Contents

1 Overview 1

2 Introduction to pGCL 3
   2.1 Language Primitives ................................. 3
      2.1.1 The Basics .................................... 3
      2.1.2 Assertion and Annotation ...................... 4
      2.1.3 Probability .................................... 4
      2.1.4 Nondeterminism .................................. 5
      2.1.5 Properties of Expectations ..................... 5
   2.2 Loops .................................................. 6
      2.2.1 Guaranteed Termination ......................... 6
      2.2.2 Probabilistic Termination ....................... 7
   2.3 The Monty Hall Problem ............................... 8
      2.3.1 The State Space ................................ 8
      2.3.2 The Game ........................................ 9
      2.3.3 A Brute Force Solution ......................... 9
      2.3.4 A Modular Approach ............................ 10

3 Semantic Structures 13
   3.1 Expectations .......................................... 13
      3.1.1 Bounded Functions ............................... 14
      3.1.2 Non-Negative Functions .......................... 16
      3.1.3 Sound Expectations .............................. 17
      3.1.4 Unitary expectations ............................. 19
      3.1.5 Standard Expectations ........................... 19
      3.1.6 Entailment ...................................... 22
      3.1.7 Expectation Conjunction ......................... 22
      3.1.8 Rules Involving Conjunction .................... 24
      3.1.9 Rules Involving Entailment and Conjunction Together 25
   3.2 Expectation Transformers ............................... 26
      3.2.1 Comparing Transformers .......................... 29
      3.2.2 Healthy Transformers ............................ 32
      3.2.3 Sublinearity ................................... 37
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.4</td>
<td>Determinism</td>
<td>42</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Modular Reasoning</td>
<td>44</td>
</tr>
<tr>
<td>3.2.6</td>
<td>Transforming Standard Expectations</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>Induction</td>
<td>46</td>
</tr>
<tr>
<td>3.3.1</td>
<td>The Lattice of Expectations</td>
<td>46</td>
</tr>
<tr>
<td>3.3.2</td>
<td>The Lattice of Transformers</td>
<td>49</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Tail Recursion</td>
<td>52</td>
</tr>
<tr>
<td>4</td>
<td>The pGCL Language</td>
<td>55</td>
</tr>
<tr>
<td>4.1</td>
<td>A Shallow Embedding of pGCL in HOL</td>
<td>55</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Core Primitives and Syntax</td>
<td>55</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Unfolding rules for non-recursive primitives</td>
<td>58</td>
</tr>
<tr>
<td>4.2</td>
<td>Healthiness</td>
<td>60</td>
</tr>
<tr>
<td>4.2.1</td>
<td>The Healthiness of the Embedding</td>
<td>61</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Healthiness for Loops</td>
<td>63</td>
</tr>
<tr>
<td>4.3</td>
<td>Continuity</td>
<td>66</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Continuity of Primitives</td>
<td>67</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Continuity of a Single Loop Step</td>
<td>70</td>
</tr>
<tr>
<td>4.4</td>
<td>Continuity and Induction for Loops</td>
<td>70</td>
</tr>
<tr>
<td>4.4.1</td>
<td>The Limit of Iterates</td>
<td>71</td>
</tr>
<tr>
<td>4.5</td>
<td>Sublinearity</td>
<td>73</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Nonrecursive Primitives</td>
<td>73</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Sublinearity for Loops</td>
<td>75</td>
</tr>
<tr>
<td>4.6</td>
<td>Determinism</td>
<td>76</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Additivity</td>
<td>76</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Maximality</td>
<td>77</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Determinism</td>
<td>78</td>
</tr>
<tr>
<td>4.7</td>
<td>Well-Defined Programs.</td>
<td>79</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Strict Implies Liberal</td>
<td>80</td>
</tr>
<tr>
<td>4.7.2</td>
<td>Sub-Distributivity of Conjunction</td>
<td>82</td>
</tr>
<tr>
<td>4.7.3</td>
<td>The Well-Defined Predicate</td>
<td>84</td>
</tr>
<tr>
<td>4.8</td>
<td>The Loop Rules</td>
<td>86</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Liberal and Strict Invariants</td>
<td>86</td>
</tr>
<tr>
<td>4.8.2</td>
<td>Partial Correctness</td>
<td>87</td>
</tr>
<tr>
<td>4.8.3</td>
<td>Total Correctness</td>
<td>87</td>
</tr>
<tr>
<td>4.8.4</td>
<td>Unfolding</td>
<td>88</td>
</tr>
<tr>
<td>4.9</td>
<td>The Algebra of pGCL</td>
<td>88</td>
</tr>
<tr>
<td>4.9.1</td>
<td>Program Refinement</td>
<td>88</td>
</tr>
<tr>
<td>4.9.2</td>
<td>Simple Identities</td>
<td>89</td>
</tr>
<tr>
<td>4.9.3</td>
<td>Deterministic Programs are Maximal</td>
<td>92</td>
</tr>
<tr>
<td>4.9.4</td>
<td>The Algebraic Structure of Refinement</td>
<td>93</td>
</tr>
<tr>
<td>4.9.5</td>
<td>Data Refinement</td>
<td>94</td>
</tr>
<tr>
<td>4.9.6</td>
<td>The Algebra of Data Refinement</td>
<td>96</td>
</tr>
<tr>
<td>4.9.7</td>
<td>Structural Rules for Correspondence</td>
<td>97</td>
</tr>
</tbody>
</table>
4.9.8 Structural Rules for Data Refinement ............... 97
4.10 Structured Reasoning ........................................ 98
  4.10.1 Syntactic Decomposition ............................... 98
  4.10.2 Algebraic Decomposition ............................... 102
  4.10.3 Hoare triples ........................................... 103
4.11 Loop Termination ............................................. 103
  4.11.1 Trivial Termination ..................................... 103
  4.11.2 Classical Termination .................................. 104
  4.11.3 Probabilistic Termination ............................. 104
4.12 Automated Reasoning ....................................... 105

