{ and sS: } \textit{sound} \texttt{S}
\end{assumes}
\begin{shows}
\texttt{P} \&\& \texttt{Q} \Downarrow \texttt{t} (\texttt{R} \&\& \texttt{S})
\end{shows}
\end{lemma}

\begin{proof}
\end{proof}

These allow mismatched results to be composed

\begin{lemma} \textit{entails-strengthen-post}:
\begin{assumes}
\texttt{P} \Downarrow \texttt{t} \texttt{Q; healthy} \texttt{t}; \textit{sound} \texttt{R}; \texttt{Q} \Downarrow \texttt{R}; \textit{sound} \texttt{Q}
\end{assumes}
\Rightarrow \texttt{P} \Downarrow \texttt{t} \texttt{R}
\end{lemma}

\begin{proof}
\end{proof}
3.2. EXPECTATION TRANSFORMERS

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

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

lemma entails-scale:
assumes wp: $P \vdash t Q$ and h: healthy $t$
and sQ: sound $Q$ and pos: $0 \leq c$
shows $(\lambda s. c * P s) \vdash t (\lambda s. c * Q s)$
\(\langle proof \rangle\)

3.2.6 Transforming Standard Expectations

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

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

lemma use-premise:
assumes h: healthy $t$ and wP: $\land s. P s \implies 1 \leq t Q s$
shows $P s \vdash t Q s$
\(\langle proof \rangle\)

The other direction works too.

lemma fold-premise:
assumes ht: healthy $t$
and wp: $P s \vdash t Q s$
shows $\forall s. P s \implies 1 \leq t Q s$
\(\langle proof \rangle\)

Predicate conjunction behaves as expected:

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

Similar to $[healthy ?t; \land s. ?P s \implies 1 \leq ?t ?Q s] \implies ?P ?t ?Q$, but more general.

lemma entails-peonj-assumption:
assumes f: feasible $t$ and wP: $\land s. P s \implies Q s \leq t R s$
and wQ: unitary $Q$ and wR: unitary $R$
shows $P s \land Q \vdash t R$
\(\langle proof \rangle\)

end
3.3 Induction

theory Induction
  imports Expectations Transformers
begin

3.3.1 The Lattice of Expectations

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

Finding a lower bound is easy (it’s \( \lambda-. \ 0::'b \)), but as we do not insist on any global bound on expectations (and work directly in HOL’s real type, rather than extending it with a point at infinity), there is no top element.

We solve the problem by defining the least (greatest) fixed point, restricted to an internally-bounded set, allowing us to substitute this bound for the top element. This works as long as the set contains at least one fixed point, which appears as an extra assumption in all the theorems.

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

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

lemma Inf-exp-lower:
\[ [ P \in S; \forall P \in S. \text{nneg } P ] \Rightarrow \text{Inf-exp } S \leq P \]
(proof)

lemma Inf-exp-greatest:
\[ [ S \neq \emptyset; \forall P \in S. \text{bounded-by } b \ P ] \Rightarrow Q \leq \text{Inf-exp } S \]
(proof)

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

lemma Sup-exp-upper:
\[ [ P \in S; \forall P \in S. \text{bounded-by } b \ P ] \Rightarrow P \leq \text{Sup-exp } S \]
(proof)

lemma Sup-exp-least:
\[ [ \forall P \in S. \ P \leq Q; \text{nneg } Q ] \Rightarrow \text{Sup-exp } S \leq Q \]
3.3. INDUCTION

⟨proof⟩

lemma Sup-exp-sound:
  assumes sS: \( \forall P. P \in S \implies \text{sound } P \)
  and bS: \( \forall P. P \in S \implies \text{bounded-by } b \) P
  shows sound (\( \text{Sup-exp } S \))
⟨proof⟩

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

lemma lfp-exp-lowerbound:
  \( \lbrack \ l t \leq P; \text{sound } P \rbrack \implies \text{lfp-exp } t \leq P \)
⟨proof⟩

lemma lfp-exp-greatest:
  \( \lbrack \ l \( \forall P. \lbrack \ l t \leq P; \text{sound } P \rbrack \implies Q \leq P; \text{sound } Q; t R \vdash R; \text{sound } R \rbrack \implies Q \leq \text{lfp-exp } t \)
⟨proof⟩

lemma feasible-lfp-exp-sound:
  \( \text{feasible } t \implies \text{sound } (\text{lfp-exp } t) \)
⟨proof⟩

lemma lfp-exp-bound:
  \( \lbrack \ l \( \forall P. \text{unitary } P \implies \text{unitary } (t P) \) \implies \text{bounded-by } t \ (\text{lfp-exp } t) \)
⟨proof⟩

lemma lfp-exp-unitary:
  \( \lbrack \ l \( \forall P. \text{unitary } P \implies \text{unitary } (t P) \) \implies \text{unitary } (\text{lfp-exp } t) \)
⟨proof⟩

lemma lfp-exp-lemma2:
  fixes t::'s trans
  assumes st: \( \forall P. \text{sound } P \implies \text{sound } (t P) \)
  and mt: mono-trans t
  and fR: t R ⊢ R and sR: sound R
  shows t (\( \text{lfp-exp } t \)) \leq \text{lfp-exp } t
⟨proof⟩

lemma lfp-exp-lemma3:
  assumes st: \( \forall P. \text{sound } P \implies \text{sound } (t P) \)
  and mt: mono-trans t
  and fR: t R ⊢ R and sR: sound R
shows lfp-exp t ≤ t (lfp-exp t)  
⟨proof⟩

lemma lfp-exp-unfold:
assumes nt: ∃P. sound P ⇒ sound (t P)
  and mt: mono-trans t
  and ff: t R ⊢ R and sR: sound R
shows lfp-exp t = t (lfp-exp t)  
⟨proof⟩

definition gfp-exp :: ‘s trans ⇒ ‘s expect
where gfp-exp t = Sup-exp {P. unitary P ∧ P ≤ t P}

lemma gfp-exp-upperbound:
  [ [ P ≤ t P; unitary P ] ⇒ P ≤ gfp-exp t ]  
  ⟨proof⟩

lemma gfp-exp-least:
  [ [ λP. [ P ≤ t P; unitary P ] ⇒ P ≤ Q; unitary Q ] ⇒ gfp-exp t ≤ Q ]  
  ⟨proof⟩

lemma gfp-exp-bound:
  (λP. unitary P ⇒ unitary (t P)) ⇒ bounded-by 1 (gfp-exp t)  
  ⟨proof⟩

lemma gfp-exp-nneg[iff]:
  nneg (gfp-exp t)  
  ⟨proof⟩

lemma gfp-exp-unitary:
  (λP. unitary P ⇒ unitary (t P)) ⇒ unitary (gfp-exp t)  
  ⟨proof⟩

lemma gfp-exp-lemma2:
assumes fl: λP. unitary P ⇒ unitary (t P)
  and mt: λP Q. [ unitary P; unitary Q; P ⊢ Q ] ⇒ t P ⊢ t Q
shows gfp-exp t ≤ t (gfp-exp t)  
⟨proof⟩

lemma gfp-exp-lemma3:
assumes fl: λP. unitary P ⇒ unitary (t P)
  and mt: λP Q. [ unitary P; unitary Q; P ⊢ Q ] ⇒ t P ⊢ t Q
shows t (gfp-exp t) ≤ gfp-exp t  
⟨proof⟩

lemma gfp-exp-unfold:
(λP. unitary P ⇒ unitary (t P)) ⇒ (λP Q. [ unitary P; unitary Q; P ⊢ Q ] ⇒ t P ⊢ t Q) ⇒
  gfp-exp t = t (gfp-exp t)
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 $\lambda \cdot \cdot. \emptyset : 'c$, and the top element is the greatest allowed by feasibility: $\lambda P \cdot. \text{bound-of } P$.

**definition** Inf-trans :: 's trans set $\Rightarrow$ 's trans

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

**lemma** Inf-trans-lower:

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

**lemma** Inf-trans-greatest:

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

**definition** Sup-trans :: 's trans set $\Rightarrow$ 's trans

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

**lemma** Sup-trans-upper:

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

**lemma** Sup-trans-upper2:

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

\[
\text{negg } P; \text{bounded-by } b P \] \Rightarrow \forall P. \text{unitary } P = \Rightarrow \text{le-utrans } t (\text{Sup-trans } S)
\]

**lemma** Sup-trans-least:

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

**lemma** Sup-trans-least2:

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

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

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

\[
\Rightarrow \text{Sup-trans } S P \uparrow u P
\]

**lemma** feasible-Sup-trans:
CHAPTER 3. SEMANTIC STRUCTURES

\textbf{fixes} \( S \): ‘s trans set
\textbf{assumes} \( fS \): \( \forall t \in S. \ \text{feasible} \ t \)
\textbf{shows} \( \text{feasible} (\text{Sup}-\text{trans} \ S) \)
\( \langle \text{proof} \rangle \)

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

\textbf{lemma} \( \text{lfp-trans-lowerbound} \):
\( [ \ \text{le-trans} \ (T \ t) \ t; \ \land P. \ \text{sound} \ P \Implies \ \text{sound} \ (t \ P) ] \Implies \ \text{le-trans} \ (\text{lfp-trans} \ T) \ t \)
\( \langle \text{proof} \rangle \)

\textbf{lemma} \( \text{lfp-trans-greatest} \):
\( [ \ \land P. \ [ \ \text{le-trans} \ (T \ t) \ t; \ \land P. \ \text{sound} \ P \Implies \ \text{sound} \ (t \ P) ] \Implies \ \text{le-trans} \ u \ t; \ \land P. \ \text{sound} \ P \Implies \ \text{sound} \ (v \ P); \ \text{le-trans} \ (T \ v) \ v ] \Implies \ \text{le-trans} \ u \ ((\text{lfp-trans} \ T)) \)
\( \langle \text{proof} \rangle \)

\textbf{lemma} \( \text{lfp-trans-sound} \):
\textbf{fixes} \( P \ Q \):: ‘s expect
\textbf{assumes} \( sP \): \( \text{sound} \ P \)
\textbf{and} \( fV \): \( \text{le-trans} \ (T \ v) \ v \)
\textbf{and} \( sv \): \( \land P. \ \text{sound} \ P \Implies \ \text{sound} \ (v \ P) \)
\textbf{shows} \( \text{sound} \ (\text{lfp-trans} \ T \ P) \)
\( \langle \text{proof} \rangle \)

\textbf{lemma} \( \text{lfp-trans-unitary} \):
\textbf{fixes} \( P \ Q \):: ‘s expect
\textbf{assumes} \( aP \): \( \text{unitary} \ P \)
\textbf{and} \( fV \): \( \text{le-trans} \ (T \ v) \ v \)
\textbf{and} \( sv \): \( \land P. \ \text{sound} \ P \Implies \ \text{sound} \ (v \ P) \)
\textbf{and} \( ft \): \( \text{le-trans} \ (T (\lambda P \ s. \ \text{bound-of} \ P)) \ (\lambda P \ s. \ \text{bound-of} \ P) \)
\textbf{shows} \( \text{unitary} \ (\text{lfp-trans} \ T \ P) \)
\( \langle \text{proof} \rangle \)

\textbf{lemma} \( \text{lfp-trans-lemma2} \):
\textbf{fixes} \( v \):: ‘s trans
\textbf{assumes} \( \text{mono} \): \( \land t \ u. \ [ \ \text{le-trans} \ t \ u; \ \land P. \ \text{sound} \ P \Implies \ \text{sound} \ (t \ P); \ \land P. \ \text{sound} \ P \Implies \ \text{sound} \ (u \ P)] \Implies \ \text{le-trans} \ (T \ t) \ (T \ u) \)
\textbf{and} \( nT \): \( \land \ Q. \ [ \ \land Q. \ \text{sound} \ Q \Implies \ \text{sound} \ (t \ Q); \ \text{sound} \ P \ ] \Implies \ \text{sound} \ (T \ t \ P) \)
\textbf{and} \( fV \): \( \text{le-trans} \ (T \ v) \ v \)
\textbf{and} \( sv \): \( \land P. \ \text{sound} \ P \Implies \ \text{sound} \ (v \ P) \)
\textbf{shows} \( \text{le-trans} \ (T (\text{lfp-trans} \ T)) \ (\text{lfp-trans} \ T) \)
\( \langle \text{proof} \rangle \)

\textbf{lemma} \( \text{lfp-trans-lemma3} \):
\textbf{fixes} \( v \):: ‘s trans
3.3. INDUCTION

assumes mono: \( \forall t, u. \leq \text{trans} t u; \forall P. \text{sound} P \Rightarrow \text{sound} (t P) \);
and \( sT: \quad \forall t. \left[ \forall Q. \text{sound} Q \Rightarrow \text{sound} (t Q); \text{sound} P \right] \Rightarrow \text{sound} (t T P) \)
and \( fP: \quad \text{le-trans} (T v) v \)
and \( sv: \forall P. \text{sound} P \Rightarrow \text{sound} (v P) \)
shows le-trans \( (\text{fp-trans} T) (T (\text{fp-trans} T)) \)