Additional Material .............................................. 107
4.13 Miscellaneous Mathematics ................................. 107
  4.13.1 Truncated Subtraction ................................. 109
Chapter 1

Overview

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

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

This formalisation was first presented (as an overview) in Cock [2012]. The language has previously been formalised in HOL4 by Hurd et al. [2005]. Two substantial results using this package were presented in Cock [2013], Cock [2014a] and Cock [2014b].
CHAPTER 1. OVERVIEW
Chapter 2

Introduction to pGCL

2.1 Language Primitives

theory Primitives imports ..:/pGCL begin

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

2.1.1 The Basics

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

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

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

```plaintext
lemma
  \text{Apply} \ (\lambda s. \ s(\| a := \text{Heads }\|)) = (a := (\lambda s. \text{Heads}))
\langle \text{proof} \rangle
```

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

2.1.2 Assertion and Annotation

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

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

\begin{verbatim}
lemma \forall x. « \lambda s. b s = x » $\Vdash \text{wp} (a := b) \ « \lambda s. a s = x »$
\end{verbatim}

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{verbatim}
lemma « \lambda s::real. s \neq 0 » $\Vdash \text{wp} (\text{Apply} ((\ast) 2) ;; \text{Apply} (\lambda s. s / s)) \ « \lambda s. s = 1 »$
\end{verbatim}

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

type \( 's \Rightarrow \text{real} \). The following annotation states that the probability of heads is exactly 1/2:

**definition**

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

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

**lemma**

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

**proof**

2.1.4 Nondeterminism

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

**lemma**

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

**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**

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

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

**lemma**

\[(\lambda s. 1/4) = \wp(\text{flip-a}(1/2) \sqcap \text{flip-b}(1/2)) \; « \lambda s. a = \text{Heads} \land b = \text{Heads} » \]

**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 =

\[ \text{red} :: \text{nat} \]
\[ \text{blue} :: \text{nat} \]

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

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

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

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

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

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

The expectations of independent variables add:

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

end

**2.2 Loops**

**theory** LoopExamples **imports** ../pGCL begin

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

**2.2.1 Guaranteed Termination**

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

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

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

**Definition** inv-count :: int \(\Rightarrow\) bool

where

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

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

**Lemma** wp-inv-count:

\[
\text{wp-inv} (\lambda x. \ 0 < x) ((\lambda s. s - 1)) \ « \text{inv-count}\)
\]

(proof)

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

**Lemma** term-countdown:

\[
« \text{inv-count}\uparrow wp\ \text{countdown} (\lambda s. 1)
\]

(proof)

### 2.2.2 Probabilistic Termination

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

**Type-Synonym** coin = bool

**Definition** Heads = True

**Definition** Tails = False

**Definition** flip :: coin prog

where

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

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

**Definition** wait-for-heads :: coin prog

where

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

Nonetheless, we can show termination.

**Lemma** wait-for-heads-term:

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

(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 than opens one of the remaining two doors, revealing a goat and showing that the prize is elsewhere. The contestent is then given the choice of switching their guess to the other unopened door, or sticking to their first guess.

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

2.3.1 The State Space

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

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

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

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

Invariants

We prove explicitly that only valid doors are ever chosen.

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 \( \cap \) hide-behind 2 \( \cap \) hide-behind 3

Guess uniformly at random.

definition make-guess :: game prog
where make-guess \( \equiv \) guess-behind 1 \( (\lambda s. 1/3)\) \( \oplus \)
\hspace{1cm} 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 \( \equiv \) \( \prod d \in (\lambda s. \{1,2,3\} - \{\text{prize } s, \text{guess } s\}) \). open-door \( d \)

Switch your guess to the other unopened door.

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

The complete game, either with or without switching guesses.

definition monty :: bool \( \Rightarrow \) game prog
where
\hspace{1cm} monty switch \( \equiv \) hide-prize \( \;;\)
\hspace{1.5cm} make-guess \( \;;\)
\hspace{1.5cm} reveal \( \;;\)
\hspace{1.5cm} (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-expected by unfolding.

lemma eval-win[simp]:
\( p = g \implies \langle \text{player-wins} \rangle (s[\text{prize} := p, \text{guess} := g, \text{clue} := c]) = 1 \)
chapter 2. introduction to pgcl

\begin{proof}
\end{proof}

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

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

\begin{lemma}
\texttt{wp-monty-noswitch}: \\
\ (\lambda s. \frac{1}{3}) = \texttt{wp} (\texttt{monty False}) \ \texttt{player-wins} \\
\end{lemma}

\begin{lemma}
\texttt{swap-upd}: \\
s|\ prize := p, clue := c, guess := g | = \\
s|\ prize := p, guess := g, clue := c | \\
\end{lemma}

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.

\begin{lemma}
\texttt{wp-monty-switch-brute-force}: \\
\ (\lambda s. \frac{2}{3}) = \texttt{wp} (\texttt{monty True}) \ \texttt{player-wins} \\
\end{lemma}

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.

\textbf{Healthiness}

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

\begin{lemma}
\texttt{wd-hide-prize}: \\
well-def \texttt{hide-prize} \\
\end{lemma}

\begin{lemma}
\texttt{wd-make-guess}: \\
well-def \texttt{make-guess} \\
\end{lemma}

\begin{lemma}
\texttt{wd-reveal}: \\
well-def \texttt{reveal} \\
\end{lemma}
2.3. THE MONTY HALL PROBLEM

**lemma** `wd-switch-guess`:

well-def switch-guess

(\textit{proof})

**lemmas** `monty-healthy` =

\begin{align*}
& \text{wd-switch-guess,}\ \text{wd-reveal,}\ \text{wd-make-guess,}\ \text{wd-hide-prize} \\
\end{align*}

**Annotations**

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

\textit{hide-prize} chooses a valid door.