(\text{proof})

lemma \( \text{fp-trans-unfold} \):
\[
\text{fixes} \quad P::{'}s \text{ expect}
\]
assumes mono: \( \forall t, u. \leq \text{trans} t u; \forall P. \text{sound} P \Rightarrow \text{sound} (t P) \);
and \( sT: \quad \forall t. \left[ \forall Q. \text{sound} Q \Rightarrow \text{sound} (t Q); \text{sound} P \right] \Rightarrow \text{sound} (t T P) \)
and \( fP: \quad \text{le-trans} (T v) v \)
and \( sv: \forall P. \text{sound} P \Rightarrow \text{sound} (v P) \)
shows equiv-trans \( (\text{fp-trans} T) (T (\text{fp-trans} T)) \)
(\text{proof})

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

lemma \( \text{gfp-trans-upperbound} \):
\[
\left[ \leq \text{trans} t (T t); \forall P. \text{unitary} P \Rightarrow \text{unitary} (t P) \right] \Rightarrow \leq \text{trans} t (\text{gfp-trans} T)
\]
(\text{proof})

lemma \( \text{gfp-trans-least} \):
\[
\left[ \forall t. \left[ \leq \text{trans} t (T t); \forall P. \text{unitary} P \Rightarrow \text{unitary} (t P) \right] \Rightarrow \leq \text{trans} t u; \right.
\left. \forall P. \text{unitary} P \Rightarrow \text{unitary} (u P) \right] \Rightarrow \leq \text{trans} (\text{gfp-trans} T) u
\]
(\text{proof})

lemma \( \text{gfp-trans-unitary} \):
\text{fixes} \quad P::{'}s \text{ expect}
assumes \( uP: \text{unitary} P \)
shows unitary \( (\text{gfp-trans} T P) \)
(\text{proof})

lemma \( \text{gfp-trans-lemma2} \):
\text{assumes} \quad \text{mono:} \quad \forall t, u. \leq \text{trans} t u; \forall P. \text{unitary} P \Rightarrow \text{unitary} (t P) ;
\quad \forall P. \text{unitary} P \Rightarrow \text{unitary} (u P) \Rightarrow \leq \text{trans} (T t) (T u) \)
and \( \text{htT:} \quad \forall t. \left[ \forall Q. \text{unitary} Q \Rightarrow \text{unitary} (t Q); \text{unitary} P \right] \Rightarrow \text{unitary} (T t P) \)
shows le-trans \( (\text{gfp-trans} T) (T (\text{gfp-trans} T)) \)
(\text{proof})
\textbf{lemma} gfp-trans-lemma3:
\textbf{assumes} \(\text{mono}: \forall t u. \left[ \text{le-utrans } t u; \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) \right]
\land \left[ \forall P. \text{unitary } P \Rightarrow \text{unitary } (u P) \right] \Rightarrow \text{le-utrans } (T t) (T u)
\land hT: \forall t P. \left[ \forall Q. \text{unitary } Q \Rightarrow \text{unitary } (t Q) ; \text{unitary } P \right] \Rightarrow \text{unitary } (T t P)
\text{shows} \ \text{le-utrans } (T (\text{gfp-trans } T)) (\text{gfp-trans } T)
\langle \text{proof} \rangle

\textbf{lemma} gfp-trans-unfold:
\textbf{assumes} \(\text{mono}: \forall t u. \left[ \text{le-utrans } t u; \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) \right]
\land \left[ \forall P. \text{unitary } P \Rightarrow \text{unitary } (u P) \right] \Rightarrow \text{le-utrans } (T t) (T u)
\land hT: \forall t P. \left[ \forall Q. \text{unitary } Q \Rightarrow \text{unitary } (t Q) ; \text{unitary } P \right] \Rightarrow \text{unitary } (T t P)
\text{shows} \ \text{equiv-utrans } (\text{gfp-trans } T) (T (\text{gfp-trans } T))
\langle \text{proof} \rangle

\subsection{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.

\textbf{lemma} gfp-pulldown:
\textbf{fixes} \(P::'s \text{ expect and } T::'s \text{ trans}
\textbf{assumes} \text{tailcall}: \forall u P. \text{unitary } P \Rightarrow T u P = t P (u P)
\land fT: \forall t P. \left[ \forall Q. \text{unitary } Q \Rightarrow \text{unitary } (t Q) ; \text{unitary } P \right] \Rightarrow \text{unitary } (T t P)
\land ft: \forall P Q. \text{unitary } P \Rightarrow \text{unitary } Q \Rightarrow \text{unitary } (t P Q)
\land mt: \forall P Q R. \left[ \forall P. \text{unitary } P ; \text{unitary } Q ; \text{unitary } R ; Q \dashv P \right] \Rightarrow t P Q P R
\land uP: \text{unitary } P
\land \text{monoT}: \forall t u. \left[ \text{le-trans } t u; \forall P. \text{unitary } P \Rightarrow \text{unitary } (t P) ; \forall P. \text{unitary } P \Rightarrow \text{unitary } (u P) \right] \Rightarrow \text{le-trans } (T t ) (T u)
\text{shows} \ \text{gfp-trans } T P = \text{gfp-exp } (t P) (\text{is } ?X P = ?Y P)
\langle \text{proof} \rangle

\textbf{lemma} lfp-pulldown:
\textbf{fixes} \(P::'s \text{ expect and } T::'s \text{ trans}
\textbf{assumes} \text{tailcall}: \forall u P. \text{sound } P \Rightarrow T u P = t P (u P)
\land st: \forall P Q. \text{sound } P \Rightarrow \text{sound } Q \Rightarrow \text{sound } (t P Q)
\land mt: \forall P. \text{sound } P \Rightarrow \text{mono-trans } (t P)
\land \text{monoT}: \forall t u. \left[ \text{le-trans } t u; \forall P. \text{sound } P \Rightarrow \text{sound } (t P) ; \forall P. \text{sound } P \Rightarrow \text{sound } (u P) \right] \Rightarrow \text{le-trans } (T t ) (T u)
\land nT: \forall t P. \left[ \forall Q. \text{sound } Q \Rightarrow \text{sound } (t Q) ; \text{sound } P \right] \Rightarrow \text{sound } (T t P)
\land fv: \text{le-trans } (T v) v
\land sv: \forall P. \text{sound } P \Rightarrow \text{sound } (v P)
\land sP: \text{sound } P
\langle \text{proof} \rangle
3.3. **INDUCTION**

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

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

**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** \( \text{Inf-utrans-lower:} \)

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

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

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

end
Chapter 4

The pGCL Language

4.1 A Shallow Embedding of pGCL in HOL

theory Embedding imports Misc Induction begin

4.1.1 Core Primitives and Syntax

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

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

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

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

Skip does nothing at all.

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

Apply lifts a state transformer into the space of programs.

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

Seq is sequential composition.

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

PC is probabilistic choice between programs.

definition PC :: 's prog ⇒ ('s ⇒ real) ⇒ 's prog ⇒ 's prog
(- ⊕ - [58,57,57] 57)
where
PC a P b ≡ λab Q s. P s * a ab Q s + (1 − P s) * b ab Q s
DC is demonic choice between programs.

**Definition**

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

AC is angelic choice between programs.

**Definition**

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

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

**Definition**

\[ Embed \colon \text{'s trans} \Rightarrow \text{'s prog} \]  
\[ Embed t = (\lambda ab. t) \]

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

**Definition**

\[ Mu \colon (\text{'s prog} \Rightarrow \text{'s prog}) \Rightarrow \text{'s prog} \]  
\[ Mu(T) \equiv \begin{cases} \lambda ab. \text{if ab then lfp-trans (\lambda t. T (Embed t) ab)} & \text{else gfp-trans (\lambda t. T (Embed t) ab)} \end{cases} \]

Repeat expresses finite repetition

**Primrec**

\[ repeat :: \text{nat} \Rightarrow \text{'a prog} \Rightarrow \text{'a prog} \]  
\[ \begin{array}{l} \text{where} \\
repeat 0 p = \text{Skip} \\
repeat (\text{Suc } n) p = p ;; \text{repeat } n \text{ p} \end{array} \]

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

**Definition**

\[ SetDC :: (\text{'a} \Rightarrow \text{'s prog}) \Rightarrow (\text{'s} \Rightarrow \text{'a set}) \Rightarrow \text{'s prog} \]  
\[ \text{where } SetDC f S \equiv \lambda ab P s. \text{Inf} ((\lambda a. f a ab P s) \cdot S s) \]

**Syntax**

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

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

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

**Definition**

\[ SetPC :: (\text{'a} \Rightarrow \text{'s prog}) \Rightarrow (\text{'s} \Rightarrow \text{'a} \Rightarrow \text{real}) \Rightarrow \text{'s prog} \]  
\[ \text{where } SetPC f p \equiv \lambda ab P s. \sum a \in \text{supp } (p s). p s a * f a ab P s \]

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

**Definition**

\[ Bind :: (\text{'s} \Rightarrow \text{'a}) \Rightarrow (\text{a} \Rightarrow \text{'s prog}) \Rightarrow \text{'s prog} \]
where

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

This gives us something like let syntax

```plaintext
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.

```plaintext
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))
```

```plaintext
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.

```plaintext
syntax -assign :: ident => 'a => 's prog (\:= - [1000,900])
⟨ML⟩
```

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

```plaintext
syntax -set-dc :: ident => (s => 'a set) => 's prog (\in - [66,66,66])
⟨ML⟩
```

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

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

definition wp :: 's prog => 's trans
where
wp pr \equiv pr True
CHAPTER 4. THE PGCL LANGUAGE

**definition**

\[ wlp :: 's prog \Rightarrow 's trans \]

**where**

\[ wlp \ pr \equiv \ pr \ False \]

If-Then-Else as a degenerate probabilistic choice.

**abbreviation**

\[ (\text{input}) \]

\[ \text{if-then-else} :: ['s \Rightarrow \text{bool}, 's \text{ prog}, 's \text{ prog}] \Rightarrow 's \text{ prog} \]

**where**

\[ \text{If } P \text{ Then } a \text{ Else } b \equiv a \ _{P \ b} \oplus b \]

Syntax for loops

**abbreviation**

\[ (\text{input}) \]

\[ \text{do-while} :: ['s \Rightarrow \text{bool}, 's \text{ prog}] \Rightarrow 's \text{ prog} \]

**where**

\[ \text{do-while } P \ a \equiv \mu x. \text{If } P \text{ Then } a \ ;; \ x \text{ Else Skip} \]

4.1.2 Unfolding rules for non-recursive primitives

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

\[ \text{wp} \ \text{Abort} \ P = (\lambda s. \ 0) \]

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

\[ \text{wlp} \ \text{Abort} \ P = (\lambda s. \ 1) \]

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

\[ \text{wp} \ \text{Skip} \ P = P \]

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

\[ \text{wlp} \ \text{Skip} \ P = P \]

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

\[ \text{wp} \ (\text{Apply } f) \ P = P \ o f \]

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

\[ \text{wlp} \ (\text{Apply } f) \ P = P \ o f \]

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

\[ \text{wp} \ (a \ ;; \ b) \ P = (\text{wp } a \ o \wp b) \ P \]

\(\langle \text{proof} \rangle\)
4.1. A SHALLOW EMBEDDING OF PGCL IN HOL

lemma eval-wlp-Seq:
\[ \text{wlp} (a ;; b) P = (\text{wlp} a \circ \text{wlp} b) P \]
\(\langle\text{proof}\rangle\)

lemma eval-wp-PC:
\[ \text{wp} (a \mathbin{Q\oplus} b) P = (\lambda s. \text{Q s} \ast \text{wp} a P s + (1 - \text{Q s}) \ast \text{wp} b P s) \]
\(\langle\text{proof}\rangle\)

lemma eval-wlp-PC:
\[ \text{wlp} (a \mathbin{Q\oplus} b) P = (\lambda s. \text{Q s} \ast \text{wlp} a P s + (1 - \text{Q s}) \ast \text{wlp} b P s) \]
\(\langle\text{proof}\rangle\)

lemma eval-wp-DC:
\[ \text{wp} (a \mathbin{\bigcap} b) P = (\lambda s. \min (\text{wp} a P s) (\text{wp} b P s)) \]
\(\langle\text{proof}\rangle\)

lemma eval-wlp-DC:
\[ \text{wlp} (a \mathbin{\bigcap} b) P = (\lambda s. \min (\text{wlp} a P s) (\text{wlp} b P s)) \]
\(\langle\text{proof}\rangle\)

lemma eval-wp-AC:
\[ \text{wp} (a \mathbin{\bigcup} b) P = (\lambda s. \max (\text{wp} a P s) (\text{wp} b P s)) \]
\(\langle\text{proof}\rangle\)

lemma eval-wlp-AC:
\[ \text{wlp} (a \mathbin{\bigcup} b) P = (\lambda s. \max (\text{wlp} a P s) (\text{wlp} b P s)) \]
\(\langle\text{proof}\rangle\)

lemma eval-wp-Embed:
\[ \text{wp} (\text{Embed} t) = t \]
\(\langle\text{proof}\rangle\)

lemma eval-wlp-Embed:
\[ \text{wlp} (\text{Embed} t) = t \]
\(\langle\text{proof}\rangle\)

lemma eval-wp-SetDC:
\[ \text{wp} (\text{SetDC} p S) R s = \text{Inf} ((\lambda a. \text{wp} (p a) R s) \circ S s) \]
\(\langle\text{proof}\rangle\)

lemma eval-wlp-SetDC:
\[ \text{wlp} (\text{SetDC} p S) R s = \text{Inf} ((\lambda a. \text{wlp} (p a) R s) \circ S s) \]
\(\langle\text{proof}\rangle\)

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

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

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

lemma eval-wp-Bind:
\[ wp(Bind\ g\ f) = (\lambda P\ s.\ wp(f(g\ s))\ P\ s) \]
(proof)

lemma eval-wlp-Bind:
\[ wlp(Bind\ g\ f) = (\lambda P\ s.\ wlp(f(g\ s))\ P\ s) \]
(proof)

Use simp add:wp_eval to fully unfold a program fragment


lemma Skip-Seq:
\[ Skip \ ;\ A = A \]
(proof)

lemma Seq-Skip:
\[ A \ ;\ Skip = A \]
(proof)

Use these as simp rules to clear out Skips

lemmas skip-simps = Skip-Seq Seq-Skip

end

4.2 Healthiness

theory Healthiness imports Embedding begin
4.2. 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:**

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

⟨proof⟩

**lemma nearly-healthy-wlp-Abort:**

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

⟨proof⟩

**lemma healthy-wp-Skip:**

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

⟨proof⟩

**lemma nearly-healthy-wlp-Skip:**

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

⟨proof⟩

**lemma healthy-wp-Seq:**

fixes \( t :: \text{'}s \text{ prog} \) and \( u \)

assumes \( \text{ht: healthy} \left( \wp t \right) \) and \( \text{hu: healthy} \left( \wp u \right) \)

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

⟨proof⟩

**lemma nearly-healthy-wlp-Seq:**

fixes \( t :: \text{'}s \text{ prog} \) and \( u \)

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

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

⟨proof⟩

**lemma healthy-wp-PC:**

fixes \( f :: \text{'}s \text{ prog} \)

assumes \( \text{kf: healthy} \left( \wp f \right) \) and \( \text{hg: healthy} \left( \wp g \right) \)

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

shows \[ \text{healthy} \left( \wp \left( f P \oplus g \right) \right) \]

⟨proof⟩

**lemma nearly-healthy-wlp-PC:**

fixes \( f :: \text{'}s \text{ prog} \)

assumes \( \text{kf: nearly-healthy} \left( \text{wlp} f \right) \)

and \( \text{hg: nearly-healthy} \left( \text{wlp} g \right) \)

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

shows \[ \text{nearly-healthy} \left( \text{wlp} \left( f P \oplus g \right) \right) \]

⟨proof⟩

**lemma healthy-wp-DC:**

fixes \( f :: \text{'}s \text{ prog} \)


assumes \( hf \): healthy (wp \( f \)) and \( hg \): healthy (wp \( g \))
shows healthy (wp \( f \land g \))
⟨proof⟩

lemma nearly-healthy-wlp-DC:
fixes \( f \)'s prog
assumes \( hf \): nearly-healthy (wp \( f \))
and \( hg \): nearly-healthy (wp \( g \))
shows nearly-healthy (wp \( f \land g \))
⟨proof⟩

lemma healthy-wp-AC:
fixes \( f \)'s prog
assumes \( hf \): healthy (wp \( f \)) and \( hg \): healthy (wp \( g \))
shows healthy (wp \( f \cup g \))
⟨proof⟩

lemma nearly-healthy-wlp-AC:
fixes \( f \)'s prog
assumes \( hf \): nearly-healthy (wp \( f \))
and \( hg \): nearly-healthy (wp \( g \))
shows nearly-healthy (wp \( f \cup g \))
⟨proof⟩

lemma healthy-wp-Embed:
\( \text{healthy } t \Rightarrow \text{healthy } (\text{wp } (\text{Embed } t)) \)
⟨proof⟩

lemma nearly-healthy-wlp-Embed:
nearly-healthy \( t \Rightarrow \text{nearly-healthy } (\text{wp } (\text{Embed } t)) \)
⟨proof⟩

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

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

lemma healthy-wp-SetDC:
fixes \( \text{prog } ':b \Rightarrow 'a \text{ prog and } S:'a \Rightarrow 'b \text{ set} \)
assumes healthy: \( \forall x. \, x \in S \Rightarrow \text{healthy } (\text{wp } (\text{prog } x)) \)
and nonempty: \( \exists x. \, x \in S \)
shows healthy (wp (SetDC \( \text{prog } S \))) (\( \text{is } \text{healthy } ?T \))
⟨proof⟩
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 (f s)))
  shows healthy (wp (Bind f p))
⟨proof⟩

lemma nearly-healthy-wlp-Bind:
  fixes f::'s ⇒ 'a
  assumes hsub: ∀s. nearly-healthy (wlp (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 \( hb: \text{healthy} (wp \ \text{body}) \)
and \( le: \text{le-trans} \ t \ u \)
and \( ht: \bigwedge P. \text{sound} \ P \Rightarrow \text{sound} (t \ P) \)
and \( hu: \bigwedge P. \text{sound} \ P \Rightarrow \text{sound} (u \ P) \)
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-loop-step-mono:
 fixes \( t \ u: \text{'s trans} \)
assumes \( mb: \text{nearly-healthy} (wlp \ \text{body}) \)
and \( le: \text{le-utrans} \ t \ u \)
and \( ht: \bigwedge P. \text{unitary} \ P \Rightarrow \text{unitary} (t \ P) \)
and \( hu: \bigwedge P. \text{unitary} \ P \Rightarrow \text{unitary} (u \ P) \)
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-loop-fp:
 assumes \( hb: \text{healthy} (wp \ \text{body}) \)
and \( sP: \text{sound} \ P \)
shows \( \lambda s. « G » \ s \star wp \ \text{body} \ (\lambda s. \text{bound-of} \ P) \ s + « N \ G » \ s \star P \ s \vdash \lambda s. \text{bound-of} \ P \)

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

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

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

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

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

\text{lemma} \ wlp-loop-step-unitary:
 fixes \( t \ u: \text{'s trans} \)
assumes \( hb: \text{nearly-healthy} (wlp \ \text{body}) \)
and \( ht: \bigwedge P. \text{unitary} \ P \Rightarrow \text{unitary} (t \ P) \)
and \( uP: \text{unitary} \ P \)
4.2. HEALTHINESS

shows unitary \((\text{wp } (\text{body } ;; \text{Embed } t \leftarrow G \oplus \text{Skip}) P)\)
⟨proof⟩

lemma wp-loop-step-sound:
fixes \(t, u::\text{'s trans}\)
assumes \(hb::\text{healthy } (\text{wp body})\)
and \(ht::\forall P. \text{sound } P \implies \text{sound } (t P)\)
and \(sP::\text{sound } P\)
shows \(\text{sound } (\text{wp } (\text{body } ;; \text{Embed } t \leftarrow G \oplus \text{Skip}) P)\)
⟨proof⟩

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

lemma wp-Loop1:
fixes \(body::\text{'s prog}\)
assumes \(sP::\text{sound } P\)
and \(hb::\text{healthy } (\text{wp body})\)
shows \(\text{wp } (do G \rightarrow body od) P = gfp-exp (\lambda Q s. \langle G \rangle s * wp body Q s + \langle \mathcal{N} G \rangle s * P s)\)
(is \(?X = gfp-exp (?Y P)\))
⟨proof⟩

lemma wp-loop-sound:
assumes \(sP::\text{sound } P\)
and \(hb::\text{healthy } (\text{wp body})\)
shows \(\text{sound } (wp do G \rightarrow body od P)\)
⟨proof⟩

Likewise, we can rewrite strict loops.

lemma wp-Loop1:
fixes \(body::\text{'s prog}\)
assumes \(sP::\text{sound } P\)
and \(hb::\text{healthy } (\text{wp body})\)
shows \(\text{wp } (do G \rightarrow body od) P = lfp-exp (\lambda Q s. \langle G \rangle s * wp body Q s + \langle \mathcal{N} G \rangle s * P s)\)
(is \(?X = lfp-exp (?Y P)\))
⟨proof⟩

lemma nearly-healthy-wlp-loop:
fixes \(body::\text{'s prog}\)
assumes \(hb::\text{nearly-healthy } (\text{wp body})\)
shows \(\text{nearly-healthy } (\text{wp } (do G \rightarrow body od))\)
⟨proof⟩

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

lemma healthy-wp-loop:
fixes \(body::\text{'s prog}\)
**CHAPTER 4. THE PGCL LANGUAGE**

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; \( \forall i. M i \vdash M (Suc i) \); \( \forall i. sound (M i) \); \( \forall i. bounded-by b (M i) \) ] →
t (Sup-exp (range M)) = Sup-exp (range (t o M))
(proof)

lemma bd-ctsI:
(\( \forall b. \forall i. M i \vdash M (Suc i) \) → \( \forall i. sound (M i) \) → (\( \forall i. bounded-by b (M i) \)) →
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)) →
4.3. CONTINUITY

\[
\text{equiv-trans } (T (\text{Sup-trans } (M \cdot \text{UNIV}))) (\text{Sup-trans } ((T \circ M) \cdot \text{UNIV}))
\]

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

**Lemma bd-cts-tr1:**
\[
(\bigwedge M. (\bigwedge i. \text{le-trans } (M \text{ suc } i)) \implies (\bigwedge i. \text{feasible } (M \text{ i })) \implies \\
\text{equiv-trans } (T (\text{Sup-trans } (M \cdot \text{UNIV}))) (\text{Sup-trans } ((T \circ M) \cdot \text{UNIV}))
\implies \text{bd-cts-tr } T
\]

4.3.1 Continuity of Primitives

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

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

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

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

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

**Lemma cts-wp-Seq:**
\[
\text{fixes } a b :: \text{'}s \text{ prog} \\
\text{assumes } ca: \text{bd-cts } (\text{wp } a)
\]
and cb: bd-cts (wp b)
and hb: healthy (wp b)
shows bd-cts (wp (a ;; b))
⟨proof⟩

lemma cts-wp-PC:
fixes a b::'s prog
assumes ca: bd-cts (wp a)
and cb: bd-cts (wp b)
and ha: healthy (wp a)
and hb: healthy (wp b)
and up: unitary p
shows bd-cts (wp (PC a p b))
⟨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.

lemma SetPC-Bind:
SetPC a p = Bind p (λp. SetPC a (λ-. p))
⟨proof⟩

lemma SetPC-remove:
assumes nz: p x ≠ 0 and n1: p x ≠ 1
and fsupp: finite (supp p)
shows SetPC a (λ-. p) = PC (a x) (λ-. p x) (SetPC a (λ-. dist-remove p x))
⟨proof⟩

lemma cts-bot:
bd-cts (λ(P::'s expect) (s::'s). 0::real)
⟨proof⟩

lemma wp-SetPC-nil:
wp (SetPC a (λs a. 0)) = (λP s. 0)
⟨proof⟩

lemma SetPC-sgl:
supp p = {x} ⇒ SetPC a (λ-. p) = (λab P s. p x * a x ab P s)
⟨proof⟩

lemma bd-cts-scale:
fixes a::'s trans
assumes ca: bd-cts a
and ha: healthy a
and nnc: 0 ≤ c
shows bd-cts (λP s. c * a P s)
⟨proof⟩
4.3. CONTINUITY

lemma cts-wp-SetPC-const:
  fixes a :: 'a ⇒ 's prog
  assumes ca: \( \forall x. \ x \in (\text{supp} \ p) \Rightarrow \text{bd-cts} (wp (a \ x)) \)
  and ha: \( \forall x. \ x \in (\text{supp} \ p) \Rightarrow \text{healthy} (wp (a \ x)) \)
  and up: unitary p
  and sump: \( \text{sum} \ p \ (\text{supp} \ p) \leq 1 \)
  and fsupp: finite (supp p)
  shows \( \text{bd-cts} (wp (\text{SetPC} \ a (\lambda -. \ p))) \)
⟨proof⟩

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

lemma wp-SetDC-Bind:
  \( \text{SetDC} \ a \ S = \text{Bind} \ S (\lambda S. \text{SetDC} \ a (\lambda -. \ S)) \)
⟨proof⟩

lemma SetDC-finite-insert:
  assumes fS: finite S
  and neS: S \( \neq \) \{\}
  shows \( \text{SetDC} \ a (\lambda -. \ \text{insert} \ x \ S) = a \ x \prod \text{SetDC} \ a (\lambda -. \ S) \)
⟨proof⟩

lemma SetDC-singleton:
  \( \text{SetDC} \ a (\lambda -. \ \{x\}) = a x \)
⟨proof⟩

lemma cts-wp-SetDC-const:
  fixes a :: 'a ⇒ 's prog
  assumes ca: \( \forall x. x \in S \Rightarrow \text{bd-cts} (wp (a x)) \)
  and ha: \( \forall x. x \in S \Rightarrow \text{healthy} (wp (a x)) \)
  and fS: finite S
  and neS: S \( \neq \) \{\}
  shows \( \text{bd-cts} (wp (\text{SetDC} \ a (\lambda -. \ S))) \)
⟨proof⟩

lemma cts-wp-SetDC:
  fixes a :: 'a ⇒ 's prog
  assumes ca: \( \forall x. x \in S \Rightarrow \text{bd-cts} (wp (a x)) \)
  and ha: \( \forall x. x \in S \Rightarrow \text{healthy} (wp (a x)) \)
  and fS: \( \forall s. \text{finite} (S s) \)
  and neS: \( \forall s. S s \neq \{\} \)
  shows \( \text{bd-cts} (wp (\text{SetDC} \ a (\lambda -. \ S))) \)
⟨proof⟩
show \textit{bd-cts} (\textit{wp} (SetDC a S))
\langle\text{proof}\rangle

\textbf{lemma \textit{cts-wp-repeat}:}
\textit{bd-cts} (\textit{wp} a) \implies \textit{healthy} (\textit{wp} a) \implies \textit{bd-cts} (\textit{wp} (\text{repeat} n a))
\langle\text{proof}\rangle

\textbf{lemma \textit{cts-wp-Embed}:}
\textit{bd-cts} t \implies \textit{bd-cts} (\textit{wp} (\text{Embed} t))
\langle\text{proof}\rangle

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

\textit{end}

\section{4.4 Continuity and Induction for Loops}

\textbf{theory LoopInduction imports Healthiness Continuity begin}

Showing continuity for loops requires a stronger induction principle than we have used so far, which in turn relies on the continuity of loops (inductively). Thus, the proofs are intertwined, and broken off from the main set of continuity proofs. This result is also essential in showing the sublinearity of loops.