**lemma** `wp-hide-prize`:

\begin{align*}
(\lambda s. 1) & \vdash wp \text{ hide-prize «inv-prize»} \\
(\textit{proof})
\end{align*}

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

**lemma** `wp-make-guess`:

\begin{align*}
(\lambda s. 2/3 * «\lambda g. \text{ inv-prize g» s}) & \vdash wp \text{ make-guess «\lambda g. guess g ≠ prize g ∧ inv-prize g ∧ inv-guess g»} \\
(\textit{proof})
\end{align*}

**lemma** `last-one`:

\begin{align*}
\text{assumes} & a \neq b \text{ and } a \in \{1::\text{nat},2,3\} \text{ and } b \in \{1,2,3\} \\
\text{shows} & \exists! c. \{1,2,3\} - \{b,a\} = \{c\} \\
(\textit{proof})
\end{align*}

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

**lemma** `wp-reveal`:

\begin{align*}
«\lambda g. \text{ guess g ≠ prize g ∧ inv-prize g ∧ inv-guess g»} & \vdash wp \text{ reveal «\lambda g. guess g ≠ prize g ∧} \\
& \text{clue g ≠ prize g ∧} \\
& \text{clue g ≠ guess g ∧} \\
& \text{inv-prize g ∧ inv-guess g ∧ inv-clue g»} \\
(is ?X \vdash wp \text{ reveal ?Y}) \\
(\textit{proof})
\end{align*}

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`:

\begin{align*}
& \{\text{guess g ≠ prize g; clue g ≠ prize g; clue g ≠ guess g;}
\text{inv-prize g; inv-guess g; inv-clue g} \} \implies \\
& \{1, 2, 3\} = \{\text{guess g, prize g, clue g}\} \\
(\textit{proof})
\end{align*}
CHAPTER 2. INTRODUCTION TO PGCL

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

**Lemma** `wp-switch-guess`:

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

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

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

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

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

**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`:

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

```
end
```
Chapter 3

Semantic Structures

3.1 Expectations

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

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

\[
\begin{array}{cccc}
  a & b & a \rightarrow b & x & y & x \leq y \\
  F & F & T & 0 & 0 & T \\
  F & T & T & 0 & 1 & T \\
  T & F & F & 1 & 0 & F \\
  T & T & T & 1 & 1 & T \\
\end{array}
\]

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

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

All expectations must be non-negative and bounded i.e. ∀s. 0 ≤ P s and
∃b.∀s.P s ≤ 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 ⇒ (′a ⇒ real) ⇒ bool

**where** bounded-by b P ≡ ∀x. P x ≤ 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 :: (′a ⇒ real) ⇒ bool

**where** bounded P ≡ (∃ b. bounded-by b P)

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

**definition** bound-of :: (′a ⇒ real) ⇒ real

**where** bound-of P ≡ Sup (P ' UNIV)

**lemma** bounded-bdd-above[ intro]:
\[ \text{assumes } bP; \text{ bounded } P \]

\[ \text{shows } \text{bdd-above } (\text{range } P) \]

**(proof)**

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

lemma bound-of-least[intro]:
    assumes bP: bounded-by b P
    shows bound-of P ≤ b
    ⟨proof⟩

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

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

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

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

This is occasionally easier to apply:

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

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

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

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

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

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

**Lemma bound-of-mono** [intro]:
\[
\begin{aligned}
P \leq Q; \text{bounded } P; \text{bounded } Q \Rightarrow \text{bound-of } P \leq \text{bound-of } Q
\end{aligned}
\]

**Lemma bounded-by-o** [intro, simp]:
\[
\begin{aligned}
\forall b. \text{bounded-by } b \ P \Rightarrow \text{bounded-by } b \ (P \circ f)
\end{aligned}
\]

**Lemma le-bound-of** [intro]:
\[
\begin{aligned}
\forall x. \text{bounded } f \Rightarrow f \ x \leq \text{bound-of } f
\end{aligned}
\]