A loop step is monotonic.

\textbf{lemma \textit{wp-loop-step-mono-trans}:}
\textit{fixes} body::'s prog
\textit{assumes} sP: \textit{sound} P
\textit{and} hb: \textit{healthy} (\textit{wp} body)
\textit{shows} \textit{mono-trans} (\lambda Q s. \textit{wp} body \in G s \oplus P s)
\langle\text{proof}\rangle

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

\textbf{lemma lfp-wp-loop-unfold:}
\textit{fixes} body::'s prog
\textit{assumes} hb: \textit{healthy} (\textit{wp} body)
\textit{and} sP: \textit{sound} P
4.4. CONTINUITY AND INDUCTION FOR LOOPS

shows \( \text{lfp-exp} (\lambda Q \ s. \ « \text{G} » \ s \ * \ wp \ body \ Q \ s + \ « \text{N} » \ G \ s \ * \ P \ s) \) =
\( (\lambda s. \ « \text{G} » \ s \ * \ wp \ body \ (\text{lfp-exp} (\lambda Q \ s. \ « \text{G} » \ s \ * \ wp \ body \ Q \ s + \ « \text{N} » \ G \ s \ * \ P \ s)) \ s + \ < \text{N} » \ G \ s \ * \ P \ s) \)
⟨ proof ⟩

lemma wp-loop-step-unitary:
 fixes body::'s prog
 assumes hb: \text{healthy} (wp body)
 and uP: \text{unitary} \ P \ and \ uQ: \text{unitary} \ Q
 shows \text{unitary} (\lambda s. \ « \text{G} » \ s \ * \ wp \ body \ Q \ s + \ < \text{N} » \ G \ s \ * \ P \ s)
⟨ proof ⟩

lemma lfp-loop-unitary:
 fixes body::'s prog
 assumes hb: \text{healthy} (wp body)
 and uP: \text{unitary} \ P
 shows \text{unitary} (lfp-exp (\lambda Q \ s. \ « \text{G} » \ s \ * \ wp \ body \ Q \ s + \ « \text{N} » \ G \ s \ * \ P \ s))
⟨ proof ⟩

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

lemma loop-induct:
 fixes body::'s prog
 assumes hwp: \text{healthy} (wp body)
 andhwlp: \text{nearly-healthy} (wlp body)
 — The body must be healthy, both in strict and liberal semantics.
 and \text{Limit}: \( \forall S. \ [ \forall x \in S. \ P (\text{fst} x) (\text{snd} x); \forall x \in S. \ \text{feasible} (\text{fst} x); \forall x \in S. \ \forall Q. \ \text{unitary} Q \rightarrow \text{unitary} (\text{snd} x Q) ] \rightarrow P (\text{Sup-trans} (\text{fst} ' S)) (\text{Inf-utrans} (\text{snd} ' S)) \)
 — The property holds at limit points.
 and \text{IH}: \( \forall t \ u. \ [ P (\text{body} ;; \text{Embed} t « \text{G} » + \text{Skip}) \rightarrow P (\text{wlp} (\text{body} ;; \text{Embed} u « \text{G} » + \text{Skip})) \)
 — The inductive step. The property is preserved by a single loop iteration.
 and \text{P-equiv}: \( \forall t \ u' \ u'. \ [ P (\text{body} ;; \text{Embed} t « \text{G} » + \text{Skip}) \rightarrow P (\text{equiv-utrans} t u') \rightarrow P (\text{equiv-utrans} u u') \) \rightarrow P t'
 — The property must be preserved by equivalence
 shows \( P (\text{wp} (\text{do} \text{G} \rightarrow \text{body} \text{od})) (\text{wlp} (\text{do} \text{G} \rightarrow \text{body} \text{od})) \)
 — The property can refer to both interpretations simultaneously. The unifier will happily apply the rule to just one or the other, however.
⟨ proof ⟩

4.4.1 The Limit of Iterates

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

**definition** iterates :: 's 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]:

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

⟨proof⟩

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

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

⟨proof⟩

All iterates are healthy.

**lemma** iterates-healthy:

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

⟨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) −→ 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
  cts-wp-Seq cts-wp-PC
  cts-wp-DC cts-wp-Embed
  cts-wp-Apply cts-wp-SetDC
  cts-wp-SetPC cts-wp-Bind
  cts-wp-repeat
end
```

4.5 **Sublinearity**

```
theory Sublinearity imports Embedding Healthiness LoopInduction begin
```

4.5.1 Nonrecursive Primitives

Sublinearity of non-recursive programs is generally straightforward, and follows from the algebraic properties of the underlying operations, together with healthiness.

```
lemma sublinear-wp-Skip:
  sublinear (wp Skip)
⟨proof⟩
```

```
lemma sublinear-wp-Abort:
```
CHAPTER 4. THE PGCL LANGUAGE

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)
shows 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)
shows sublinear (wp (x ∩ y))
⟨proof⟩

As for continuity, we insist on a finite support.

lemma sublinear-wp-SetPC:
fixes p::'a ⇒ 's prog
assumes slp: \( \forall s. a \in \text{supp} (P s) \implies \text{sublinear} (wp (p a)) \)
and sum: \( \forall s. (\sum a \in \text{supp} (P s). P s a) \leq 1 \)
and nnP: \( \forall s. 0 \leq P s a \)
and fin: \( \forall s. \text{finite} (\text{supp} (P s)) \)
shows sublinear (wp (SetPC p P))
⟨proof⟩

lemma sublinear-wp-SetDC:
fixes p::'a ⇒ 's prog
assumes slp: \( \forall s. a \in S s \implies \text{sublinear} (wp (p a)) \)
and hp: \( \forall s a. a \in S s \implies \text{healthy} (wp (p a)) \)
and ne: \( \forall s. S s \neq \emptyset \)
shows sublinear (wp (SetDC p S))
⟨proof⟩

lemma sublinear-wp-Embed:
sublinear t \implies \text{sublinear} (wp (Embed t))
⟨proof⟩
4.5. **SUBLINEARITY**

```
lemma sublinear-wp-repeat:
  [ sublinear (wp p); healthy (wp p) ] \implies sublinear (wp (repeat n p))
 ⟨proof⟩
```

```
lemma sublinear-wp-Bind:
  [ ∀s. sublinear (wp (a (f s))) ] \implies sublinear (wp (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 (wlp body)
  shows sub-distrib (wp (do G \rightarrow 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 \rightarrow body od))
 ⟨proof⟩
```

```
lemma sublinear-wp-loop:
  fixes body::'s prog
  assumes hb: healthy (wp body)
  and nhb: nearly-healthy (wlp body)
  and sb: sublinear (wp body)
  and cb: bd-cts (wp body)
  shows sublinear (wp (do G \rightarrow body od))
 ⟨proof⟩
```

```
lemmas sublinear-intros =
```

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) ] \rightarrow additive (wp (a ⊕ b))
4.6. DETERMINISM

DC is not additive.

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

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

**lemma** additive-wp-Embed:
\[ \text{additive t} \implies \text{additive (wp (Embed t))} \]

**lemma** additive-wp-repeat:
\[ \text{additive (wp a)} \implies \text{well-def a} \implies \text{additive (wp (repeat n a))} \]

**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:
\[ \text{maximal (wp Skip)} \]

**lemma** max-wp-Apply:
\[ \text{maximal (wp (Apply f))} \]

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

**lemma** max-wp-PC:
\[ \text{maximal (wp a)} ; \text{maximal (wp b)} \implies \text{maximal (wp (a P b))} \]

**lemma** max-wp-DC:
\[ \text{maximal (wp a)} ; \text{maximal (wp b)} \implies \text{maximal (wp (a \cap b))} \]
lemma max-wp-SetPC:
\[ \forall s \ a \in \text{supp} (P s) \implies \text{maximal (wp } (p a)) \]; \sa \ (\sum a\in\text{supp} (P s). P s a) = 1 \implies \text{maximal (wp (SetPC } p P))

lemma max-wp-SetDC:
fixes p:’a \Rightarrow ’s prog
assumes wp: \forall s \ a \in S s \implies \text{maximal (wp } (p a))
and ne: \forall s. S s \neq \{} \shows \text{maximal (wp (SetDC } p S))

lemma max-wp-Embed:
\text{maximal } t \implies \text{maximal (wp (Embed } t))

lemma max-wp-repeat:
\text{maximal (wp } a \implies \text{maximal (wp (repeat } n a))

lemma max-wp-Bind:
assumes ma: \forall s. \text{maximal (wp (a (f s)))}
shows \text{maximal (wp ((Bind f } a))

lemmas max-intros =
max-wp-Skip max-wp-Apply
max-wp-Seq max-wp-PC
max-wp-DC max-wp-SetPC
max-wp-SetDC max-wp-Embed
max-wp-Bind max-wp-repeat

A healthy transformer that terminates is maximal.

lemma healthy-term-max:
assumes ht: healthy t
and trm: \\lambda s. 1 \vdash t (\lambda s. 1)
shows \text{maximal } t

4.6.3 Determinism

lemma det-wp-Skip:
determ (wp Skip)

lemma det-wp-Apply:
determ (wp (Apply f))
lemmas determ-intros =
det-wp-Skip det-wp-Apply det-wp-Seq det-wp-PC det-wp-SetPC det-wp-Bind det-wp-Embed det-wp-repeat

end

4.7 Well-Defined Programs.

theory WellDefined imports
  Healthiness Sublinearity LoopInduction
begin

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

4.7.1 Strict Implies Liberal

This establishes the first connection between the strict and liberal interpretations ($wp$ and $wlp$).

**definition**

$\text{wp-under-wlp} :: \text{\textquotesingle}s\ \text{prog}\Rightarrow\text{bool}$

**where**

$\text{wp-under-wlp} \ \text{prog} \equiv \forall \ \text{P. unitary} \ \text{P} \rightarrow \ wp \ \text{prog} \ P \vdash \ wlp \ \text{prog} \ P$

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

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

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

$[ \ wp\text{-under-wlp} \ \text{prog}; \ \text{unitary} \ \text{P} ] \Rightarrow \ wp \ \text{prog} \ P \vdash \ wlp \ \text{prog} \ P$

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

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

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

$\text{wp-under-wlp} \ \text{Abort}$

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

$\text{wp-under-wlp} \ \text{Skip}$

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

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

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

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

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

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

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

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

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

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

assumes $\text{h-wp-a: healthy} \ (wp \ a)$
and h-wlp-a: nearly-healthy (wlp a)
and h-wp-b: healthy (wp b)
and h-wlp-b: nearly-healthy (wlp b)
and wp-a-a: wp-under-wlp a
and wp-a-b: wp-under-wlp b
and uP: unitary P
shows wp-under-wlp (a ⊕ b)
⟨proof⟩

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

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

lemma wp-under-wlp-SetDC:
assumes wp-u-f: \( \forall s. a \in S s \implies wp-under-wlp(f a) \)
and hf: \( \forall s. a \in S s \implies \text{healthy}(wp(f a)) \)
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:
[ healthy (wp a); nearly-healthy (wlp a); wp-under-wlp a ] \implies
wp-under-wlp (repeat n a)
⟨proof⟩

lemma wp-under-wlp-Bind:
[ \( \forall s. \text{wp-under-wlp}(a (f s)) \) ] \implies wp-under-wlp (Bind f a)
⟨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 &amp; wp prog Q ⊢ wp prog (P &amp; Q)

lemma sub-distrib-pconjI[intro]:
[∀ P Q. [ unitary P; unitary Q ] ⇒
 wlp prog P &amp; 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⟩
4.7. WELL-DEFINED PROGRAMS.

**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 [] 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 ⊕ b)

(proof)

**Lemma sdp-Embed:**

\[
\bigwedge P\ Q. \left[\left[\text{unitary } P\right] \land \left[\text{unitary } Q\right]\right] \implies t\ P \land\&\land t\ Q \lor t\ (P \land\&\land Q)\right]\implies sub-distrib-pconj (Embed t)
\]

(proof)

**Lemma sdp-repeat:**

fixes a::'s prog

assumes sdpa: 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 ⇒ 's prog

assumes sdp: \(\bigwedge s\ a.\ a\in\supp\ (P\ s)\implies\text{sub-distrib-pconj}\ (p\ a)\)
and fin: \(\bigwedge s.\ \text{finite}\ (\supp\ (P\ s))\)
and nnp: \(\bigwedge s\ a.\ 0\leq P\ s\ a\)
and sub: \(\bigwedge s.\ \sum\ (P\ s)\ (\supp\ (P\ s))\leq 1\)

shows sub-distrib-pconj (SetPC p P)

(proof)

**Lemma sdp-SetDC:**

fixes p::a ⇒ 's prog

assumes sdp: \(\bigwedge s\ a.\ a\in S\ s\implies\text{sub-distrib-pconj}\ (p\ a)\)
and hwp: \(\bigwedge s\ a.\ a\in S\ s\implies\text{healthy}\ (wp\ (p\ a))\)
and hwlp: \(\bigwedge s\ a.\ a\in S\ s\implies\text{nearly-healthy}\ (wlp\ (p\ a))\)
and nc: \(\bigwedge s.\ S\ s\neq\{\}\)
shows sub-distrib-pconj (SetDC p S)
⟨proof⟩

lemma sdp-Bind:

\[ \forall s. \text{sub-distrib-pconj} (p \ (f \ s)) \implies \text{sub-distrib-pconj} (\text{Bind} \ f \ p) \]
⟨proof⟩

For loops, we again appeal to our transfinite induction principle, this time taking advantage of the simultaneous treatment of both strict and liberal transformers.

lemma sdp-loop:

fixes body :: 's prog
assumes sdp-body: sub-distrib-pconj body
         and hwlp: nearly-healthy (wlp body)
         and hwp: healthy (wp body)
shows sub-distrib-pconj (do G → body od)
⟨proof⟩

lemmas sdp-intros =
    sdp-Abort sdp-Skip sdp-Apply
    sdp-Seq sdp-DC sdp-PC
    sdp-SetPC sdp-SetDC sdp-Embed
    sdp-repeat sdp-Bind sdp-loop

4.7.3 The Well-Defined Predicate.

definition

well-def :: 's prog ⇒ bool
where

well-def prog ≡ healthy (wp prog) ∧ nearly-healthy (wlp prog)
∧ wp-under-wlp prog ∧ sub-distrib-pconj prog
∧ sublinear (wp prog) ∧ bd-cts (wp prog)

lemma well-defI[intro]:

\[ \text{healthy} (\text{wp} \ \text{prog}); \ \text{nearly-healthy} (\text{wlp} \ \text{prog});
\ \text{wp-under-wlp} \ \text{prog}; \ \text{sub-distrib-pconj} \ \text{prog}; \ \text{sublinear} (\text{wp} \ \text{prog});
\ \text{bd-cts} (\text{wp} \ \text{prog}) \implies \text{well-def} \ \text{prog} \]
⟨proof⟩

lemma well-def-wp-healthy[dest]:

well-def prog ⇒ healthy (wp prog)
⟨proof⟩

lemma well-def-wlp-nearly-healthy[dest]:

well-def prog ⇒ nearly-healthy (wlp prog)
⟨proof⟩

lemma well-def-wp-under[dest]:
4.7. WELL-DEFINED PROGRAMS.

well-def prog \implies wp-under-wlp prog
\langle \text{proof} \rangle

lemma well-def-sdp[dest]:
\begin{align*}
\text{well-def prog} & \implies \text{sub-distrib-pconj prog} \\
\langle \text{proof} \rangle
\end{align*}

lemma well-def-wp-sublinear[dest]:
\begin{align*}
\text{well-def prog} & \implies \text{sublinear (wp prog)} \\
\langle \text{proof} \rangle
\end{align*}

lemma well-def-wp-cts[dest]:
\begin{align*}
\text{well-def prog} & \implies \text{bd-cts (wp prog)} \\
\langle \text{proof} \rangle
\end{align*}

lemmas wd-dests =
\begin{align*}
\text{well-def-wp-healthy} & \text{well-def-wlp-nearly-healthy} \\
\text{well-def-wp-under} & \text{well-def-sdp} \\
\text{well-def-wp-sublinear} & \text{well-def-wp-cts}
\end{align*}

lemma wd-Abort:
\begin{align*}
\text{well-def Abort} \\
\langle \text{proof} \rangle
\end{align*}

lemma wd-Skip:
\begin{align*}
\text{well-def Skip} \\
\langle \text{proof} \rangle
\end{align*}

lemma wd-Apply:
\begin{align*}
\text{well-def (Apply f)} \\
\langle \text{proof} \rangle
\end{align*}

lemma wd-Seq:
\begin{align*}
[ \text{well-def a}; \text{well-def b} ] & \implies \text{well-def (a ;; b)} \\
\langle \text{proof} \rangle
\end{align*}

lemma wd-PC:
\begin{align*}
[ \text{well-def a}; \text{well-def b}; \text{unitary P} ] & \implies \text{well-def (a p\uplus b)} \\
\langle \text{proof} \rangle
\end{align*}

lemma wd-DC:
\begin{align*}
[ \text{well-def a}; \text{well-def b} ] & \implies \text{well-def (a \cap b)} \\
\langle \text{proof} \rangle
\end{align*}

lemma wd-SetDC:
\begin{align*}
[ \forall x s. x \in S \iff \text{well-def (a x)}; \exists s. S \neq \{\}; \exists s. \text{finite (S s)} ] & \implies \text{well-def (SetDC a S)} \\
\langle \text{proof} \rangle
\end{align*}
4.8 The Loop Rules

theory Loops imports WellDefined begin

Given a well-defined body, we can annotate a loop using an invariant, just as in the classical setting.

4.8.1 Liberal and Strict Invariants.

A probabilistic invariant generalises a boolean one: it entails itself, given the loop guard.

definition wp-inv :: (′s ⇒ bool) ⇒ ′s prog ⇒ (′s ⇒ real) ⇒ bool

where

wp-inv G body I = (∀ s. «G» s * I s ≤ wp body I s)
4.8. THE LOOP RULES

**lemma** \( \text{wp-invI} \):
\[
\forall I. (\forall s. \langle s \rangle s \leq \text{wp body } I s) \implies \text{wp-inv } G \text{ body } I
\]
\(\langle \text{proof}\rangle\)

**definition**
\( \text{wlp-inv} :: (\forall s. \langle s \rangle s \leq \text{wp body } I s) \implies \text{wp-inv } G \text{ body } I \)

**lemma** \( \text{wlp-invI} \):
\[
\forall I. (\forall s. \langle s \rangle s \leq \text{wp body } I s) \implies \text{wlp-inv } G \text{ body } I
\]
\(\langle \text{proof}\rangle\)

**lemma** \( \text{wlp-invD} \):
\[
\text{wlp-inv } G \text{ body } I \implies \langle s \rangle s \leq \text{wp body } I s
\]
\(\langle \text{proof}\rangle\)

For standard invariants, the multiplication reduces to conjunction.

**lemma** \( \text{wp-inv-stdD} \):
\[
\text{assumes inv: wp-inv } G \text{ body } \langle I \rangle
\]
\[
\text{and } h: \text{healthy } (\text{wp body})
\]
\[
\text{shows } \langle G \rangle \& \& \langle I \rangle \vdash \text{wp body } \langle I \rangle
\]
\(\langle \text{proof}\rangle\)

### 4.8.2 Partial Correctness


**lemma** \( \text{wlp-Loop} \):
\[
\text{assumes wd: well-def body}
\]
\[
\text{and uI: unitary } I
\]
\[
\text{and inv: wlp-inv } G \text{ body } I
\]
\[
\text{shows } I \leq \text{wlp do } G \implies \text{body od } (\forall s. \langle N \rangle s \leq \text{wlp body } I s)
\]
\(\langle \text{proof}\rangle\)

### 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** \( \text{wp-Loop} \):
\[
\text{assumes wd: well-def body}
\]
\[
\text{and inv: wlp-inv } G \text{ body } I
\]
\[
\text{and unit: unitary } I
\]
\[