### 3.1.2 Non-Negative Functions.

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

**Definition**
\[
nneg :: (\forall x. 0 \leq P \ x) \Rightarrow \text{bool}
\]

**Lemma nnegI** [intro]:
\[
\begin{aligned}
\forall x. 0 \leq P \ x \Rightarrow \text{nneg } P
\end{aligned}
\]

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

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

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

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

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

**Lemma nneg-o** [intro, simp]:
\[
\text{nneg } P \Rightarrow \text{nneg } (P \circ f)
\]
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 ⇒ real
  shows \[ \text{bounded-by } b \text{ } P; \text{nneg } P \implies 0 \leq b \]
  ⟨proof⟩

3.1.3 Sound Expectations

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

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

lemma soundI:
  \[ \text{bounded } P; \text{nneg } P \implies \text{sound } P \]
  ⟨proof⟩

lemma soundI2[intro]:
  \[ \text{bounded-by } b \text{ } P; \text{nneg } P \implies \text{sound } P \]
  ⟨proof⟩

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

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

lemma bound-of-sound[intro]:
  assumes sP: sound P
  shows 0 ≤ 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: sound P and sQ: sound Q
  shows sound \((\lambda s. P s + Q s)\)
  ⟨proof⟩
lemma mult-sound:
  assumes sP: sound P and sQ: sound Q
  shows sound (\lambda s. P s * Q s)
⟨proof⟩

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

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

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

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

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

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

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

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

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

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

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

lemma unit-mult[intro]:
  assumes sP: sound P and bP: bounded-by 1 P
  and sQ: sound Q and bQ: bounded-by 1 Q
  shows bounded-by 1 (λs. P s * Q s)
⟨proof⟩

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

3.1.4 Unitary expectations

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

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

lemma unitaryI[intro]:
  [ sound P; bounded-by 1 P ] ⇒ unitary P
⟨proof⟩

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

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

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

3.1.5 Standard Expectations

definition
**CHAPTER 3. SEMANTIC STRUCTURES**

```plaintext
embed-bool :: (′s ⇒ bool) ⇒ ′s ⇒ real (« - » 1000)

where
«P» ≡ (λs. if P s then 1 else 0)

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

**lemma** embed-bool-nneg[ simp, intro]:
  nneg «P»
  ⟨proof⟩

**lemma** embed-bool-bounded-by-1[ simp, intro]:
  bounded-by 1 «P»
  ⟨proof⟩

**lemma** embed-bool-bounded[ simp, intro]:
  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[ simp]:
  P s ⇒ «P» s = 1
  ⟨proof⟩

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

**lemma** embed-ge-0[ simp, intro]:
  0 ≤ «G» s
  ⟨proof⟩

**lemma** embed-le-1[ simp, intro]:
  «G» s ≤ 1
  ⟨proof⟩

**lemma** embed-le-1-alt[ simp, intro]:
  0 ≤ 1 − «G» s
  ⟨proof⟩

**lemma** expect-1-I:
  P x ⇒ 1 ≤ «P» x
  ⟨proof⟩
```
Negating a predicate has the expected effect in its embedding as an expectation:

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

**lemma** negateI: ¬ P s ⇒ N P s
⟨proof⟩

**lemma** embed-split:
 f s = «P» s * f s + «N P» s * f s
⟨proof⟩

**lemma** negate-embed:
 «N P» s = 1 − «P» s
⟨proof⟩

**lemma** eval-nembed-true[simp]:
 P s ⇒ «N P» s = 0
⟨proof⟩

**lemma** eval-nembed-false[simp]:
 ¬P s ⇒ «N P» s = 1
⟨proof⟩

**lemma** negate-Not[simp]:
 N Not = (λx. x)
⟨proof⟩

**lemma** negate-negate[simp]:
 N (N P) = P
⟨proof⟩

**lemma** embed-bool-cancel:
 «G» s * «N G» s = 0
⟨proof⟩
CHAPTER 3. SEMANTIC STRUCTURES

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]:
\[ [\forall s. P s ≤ Q s] ⇒ P ⊢ Q \]
\langle proof \rangle