\text{shows } I \& \& \text{wp } (\text{do } G \implies \text{body od}) (\forall s. I s) \vdash \text{wp } (\text{do } G \implies \text{body od}) (\forall s. \langle N \rangle s \leq \text{wlp body } I s)
\]
\(\langle \text{proof}\rangle\)
4.8.4 Unfolding

lemma wp-loop-unfold:
  fixes body :: 's prog
  assumes sP: sound P
  and h: healthy (wp body)
  shows wp (do G -> body od) P =
(λs. «N G» s * P s + «G» s * wp body (wp (do G -> body od) P) s)

⟨proof⟩

lemma wp-loop-nguard:
[ [ healthy (wp body); sound P; ¬ G s ] ] =⇒ wp do G -> body od P s = P s
⟨proof⟩

lemma wp-loop-guard:
[ [ healthy (wp body); sound P; G s ] ] =⇒ wp do G -> body od P s = wp (body ;; do G -> body od) P s
⟨proof⟩

end

4.9 The Algebra of pGCL

theory Algebra imports WellDefined begin

Programs in pGCL have a rich algebraic structure, largely mirroring that for GCL. We show that programs form a lattice under refinement, with a ∩ b and a ∪ b as the meet and join operators, respectively. We also take advantage of the algebraic structure to establish a framework for the modular decomposition of proofs.

4.9.1 Program Refinement

Refinement in pGCL relates to refinement in GCL exactly as probabilistic entailment relates to implication. It turns out to have a very similar algebra, the rules of which we establish shortly.

definition refines :: 's prog ⇒ 's prog ⇒ bool (infix ⊑)
where
prog ⊑ prog’ ≡ ∀ P. sound P ⇒ wp prog P ⇒ wp prog’ P

lemma refines[intro]:
[ [ P. sound P ⇒ wp prog P ⇒ wp prog’ P ] ] =⇒ prog ⊑ prog’
⟨proof⟩
4.9. THE ALGEBRA OF PGCL

lemma refinesD[dest]:
\[
\begin{align*}
\text{[ prog } &\sqsupseteq \text{ prog}; \text{ sound } P ] \implies \text{ wp prog } P \neq \text{ wp prog'} P \\
\end{align*}
\]
(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 ≃ 70)
where
\[
\text{prog } \simeq \text{ prog'} \equiv \forall P, \text{ sound } P \implies \text{ wp prog } P = \text{ wp prog'} P
\]

lemma pequivI[intro]:
\[
\begin{align*}
\text{[ } &\forall P. \text{ sound } P \implies \text{ wp prog } P = \text{ wp prog'} P \text{ ] } \implies \text{ prog } \simeq \text{ prog'}
\end{align*}
\]
(proof)

lemma pequivD[dest,simp]:
\[
\begin{align*}
\text{[ prog } &\simeq \text{ prog'}; \text{ sound } P ] \implies \text{ wp prog } P = \text{ wp prog'} P
\end{align*}
\]
(proof)

lemma pequiv-equiv-trans:
\[
\begin{align*}
a \simeq b \iff \text{ equiv-trans } (\text{ wp } a) (\text{ wp } b)
\end{align*}
\]
(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]:
\[
\begin{align*}
a \sqcap b &= b \sqcap a
\end{align*}
\]
(proof)

lemma DC-assoc[ac-simps]:
\[
\begin{align*}
a \sqcap (b \sqcap c) &= (a \sqcap b) \sqcap c
\end{align*}
\]
(proof)

lemma DC-idem:
\[
\begin{align*}
a \sqcap a &= a
\end{align*}
\]
(proof)

lemma AC-comm[ac-simps]:
\[
\begin{align*}
a \sqcup b &= b \sqcup a
\end{align*}
\]
(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 PC-idem:
\[ a p \oplus a = a \]

lemma Seq-assoc[ac-simps]:
\[ A ;; (B ;; C) = A ;; B ;; C \]

lemma Abort-refines[intro]:
\[ \text{well-def } a \implies \text{Abort } \subseteq a \]

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:
\[ \text{fixes } a::'s \text{ prog and } b \text{ and } c \]
\[ \text{assumes } rab: a \subseteq b \text{ and } rac: a \subseteq c \]
\[ \text{shows } a \subseteq (b \cap c) \]

lemma DC-mono:
\[ \text{fixes } a::'s \text{ prog} \]
\[ \text{assumes } rab: a \subseteq b \text{ and } rcd: c \subseteq d \]
\[ \text{shows } (a \cap c) \subseteq (b \cap d) \]

Laws relating angelic choice and refinement

lemma left-refines-AC:
\[ a \subseteq (a \bigcup b) \]
lemma right-refines-AC:
  \( b \subseteq (a \bigvee b) \)
  \( \langle \text{proof} \rangle \)

lemma AC-refines:
  fixes a::'s prog and b and c
  assumes rac: \( a \subseteq c \) and rbc: \( b \subseteq c \)
  shows \( (a \bigvee b) \subseteq c \)
  \( \langle \text{proof} \rangle \)

lemma AC-mono:
  fixes a::'s prog
  assumes rab: \( a \subseteq b \) and rcd: \( c \subseteq d \)
  shows \( (a \bigvee c) \subseteq (b \bigvee d) \)
  \( \langle \text{proof} \rangle \)

Laws depending on the arithmetic of \( a \bigoplus b \) and \( a \bigcap b \) together

lemma DC-refines-PC:
  assumes unit: unitary p
  shows \( (a \bigcap b) \subseteq (a \bigoplus b) \)
  \( \langle \text{proof} \rangle \)

Laws depending on the arithmetic of \( a \bigoplus b \) and \( a \bigvee b \) together

lemma PC-refines-AC:
  assumes unit: unitary p
  shows \( (a \bigoplus b) \subseteq (a \bigvee b) \)
  \( \langle \text{proof} \rangle \)

Laws depending on the arithmetic of \( a \bigvee b \) and \( a \bigcap b \) together

lemma DC-refines-AC:
  \( (a \bigcap b) \subseteq (a \bigvee b) \)
  \( \langle \text{proof} \rangle \)

Laws Involving Refinement and Equivalence

lemma pr-trans[trans]:
  fixes A::'a prog
  assumes prAB: \( A \subseteq B \)
    and prBC: \( B \subseteq C \)
  shows \( A \subseteq C \)
  \( \langle \text{proof} \rangle \)

lemma pequiv-refl[intro,simp]:
  \( a \simeq a \)
  \( \langle \text{proof} \rangle \)
lemma pequiv-comm[ac-simps]:
\[ a \simeq b \iff b \simeq a \]
\(\langle proof\rangle\)

lemma pequiv-pr[dest]:
\[ a \simeq b \implies a \subseteq b \]
\(\langle proof\rangle\)

lemma pequiv-trans[intro,trans]:
\[ [a \simeq b; b \simeq c] \implies a \simeq c \]
\(\langle proof\rangle\)

lemma pequiv-pr-trans[intro,trans]:
\[ [a \subseteq b; b \simeq c] \implies a \subseteq c \]
\(\langle proof\rangle\)

lemma pr-pequiv-trans[intro,trans]:
\[ [a \subseteq b; b \simeq c] \implies a \subseteq c \]
\(\langle proof\rangle\)

Refinement induces equivalence by antisymmetry:

lemma pequiv-antisym:
\[ [a \subseteq b; b \subseteq a] \implies a \simeq b \]
\(\langle proof\rangle\)

lemma pequiv-DC:
\[ [a \simeq c; b \simeq d] \implies (a \sqcap b) \simeq (c \sqcap d) \]
\(\langle proof\rangle\)

lemma pequiv-AC:
\[ [a \simeq c; b \simeq d] \implies (a \sqcup b) \simeq (c \sqcup d) \]
\(\langle proof\rangle\)

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:
\[
\text{fixes } a::'s\ prog\\
\text{assumes } da: \text{ determ } (wp\ a)\\
\text{and } wa: \text{ well-def } a\\
\text{and } wb: \text{ well-def } b\\
\text{and } dr: a \subseteq b\\
\text{shows } a \simeq b
\]

\(\langle proof\rangle\)

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 \( \text{Abort} \) is bottom, and \( a \sqcap b \) is \( \text{inf} \). There is no unique top element, but all fully-deterministic programs are maximal.

The type that we construct here is not especially useful, but serves as a convenient way to express this result.

\[
\text{quotient-type} \ 's \text{ program} = \\
' s \text{ prog} / \text{ partial} : \lambda a \ b. \ a \simeq b \wedge \text{well-def } a \wedge \text{well-def } b
\]

\[
\text{instantiation} \ program :: (\text{type}) \ \text{semilattice-inf} \ \begin{proof}
\text{lift-definition} \less-eq-program :: 'a \text{ program} \Rightarrow 'a \text{ program} \Rightarrow \text{bool} \ \text{is refines}
\end{proof}
\]

\[
\text{lift-definition} \less-program :: 'a \text{ program} \Rightarrow 'a \text{ program} \Rightarrow \text{bool}
\text{is} \ \lambda a \ b. \ a \sqsubseteq b \wedge \neg b \sqsubseteq a
\]

\[
\text{lift-definition} \inf-program :: 'a \text{ program} \Rightarrow 'a \text{ program} \Rightarrow 'a \text{ program} \ \text{is DC}
\end{proof}
\]

\[
\text{instance} \ \begin{proof}
\end{proof}
\]

\[
\text{instantiation} \ program :: (\text{type}) \ \text{bot} \ \begin{proof}
\text{lift-definition} \bot-program :: 'a \text{ program} \ \text{is} \ \text{Abort}
\end{proof}
\]

\[
\text{instance} \ \begin{proof}
\end{proof}
\]

\[
\text{lemma eq-det:} \ (a \ b:'s \ \text{prog}. \ [ \ a \simeq b; \ \text{determ } (wp a) ] \implies \ \text{determ } (wp b)
\]

\[
\text{lift-definition} \ pdeterm :: 's \ \text{program} \Rightarrow \text{bool}
\text{is} \ \lambda \ a. \ \text{determ } (wp \ a)
\]

\[
\text{lemma determ-maximal:}
\ [ \ pdeterm \ a; \ a \leq x ] \implies a = x
\]

\[
\]
4.9.5 Data Refinement

A projective data refinement construction for pGCL. By projective, we mean that the abstract state is always a function ($\varphi$) of the concrete state. Refinement may be predicated ($G$) on the state.

**Definition**

$drefines :: (\texttt{'b \Rightarrow \texttt{'a}}) \Rightarrow (\texttt{'b \Rightarrow \texttt{bool}}) \Rightarrow \texttt{'a prog} \Rightarrow \texttt{'b prog} \Rightarrow \texttt{bool}$

**Where**

$drefines \varphi G A B \equiv \forall P Q. \ (\text{unitary } P \land \text{unitary } Q \land (P \vdash wp A Q)) \implies (\langle G \rangle \land (P \circ \varphi) \vdash wp B (Q \circ \varphi))$

**Lemma** $drefinesD[dest]$: \[
\text{\texttt{\{ drefines } \varphi G A B; \text{unitary } P; \text{unitary } Q; P \vdash wp A Q \text{\}\implies \langle G \rangle \land (P \circ \varphi) \vdash wp B (Q \circ \varphi)\}}
\]

(proof)

We can alternatively use $G$ as an assumption:

**Lemma** $drefinesD2$: \[
\text{\texttt{\{ drefines } \varphi G A B; \text{unitary } P; \text{unitary } Q; P \vdash wp A Q \text{\}\implies \langle G \rangle \land (P \circ \varphi) \vdash wp B (Q \circ \varphi)\}}
\]

(proof)

This additional form is sometimes useful:

**Lemma** $drefinesD3$: \[
\text{\texttt{\{ drefines } \varphi G a b; \text{unitary } G; \text{unitary } Q; \text{well-def } a \\}\implies \langle G \rangle \land (P \circ \varphi) \vdash wp b (Q \circ \varphi) s \}}
\]

(proof)

**Lemma** $drefinesI[intro]$: \[
\text{\texttt{\{ \forall P Q. \{ \text{unitary } P; \text{unitary } Q; P \vdash wp A Q \}\implies \langle G \rangle \land (P \circ \varphi) \vdash wp B (Q \circ \varphi) \}\implies drefines } \varphi G A B\}}
\]

(proof)

Use $G$ as an assumption, when showing refinement:

**Lemma** $drefinesI2$: \[
\text{\texttt{\{ A::'a prog \land B::'b prog \land }\varphi::'b \Rightarrow 'a \land G::'b \Rightarrow \texttt{bool} \\}\implies \langle G \rangle \land (P \circ \varphi) \vdash wp b (Q \circ \varphi) s \}}
\]

(proof)
and withAs:
\[ \forall P \ Q \ s. \ [ \ \text{unitary } P; \ \text{unitary } Q; \ G \ s; \ P \vdash \ wp \ A \ Q ] \implies (P \circ \varphi) \ s \leq wp \ B \ (Q \circ \varphi) \ s \]
shows drefines \( \varphi \) \( G \ A \ B \)

(proof)

**lemma** dr-strengthen-guard:
fixes \( a::'s \ \text{prog} \) and \( b::'t \ \text{prog} \)
assumes \( fg: \ \forall s. \ F \ s \implies G \ s \)
and \( drab: \ \text{drefines } \varphi \) \( G \ a \ b \)
shows drefines \( \varphi \) \( F \ a \ b \)

(proof)

Probabilistic correspondence, \( pcorres \), is equality on distribution transformers, modulo a guard. It is the analogue, for data refinement, of program equivalence for program refinement.

definition pcorres :: \( ('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow \text{bool}) \Rightarrow 'a \ \text{prog} \Rightarrow 'b \ \text{prog} \Rightarrow \text{bool} \)
where
\( \text{pcorres } \varphi \) \( G \ A \ B \leftarrow\rightarrow\)
(\( \forall Q. \ \text{unitary } Q \implies \langle G \rangle \ \&\& \ (wp \ A \ Q \circ \varphi) = \langle G \rangle \ \&\& \ wp \ B \ (Q \circ \varphi) \)\)

**lemma** pcorres\(_I\):
\[ \[ \forall Q. \ \text{unitary } Q \implies \langle G \rangle \ \&\& \ (wp \ A \ Q \circ \varphi) = \langle G \rangle \ \&\& \ wp \ B \ (Q \circ \varphi) \] \implies \text{pcorres } \varphi \) \( G \ A \ B \)

(proof)

Often easier to use, as it allows one to assume the precondition.

**lemma** pcorres\(_{I2}[\text{intro}]\):
fixes \( A::'a \ \text{prog} \) and \( B::'b \ \text{prog} \)
assumes \( \text{withG: } \forall Q \ s. \ [ \ \text{unitary } Q; \ G \ s] \implies wp \ A \ Q \ (\varphi \ s) = wp \ B \ (Q \circ \varphi) \ s \)
and \( wA: \ \text{well-def } A \)
and \( wB: \ \text{well-def } B \)
shows pcorres \( \varphi \) \( G \ A \ B \)

(proof)

**lemma** pcorres\(_D\):
\[ \[ \text{pcorres } \varphi \) \( G \ A \ B; \ \text{unitary } Q \] \implies \langle G \rangle \ \&\& \ (wp \ A \ Q \circ \varphi) = \langle G \rangle \ \&\& \ wp \ B \ (Q \circ \varphi) \]

(proof)

Again, easier to use if the precondition is known to hold.

**lemma** pcorres\(_D2\):
assumes \( \text{pc: } \text{pcorres } \varphi \) \( G \ A \ B \)
and \( wQ: \ \text{unitary } Q \)
and \( wA: \ \text{well-def } A \) and \( wB: \ \text{well-def } B \)
and \( G: \ G \ s \)
shows wp \ A \ Q \ (\varphi \ s) = wp \ B \ (Q \circ \varphi) \ s \)

(proof)
4.9.6 The Algebra of Data Refinement

Program refinement implies a trivial data refinement:

\textbf{lemma} \textit{refines-drefines}:

- \texttt{fixes} \texttt{a::s prog}
- \texttt{assumes} \texttt{rab: a ⊑ b and wb: well-def b}
- \texttt{shows} \texttt{drefines (λs. s) G a b}

\langle proof \rangle

Data refinement is transitive:

\textbf{lemma} \textit{dr-trans[trans]}:

- \texttt{fixes} \texttt{A::a prog and B::b prog and C::c prog}
- \texttt{assumes} \texttt{drAB: drefines ϕ G A B}
- \texttt{and} \texttt{drBC: drefines ϕ' G' B C}
- \texttt{and Gimp: \(\forall s. G' s \implies G (ϕ' s)\)}
- \texttt{shows} \texttt{drefines (ϕ o ϕ') G' A C}

\langle proof \rangle

Data refinement composes with program refinement:

\textbf{lemma} \textit{pr-dr-trans[trans]}:

- \texttt{assumes} \texttt{prAB: A ⊑ B}
- \texttt{and} \texttt{drBC: drefines ϕ G B C}
- \texttt{shows} \texttt{drefines ϕ G A C}

\langle proof \rangle

\textbf{lemma} \textit{dr-pr-trans[trans]}:

- \texttt{assumes} \texttt{drAB: drefines ϕ G A B}
- \texttt{assumes} \texttt{prBC: B ⊑ C}
- \texttt{shows} \texttt{drefines ϕ G A C}

\langle proof \rangle

If the projection \(ϕ\) commutes with the transformer, then data refinement is reflexive:

\textbf{lemma} \textit{dr-refl}:

- \texttt{assumes} \texttt{wa: well-def a}
- \texttt{and} \texttt{comm: \(\forall Q. \text{unitary } Q \implies wp a Q o ϕ \vdash wp a (Q o ϕ)\)}
- \texttt{shows} \texttt{drefines ϕ G a a}

\langle proof \rangle

Correspondence implies data refinement

\textbf{lemma} \textit{pcorres-drefine}:

- \texttt{assumes} \texttt{corres: pcorres ϕ G A C}
- \texttt{and} \texttt{wC: well-def C}
- \texttt{shows} \texttt{drefines ϕ G A C}

\langle proof \rangle

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** drefines-determ:

**fixes** a::'a prog and b::'b prog

**assumes** da: determ (wp a)
and wa: well-def a
and wb: well-def b
and dr: drefines \( \varphi \) G a b

**shows** pcorres \( \varphi \) G a b

The proof follows exactly the same form as that for program refinement: Assuming that correspondence doesn’t hold, we show that wp b is not feasible, and thus not healthy, contradicting the assumption.

\[ \langle proof \rangle \]

4.9.7 Structural Rules for Correspondence

**lemma** pcorres-Skip:

pcorres \( \varphi \) G Skip Skip

\[ \langle proof \rangle \]

Correspondence composes over sequential composition.

**lemma** pcorres-Seq:

**fixes** A::'b prog and B::'c prog
and C::'b prog and D::'c prog
and \( \varphi ::'c \Rightarrow 'b \)

**assumes** pcAB: pcorres \( \varphi \) G A B
and pcCD: pcorres \( \varphi \) H C D
and wA: well-def A and wB: well-def B
and wC: well-def C and wD: well-def D
and p3p2: \( \bigwedge \)Q. unitary Q \( \Rightarrow \) «I» && wp B Q = wp B («H» && Q)
and p1p3: \( \bigwedge \)s. G s \( \Rightarrow \) I s

**shows** pcorres \( \varphi \) G (A;;C) (B;;D)

\[ \langle proof \rangle \]

4.9.8 Structural Rules for Data Refinement

**lemma** dr-Skip:

**fixes** \( \varphi ::'c \Rightarrow 'b \)

**shows** drefines \( \varphi \) G Skip Skip

\[ \langle proof \rangle \]

**lemma** dr-Abort:

**fixes** \( \varphi ::'c \Rightarrow 'b \)

**shows** drefines \( \varphi \) G Abort Abort

\[ \langle proof \rangle \]

**lemma** dr-Apply:

**fixes** \( \varphi ::'c \Rightarrow 'b \)

**assumes** commutes: \( f \circ \varphi = \varphi \circ g \)

**shows** drefines \( \varphi \) G (Apply f) (Apply g)
lemma \textit{dr-Seq}:
\begin{align*}
&\text{assumes } \textit{drAB}: \textit{drefines } \varphi \ P \ A \ B \\
&\text{and } \textit{drBC}: \textit{drefines } \varphi \ Q \ C \ D \\
&\text{and } \textit{wpB}: \ «P» \vdash \textit{wp } B \ «Q» \\
&\text{and } \textit{wB}: \ \text{well-def } B \\
&\text{and } \textit{wC}: \ \text{well-def } C \\
&\text{and } \textit{wD}: \ \text{well-def } D \\
\end{align*}
\text{shows } \textit{drefines } \varphi \ P \ (A;; C) \ (B;; D)
\langle \textit{proof} \rangle
\end{lemma}

lemma \textit{dr-repeat}:
\begin{align*}
&\text{fixes } \varphi :: \ 'a \Rightarrow \ 'b \\
&\text{assumes } \textit{dr-ab}: \textit{drefines } \varphi \ G \ a \ b \\
&\text{and } \textit{Gpr}: \ «G» \vdash \textit{wp } b \ «G» \\
&\text{and } \textit{wa}: \ \text{well-def } a \\
&\text{and } \textit{wb}: \ \text{well-def } b \\
\end{align*}
\text{shows } \textit{drefines } \varphi \ G \ (\text{repeat } n \ a) \ (\text{repeat } n \ b) \ (\text{is } ?X \ n)
\langle \textit{proof} \rangle
\end{lemma}

4.10 Structured Reasoning

\texttt{theory StructuredReasoning imports Algebra begin}

By linking the algebraic, the syntactic, and the semantic views of computation, we derive a set of rules for decomposing expectation entailment proofs, firstly over the syntactic structure of a program, and secondly over the refinement relation. These rules also form the basis for automated reasoning.

4.10.1 Syntactic Decomposition

lemma \textit{wp-Abort}:
\begin{align*}
&\ (\lambda s. \theta) \vdash \textit{wp } Abort \ Q \\
\end{align*}
\langle \textit{proof} \rangle

lemma \textit{wlp-Abort}:
\begin{align*}
&\ (\lambda s. \beta) \vdash \textit{wlp } Abort \ Q \\
\end{align*}
\langle \textit{proof} \rangle

lemma \textit{wp-Skip}:
\begin{align*}
&P \vdash \textit{wp } Skip \ P \\
\end{align*}
\langle \textit{proof} \rangle

lemma \textit{wlp-Skip}:
\begin{align*}
&P \vdash \textit{wlp } Skip \ P \\
\end{align*}
4.10. STRUCTURED REASONING

(⟨proof⟩)

**Lemma** *wp-Apply*:
\( Q \circ f \vdash wp (\text{Apply } f) \; Q \)
(⟨proof⟩)

**Lemma** *wlp-Apply*:
\( Q \circ f \vdash wlp (\text{Apply } f) \; Q \)
(⟨proof⟩)

**Lemma** *wp-Seq*:
assumes
\( \text{ent-a: } P \vdash wp a \; Q \)
and
\( \text{ent-b: } Q \vdash wp b \; R \)
and \( wa: \; \text{well-def } a \)
and \( wb: \; \text{well-def } b \)
and \( s-Q: \; \text{sound } Q \)
and \( s-R: \; \text{sound } R \)
shows
\( P \vdash wp (a \;;\; b) \; R \)
(⟨proof⟩)

**Lemma** *wlp-Seq*:
assumes
\( \text{ent-a: } P \vdash wlp a \; Q \)
and
\( \text{ent-b: } Q \vdash wlp b \; R \)
and \( wa: \; \text{well-def } a \)
and \( wb: \; \text{well-def } b \)
and \( u-Q: \; \text{unitary } Q \)
and \( u-R: \; \text{unitary } R \)
shows
\( P \vdash wlp (a \;;\; b) \; R \)
(⟨proof⟩)

**Lemma** *wp-PC*:
\( (\lambda s. \; P \; s \; * \; wp \; a \; Q \; s \; + \; (1 \; - \; P \; s) \; * \; wp \; b \; Q \; s) \; \vdash \; wp (a \; p \oplus \; b) \; Q \)
(⟨proof⟩)

**Lemma** *wlp-PC*:
\( (\lambda s. \; P \; s \; * \; wlp \; a \; Q \; s \; + \; (1 \; - \; P \; s) \; * \; wlp \; b \; Q \; s) \; \vdash \; wlp (a \; p \oplus \; b) \; Q \)
(⟨proof⟩)

A simpler rule for when the probability does not depend on the state.

**Lemma** *PC-fixed*:
assumes
\( wp: \; P \; \vdash \; a \; ab \; R \)
and \( wpb: \; Q \; \vdash \; b \; ab \; R \)
and \( np: \; 0 \leq \; p \; \text{ and } \; bp: \; p \leq \; 1 \)
shows
\( (\lambda s. \; p \; * \; P \; s \; + \; (1 \; - \; p) \; * \; Q \; s) \; \vdash \; (a \; (\lambda s. \; p) \oplus \; b) \; ab \; R \)
(⟨proof⟩)

**Lemma** *wp-PC-fixed*:
\[
\left[ \begin{array}{c} P \vdash wp a \; R; \; Q \vdash wp b \; R; \; 0 \leq p; \; p \leq 1 \end{array} \right] \implies \\
(\lambda s. \; p \; * \; P \; s \; + \; (1 \; - \; p) \; * \; Q \; s) \; \vdash \; wp (a \; (\lambda s. \; p) \oplus \; b) \; R
\]

A simpler rule for when the probability does not depend on the state.
(proof)

**Lemma wlp-PC-fixed:**

\[
\begin{array}{c}
P \vdash wlp \ a \ R; \ Q \vdash wlp \ b \ R; \ 0 \leq p; \ p \leq 1 \ \Rightarrow \\
(\lambda s. p \cdot P s + (1 - p) \cdot Q s) \vdash wlp \ (a \ (\lambda s. p) \oplus b) \ R
\end{array}
\]

(\proof)

**Lemma wp-DC:**

\[
\begin{array}{c}
(\lambda s. \min (wp \ a \ Q s) (wp \ b \ Q s)) \vdash wp \ (a \ \sqcap \ b) \ Q
\end{array}
\]

(\proof)

**Lemma wlp-DC:**

\[
\begin{array}{c}
(\lambda s. \min (wlp \ a \ Q s) (wlp \ b \ Q s)) \vdash wlp \ (a \ \sqcap \ b) \ Q
\end{array}
\]

(\proof)

Combining annotations for both branches:

**Lemma DC-split:**

\[
\begin{array}{c}
\text{fixes } a::'s \ prog \ \text{and } b \\
\text{assumes } wpa: \ P \vdash a \ ab \ R \\
\text{and } wpb: \ Q \vdash b \ ab \ R \\
\text{shows } (\lambda s. \min (P s) (Q s)) \vdash (a \ \sqcap \ b) \ ab \ R
\end{array}
\]

(\proof)

**Lemma wp-DC-split:**

\[
\begin{array}{c}
[\ P \vdash wp \ prog \ R; \ Q \vdash wp \ prog' \ R \ ] \Rightarrow \\
(\lambda s. \min (P s) (Q s)) \vdash wp \ (prog \ \sqcap \ prog') \ R
\end{array}
\]

(\proof)

**Lemma wlp-DC-split:**

\[
\begin{array}{c}
[\ P \vdash wlp \ prog \ R; \ Q \vdash wlp \ prog' \ R \ ] \Rightarrow \\
(\lambda s. \min (P s) (Q s)) \vdash wlp \ (prog \ \sqcap \ prog') \ R
\end{array}
\]

(\proof)

**Lemma wp-DC-split-same:**

\[
\begin{array}{c}
[\ P \vdash wp \ prog \ Q; \ P \vdash wp \ prog' \ Q \ ] \Rightarrow P \vdash wp \ (prog \ \sqcap \ prog') \ Q
\end{array}
\]

(\proof)

**Lemma wlp-DC-split-same:**

\[
\begin{array}{c}
[\ P \vdash wlp \ prog \ Q; \ P \vdash wlp \ prog' \ Q \ ] \Rightarrow P \vdash wlp \ (prog \ \sqcap \ prog') \ Q
\end{array}
\]

(\proof)

**Lemma SetPC-split:**

\[
\begin{array}{c}
\text{fixes } f::'x \Rightarrow 'y \ prog \\
\text{and } p::'y \Rightarrow 'x \Rightarrow 'real \\
\text{assumes } \text{rec: } \bigwedge x s. \ x \in \supp (p s) \Rightarrow P x \vdash f \ x \ ab \ Q \\
\text{and } nnp: \bigwedge s. \ nneg (p s) \\
\text{shows } (\lambda s. \sum x \in \supp (p s). \ p s x \cdot P x s) \vdash SetPC \ f \ p \ ab \ Q
\end{array}
\]

(\proof)
lemma \( \text{wp-SetPC-split} \):
\[
\left[ \forall x \in \text{supp} (p \cdot s) \Rightarrow x \vdash \text{wp} \left( f \cdot x \right) \cdot Q ; \forall s. \text{neg} (p \cdot s) \right] \Rightarrow \\
(\lambda s. \sum x \in \text{supp} (p \cdot s). p \cdot s \cdot x \vdash \text{wp} \left( \text{SetPC} \cdot f \right) \cdot Q)
\]
\( \langle \text{proof} \rangle \)

lemma \( \text{wlp-SetPC-split} \):
\[
\left[ \forall x \in \text{supp} (p \cdot s) \Rightarrow x \vdash \text{wlp} \left( f \cdot x \right) \cdot Q ; \forall s. \text{neg} (p \cdot s) \right] \Rightarrow \\
(\lambda s. \sum x \in \text{supp} (p \cdot s). p \cdot s \cdot x \vdash \text{wlp} \left( \text{SetPC} \cdot f \right) \cdot Q)
\]
\( \langle \text{proof} \rangle \)

lemma \( \text{wp-SetDC-split} \):
\[
\left[ \forall s \cdot x \in S \Rightarrow x \vdash \text{wp} \left( f \cdot x \right) \cdot Q ; \forall s. S \cdot s \neq \emptyset \right] \Rightarrow \\
P \vdash \text{wp} \left( \text{SetDC} \cdot f \cdot S \right) \cdot Q
\]
\( \langle \text{proof} \rangle \)

lemma \( \text{wlp-SetDC-split} \):
\[