**lemma** entailsD[dest]:
\[ P ⊢ Q ⇒ P s ≤ Q s \]
\langle proof \rangle

**lemma** eq-entails[intro]:
\[ P = Q ⇒ P ⊢ Q \]
\langle proof \rangle

**lemma** entails-trans[trans]:
\[ [ P ⊢ Q; Q ⊢ R ] ⇒ P ⊢ R \]
\langle proof \rangle

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

**lemma** implies-entails:
\[ [ [\forall s. P s ⇒ Q s ] ⇒ «P» ⊢ «Q» \]
\langle proof \rangle

**lemma** entails-implies:
\[ [ [«P» ⊢ «Q»; P s ] ⇒ Q s \]
\langle proof \rangle

3.1.7 Expectation Conjunction

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

**where**
\[ p .& q ≡ p + q ⊔ 1 \]

**definition** exp-conj :: ('s ⇒ real) ⇒ ('s ⇒ real) ⇒ ('s ⇒ real) (infixl && 71)

**where**
\[ a && b ≡ λs. (a s .& b s) \]

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

**Lemma pconj-lzero[intro,simp]:**
\[ b \leq 1 \Rightarrow 0 \& b = 0 \]
⟨proof⟩

**Lemma pconj-rzero[intro,simp]:**
\[ b \leq 1 \Rightarrow b \& 0 = 0 \]
⟨proof⟩

**Lemma pconj-lone[intro,simp]:**
\[ 0 \leq b \Rightarrow 1 \& b = b \]
⟨proof⟩

**Lemma pconj-rone[intro,simp]:**
\[ 0 \leq b \Rightarrow b \& 1 = b \]
⟨proof⟩

**Lemma pconj-bconj:**
\[ «a» s \& «b» s = «λs. a s \& b s» s \]
⟨proof⟩

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

**Lemma pconj-assoc:**
\[ [ \theta \leq a; a \leq 1; \theta \leq b; b \leq 1; \theta \leq c; c \leq 1 ] \Rightarrow a \& (b \& c) = (a \& b) \& c \]
⟨proof⟩

**Lemma pconj-mono:**
\[ [ a \leq b; c \leq d ] \Rightarrow a \& c \leq b \& d \]
⟨proof⟩

**Lemma pconj-nneg[intro,simp]:**
\[ \theta \leq a \& b \]
⟨proof⟩

**Lemma min-pconj:**
\[ (\min a b) \& (\min c d) \leq \min (a \& c) (b \& d) \]
⟨proof⟩

**Lemma pconj-less-one[simp]:**
\[ a + b < 1 \Rightarrow a \& b = 0 \]
⟨proof⟩

**Lemma pconj-ge-one[simp]:**
\[ 1 \leq a + b \Rightarrow a \& b = a + b - 1 \]
⟨proof⟩
lemma \textit{pconj-idem[simp]}:
\[ «P» s . & . «P» s = «P» s \]
\langle proof \rangle

3.1.8 Rules Involving Conjunction.

lemma \textit{exp-conj-mono-left}:
\[ P \vdash Q \implies P \&\& R \vdash Q \&\& R \]
\langle proof \rangle

lemma \textit{exp-conj-mono-right}:
\[ Q \vdash R \implies P \&\& Q \vdash P \&\& R \]
\langle proof \rangle

lemma \textit{exp-conj-comm[ac-simps]}:
\[ a \&\& b = b \&\& a \]
\langle proof \rangle

lemma \textit{exp-conj-bounded-by[intro,simp]}:
\begin{align*}
& \text{assumes } bP: \text{ bounded-by 1 } P \\
& \text{ and } bQ: \text{ bounded-by 1 } Q \\
& \text{ shows } \text{ bounded-by 1 } (P \&\& Q)
\end{align*}
\langle proof \rangle

lemma \textit{exp-conj-o-distrib[simp]}:
\[ (P \&\& Q) o f = (P o f) \&\& (Q o f) \]
\langle proof \rangle

lemma \textit{exp-conj-assoc}:
\begin{align*}
& \text{assumes unitary } P \text{ and unitary } Q \text{ and unitary } R \\
& \text{ shows } P \&\& (Q \&\& R) = (P \&\& Q) \&\& R
\end{align*}
\langle proof \rangle