\left[ \forall s \cdot x \in S \Rightarrow x \vdash \text{wlp} \left( f \cdot x \right) \cdot Q ; \forall s. S \cdot s \neq \emptyset \right] \Rightarrow \\
P \vdash \text{wlp} \left( \text{SetDC} \cdot f \cdot S \right) \cdot Q
\]
\( \langle \text{proof} \rangle \)

lemma \( \text{wp-SetDC} \):
assumes \( \text{wp} \): \( \forall s \cdot x \in S \Rightarrow x \vdash \text{wp} \left( f \cdot x \right) \cdot Q \)
and \( \text{ne} \): \( \forall s. S \cdot s \neq \emptyset \)
and \( \text{sP} \): \( \forall x. \text{sound} \left( P \cdot x \right) \)
shows \( (\lambda s. \text{Inf} \left( \left( \lambda x. P \cdot x \right) \cdot ^{\cdot} S \cdot s \right)) \vdash \text{wp} \left( \text{SetDC} \cdot f \cdot S \right) \cdot Q \)
\( \langle \text{proof} \rangle \)

lemma \( \text{wlp-SetDC} \):
assumes \( \text{wp} \): \( \forall s \cdot x \in S \Rightarrow x \vdash \text{wlp} \left( f \cdot x \right) \cdot Q \)
and \( \text{ne} \): \( \forall s. S \cdot s \neq \emptyset \)
and \( \text{sP} \): \( \forall x. \text{sound} \left( P \cdot x \right) \)
shows \( (\lambda s. \text{Inf} \left( \left( \lambda x. P \cdot x \right) \cdot ^{\cdot} S \cdot s \right)) \vdash \text{wlp} \left( \text{SetDC} \cdot f \cdot S \right) \cdot Q \)
\( \langle \text{proof} \rangle \)

lemma \( \text{wp-Embed} \):
\[
P \vdash t \cdot Q \Rightarrow P \vdash \text{wp} \left( \text{Embed} \cdot t \right) \cdot Q
\]
\( \langle \text{proof} \rangle \)

lemma \( \text{wlp-Embed} \):
\[
P \vdash t \cdot Q \Rightarrow P \vdash \text{wlp} \left( \text{Embed} \cdot t \right) \cdot Q
\]
\( \langle \text{proof} \rangle \)

lemma \( \text{wp-Bind} \):
\[
\left[ \forall s. P \cdot s \leq \text{wp} \left( a \cdot f \cdot s \right) \cdot Q \cdot s \right] \Rightarrow P \vdash \text{wp} \left( \text{Bind} \cdot f \cdot a \right) \cdot Q
\]
\( \langle \text{proof} \rangle \)

lemma \( \text{wlp-Bind} \):
\[
\left[ \forall s. P \cdot s \leq \text{wlp} \left( a \cdot f \cdot s \right) \cdot Q \cdot s \right] \Rightarrow P \vdash \text{wlp} \left( \text{Bind} \cdot f \cdot a \right) \cdot Q
\]
\( \langle \text{proof} \rangle \)
lemma \( wp\text{-}repeat \):

\[
\begin{align*}
& [ \ P \vdash wp \ a \ Q; \ Q \vdash wp \ (\text{repeat } n \ a) \ R; \\
& \text{well-def } a; \text{ sound } Q; \text{ sound } R \implies P \vdash wp \ (\text{repeat } (\text{Suc } n) \ a) \ R \\
\langle \text{proof} \rangle
\end{align*}
\]

lemma \( wlp\text{-}repeat \):

\[
\begin{align*}
& [ \ P \vdash wlp \ a \ Q; \ Q \vdash wlp \ (\text{repeat } n \ a) \ R; \\
& \text{well-def } a; \text{ unitary } Q; \text{ unitary } R \implies P \vdash wlp \ (\text{repeat } (\text{Suc } n) \ a) \ R \\
\langle \text{proof} \rangle
\end{align*}
\]

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 \( wp\text{-}strengthen\text{-}post \):

\[
\text{entails\text{-}strengthen\text{-}post}[\text{where } t=wp \ a \ for \ a]
\]

lemma \( wlp\text{-}strengthen\text{-}post \):

\[
\begin{align*}
P & \vdash wlp \ a \ Q \implies \text{nearby-healthy (wlp } a) \implies \text{unitary } R \implies Q \vdash R \implies \text{unitary } Q \\
P & \vdash wlp \ a \ R \\
\langle \text{proof} \rangle
\end{align*}
\]

lemmas \( wp\text{-}weaken\text{-}pre \):

\[
\text{entails\text{-}weaken\text{-}pre}[\text{where } t=wp \ a \ for \ a]
\]

lemmas \( wlp\text{-}weaken\text{-}pre \):

\[
\text{entails\text{-}weaken\text{-}pre}[\text{where } t=wlp \ a \ for \ a]
\]

lemmas \( wp\text{-}scale \):

\[
\text{entails\text{-}scale}[\text{where } t=wp \ a \ for \ a, \ OF - \text{well-def\text{-}wp\text{-}healthy}]
\]

4.10.2 Algebraic Decomposition

Refinement is a powerful tool for decomposition, belied by the simplicity of the rule. This is an axiomatic formulation of refinement (all annotations of the \( a \) are annotations of \( b \), rather than an operational version (all traces of \( b \) are traces of \( a \).

lemma \( wp\text{-}refines \):

\[
\begin{align*}
& [ \ a \sqsubseteq b; \ P \vdash wp \ a \ Q; \text{ sound } Q \implies P \vdash wp \ b \ Q \\
\langle \text{proof} \rangle
\end{align*}
\]

lemmas \( wp\text{-}drefines = drefinesD \)
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.

\[ \text{definition} \]
\[ wp\text{-}valid :: (\text{'a} \Rightarrow \text{real}) \Rightarrow \text{'a} \text{ prog} \Rightarrow (\text{'a} \Rightarrow \text{real}) \Rightarrow \text{bool} \ (\{\cdot\} - \{\cdot\}p) \]

\[ \text{where} \]
\[ wp\text{-}valid \ P \ \text{prog} \ Q \equiv P \vdash wp \ \text{prog} \ \text{Q} \]

\[ \text{lemma wp\text{-}validI:} \]
\[ P \vdash wp \ \text{prog} \ Q \rightarrow \{P\} \ \text{prog} \ \{Q\}p \]

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

\[ \text{lemma wp\text{-}validD:} \]
\[ \{P\} \ \text{prog} \ \{Q\}p \rightarrow P \vdash wp \ \text{prog} \ \text{Q} \]

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

\[ \text{lemma valid\text{-}Seq:} \]
\[ \left[ \{P\} \ a \ \{Q\}p; \ \{Q\} \ b \ \{R\}p; \ \text{well-def} \ a; \ \text{well-def} \ b; \ \text{sound} \ Q; \ \text{sound} \ R \right] \rightarrow \]
\[ \{P\} \ a ;; \ b \ \{R\}p \]

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

We make it available to the computational reasoner:

\[ \text{declare valid\text{-}Seq[trans]} \]

\[ \end \]

4.11 Loop Termination

\[ \text{theory Termination imports Embedding StructuredReasoning Loops begin} \]

Termination for loops can be shown by classical means (using a variant, or a measure function), or by probabilistic means: We only need that the loop terminates \emph{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).

\[ \text{lemma maximal-Seq-term:} \]
\[ \text{fixes r::'s prog and s::'s prog} \]
\[ \text{assumes mr: maximal (wp r)} \]
\[ \text{and ws: well-def s} \]
\[ \text{and ts: (\lambda s. \ 1) \vdash wp \ s (\lambda s. \ 1)} \]
\[ \text{shows (\lambda s. \ 1) \vdash wp \ (r ;; s) (\lambda s. \ 1)} \]

\[ \langle \text{proof} \rangle \]
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** «N G» ⊢ wp G → body od (λs. 1)

(⟨proof⟩)

### 4.11.2 Classical Termination

The first non-trivial termination result is quite standard: If we can provide a natural-number-valued measure, that decreases on every iteration, and implies termination on reaching zero, the loop terminates.

**lemma loop-term-nat-measure-noinv:**

- **fixes** m :: ′s ⇒ nat and body :: ′s prog
- **assumes** wb: well-def body
- **and** guard: ∀s. m s = 0 → ¬ G s
- **and** variant: ∀n. «λs. m s = Suc n» ⊨ wp body «λs. m s = n»
- **shows** λs. 1 ⊨ wp G → body od (λs. 1)

(⟨proof⟩)

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 ⇒ nat and body :: ′s prog
- **assumes** wb: well-def body
- **and** guard: ∀s. m s = 0 → ¬ G s
- **and** variant: ∀n. «λs. m s = Suc n» &«I» ⊨ wp body «λs. m s = n»
- **and** inv: wp-inv G body «I»
- **shows** «I» ⊨ wp G → body od (λs. 1)

(⟨proof⟩)

### 4.11.3 Probabilistic Termination

Any loop that has a non-zero chance of terminating after each step terminates with probability 1.

**lemma termination-0-1:**

- **fixes** body :: ′s prog
- **assumes** wb: well-def body
  - The loop terminates in one step with nonzero probability
  - **and** onestep: (λs. p) ⊨ wp body «N G»
  - **and** nzp: 0 < p
  - The body is maximal i.e. it terminates absolutely.
  - **and** mb: maximal (wp body)
- **shows** λs. 1 ⊨ wp G → body od (λs. 1)

(⟨proof⟩)

end
4.12 Automated Reasoning

theory Automation imports StructuredReasoning 
begin

This theory serves as a container for automated reasoning tactics for pGCL, implemented in ML. At present, there is a basic verification condition generator (VCG).

named-theorems wd
theorems to automatically establish well-definedness

named-theorems pwp-core
core probabilistic wp rules, for evaluating primitive terms

named-theorems pwp
user-supplied probabilistic wp rules

named-theorems 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
Additional Material

4.13 Miscellaneous Mathematics

theory Misc
imports
  HOL-Analysis.Analysis
begin

  lemma sum-UNIV:
  fixes S::'a::finite set
  assumes complete: \( \forall x. x \notin S \Rightarrow f x = 0 \)
  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 \( c \ast \max a b = \max (c \ast a) (c \ast b) \)
  ⟨proof⟩

  lemma mult-div-mono-left:
  fixes c::real
  assumes nnc: \( 0 \leq c \) and nzc: \( c \neq 0 \)
  and inv: \( a \leq \text{inverse} c \ast b \)
  shows \( c \ast a \leq b \)
  ⟨proof⟩

  lemma mult-div-mono-right:
  fixes c::real
  assumes nnc: \( 0 \leq c \) and nzc: \( c \neq 0 \)
  and inv: \( \text{inverse} c \ast a \leq b \)
  shows \( a \leq c \ast b \)
  ⟨proof⟩

⟨proof⟩
lemma min-distrib:
  fixes c::real
  assumes nnc: 0 ≤ c
  shows c * min a b = min (c * a) (c * b)
⟨proof⟩

lemma finite-set-least:
  fixes S::'a:linorder set
  assumes finite: finite S
      and ne: S ≠ {}
  shows ∃x∈S. ∀y∈S. x ≤ y
⟨proof⟩

lemma cSup-add:
  fixes c::real
  assumes ne: S ≠ {}
      and bS: ∀x. x∈S ⇒ x ≤ b
  shows Sup S + c = Sup {x + c | x. x ∈ S}
⟨proof⟩

lemma cSup-mult:
  fixes c::real
  assumes ne: S ≠ {}
      and bS: ∀x. x∈S ⇒ x ≤ b
      and nnc: 0 ≤ c
  shows c * Sup S = Sup {c * x | x. x ∈ S}
⟨proof⟩

lemma closure-contains-Sup:
  fixes S :: real set
  assumes neS: S ≠ {} and bS: ∀x∈S. x ≤ B
  shows Sup S ∈ closure S
⟨proof⟩

lemma tendsto-min:
  fixes x y::real
  assumes ta: a −−−−→ x
      and tb: b −−−−→ y
  shows (λi. min (a i) (b i)) −−−−→ min x y
⟨proof⟩

definition supp :: ('s ⇒ real) ⇒ 's set
where supp f = {x. f x ≠ 0}

definition dist-remove :: ('s ⇒ real) ⇒ 's ⇒ 's ⇒ real
where dist-remove p x = (λy. if y=x then 0 else p y / (1 − p x))

lemma supp-dist-remove:
\[ p \neq 0 \implies p \neq 1 \implies \text{supp} \left( \text{dist-remove} \ p \ x \right) = \text{supp} \ p - \{x\} \]

**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**
\[ \text{tminus} :: \text{real} \Rightarrow \text{real} \Rightarrow \text{real} \\ (\text{infixl} \ \ominus) \]

**Where**
\[ x \ominus y = \max (x - y) \ 0 \]

**Lemma** minus-le-tminus\[\text{intro,simp}]:
\[ a - b \leq a \ominus b \]

**Proof**

**Lemma** tminus-cancel-1:
\[ \theta \leq a \implies a + 1 \ominus 1 = a \]

**Proof**

**Lemma** tminus-zero-imp-le:
\[ x \ominus y \leq 0 \implies x \leq y \]

**Proof**

**Lemma** tminus-zero\[\text{simp}]:
\[ \theta \leq x \implies x \ominus 0 = x \]

**Proof**

**Lemma** tminus-left-mono:
\[ a \leq b \implies a \ominus c \leq b \ominus c \]

**Proof**

**Lemma** tminus-less:
\[ [ \ 0 \leq a; \ 0 \leq b \ ] \implies a \ominus b \leq a \]

**Proof**

**Lemma** tminus-left-distrib:
\[ \text{assumes} \ nna: \ 0 \leq a \]
shows $a \ast (b \ominus c) = a \ast b \ominus a \ast c$

⟨proof⟩

lemma tminus-le[simp]:
\[ b \leq a \implies a \ominus b = a - b \]
⟨proof⟩

lemma tminus-le-alt[simp]:
\[ a \leq b \implies a \ominus b = 0 \]
⟨proof⟩

lemma tminus-nle[simp]:
\[ \neg b \leq a \implies a \ominus b = 0 \]
⟨proof⟩

lemma tminus-add-mono:
\[(a+b) \ominus (c+d) \leq (a\ominus c) + (b\ominus d)\]
⟨proof⟩

lemma tminus-sum-mono:
assumes $fS$: finite S
shows $\text{sum } f S \ominus \text{sum } g S \leq \text{sum } (\lambda x. f x \ominus g x) S$

(is ?X S)
⟨proof⟩

lemma tminus-nneg[simp,intro]:
\[ \emptyset \leq a \ominus b \]
⟨proof⟩

lemma tminus-right-antimono:
assumes clb: $c \leq b$
shows $a \ominus b \leq a \ominus c$
⟨proof⟩

lemma min-tminus-distrib:
\[ \text{min } a \ b \ominus c = \text{min } (a \ominus c) \ (b \ominus c) \]
⟨proof⟩

end
Bibliography