lemma \textit{exp-conj-top-left[simp]}:
\[ \text{sound } P \implies «\lambda. \ True» \&\& P = P \]
\langle proof \rangle

lemma \textit{exp-conj-top-right[simp]}:
\[ \text{sound } P \implies P \&\& «\lambda. \ True» = P \]
\langle proof \rangle

lemma \textit{exp-conj-idem[simp]}:
\[ «P» \&\& «P» = «P» \]
\langle proof \rangle

lemma \textit{exp-conj-nneg[intro,simp]}:
\[ (\lambda s. 0) \leq P \&\& Q \]
\langle proof \rangle
3.1. EXPECTATIONS

lemma exp-conj-sound[intro,simp]:
assumes s-P: sound P
    and s-Q: sound Q
shows sound (P && Q)
⟨proof⟩

lemma exp-conj-rzero[simp]:
bounded-by 1 P ⇒ P && (λs. 0) = (λs. 0)
⟨proof⟩

lemma exp-conj-1-right[simp]:
assumes nn: nneg A
shows A && (λ-. 1) = A
⟨proof⟩

lemma exp-conj-std-split:
«λs. P s ∧ Q s» = «P» && «Q»
⟨proof⟩

3.1.9 Rules Involving Entailment and Conjunction Together

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

lemma entails-frame:
assumes ePR: P ⊢ R
    and eQS: Q ⊢ S
shows P && Q ⊢ R && S
⟨proof⟩

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

lemma pentails-cases:
assumes PQe: ∀x. P x ⊢ Q x
    and exhaust: ∀s. ∃x. P (x s) s = 1
    and framed: ∀x. P x && R ⊢ Q x && 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 * 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 \(a \rightarrow a \rightarrow \ldots\) thus has probability 0, and the automaton

Figure 3.2: A nondeterministic-probabilistic automaton.
terminates with probability 1. We formalise such probabilistic termination arguments in Section 4.11.

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

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

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

3.2.1 Comparing Transformers

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

definition le-trans :: 's trans ⇒ 's trans ⇒ bool

where

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

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

definition le-utrans :: 's trans ⇒ 's trans ⇒ bool

where

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

lemma le-transI[intro]:

[ ∀P. sound P ⇒ t P ≤ u P ] ⇒ le-trans t u

(proof)

lemma le-utransI[intro]:

[ ∀P. unitary P ⇒ t P ≤ u P ] ⇒ le-utrans t u

(proof)

lemma le-transD[dest]:

[ le-trans t u; sound P ] ⇒ t P ≤ u P

(proof)

lemma le-utransD[dest]:

[ le-utrans t u; unitary P ] ⇒ t P ≤ u P

(proof)

lemma le-trans-trans[trans]:

[ le-trans x y; le-trans y z ] ⇒ le-trans x z

(proof)

lemma le-utrans-trans[trans]:

[ le-utrans x y; le-utrans y z ] ⇒ le-utrans x z

(proof)

lemma le-trans-refl[iff]:

le-trans x x

(proof)

lemma le-utrans-refl[iff]:

le-utrans x x

(proof)
lemma le-trans-le-utrans[dest]:
  le-trans t u ⇒ le-utrans t u
⟨proof⟩

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

Transformer equivalence is induced by comparison:

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

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

lemma equiv-transI[intro]:
  [ ∀P. sound P ⇒ t P = u P ] ⇒ equiv-trans t u
⟨proof⟩

lemma equiv-utransI[intro]:
  [ ∀P. sound P ⇒ t P = u P ] ⇒ equiv-utrans t u
⟨proof⟩

lemma equiv-transD[dest]:
  [ equiv-trans t u; sound P ] ⇒ t P = u P
⟨proof⟩

lemma equiv-utransD[dest]:
  [ equiv-utrans t u; unitary P ] ⇒ t P = u P
⟨proof⟩

lemma equiv-trans-refl[iff]:
  equiv-trans t t
⟨proof⟩

lemma equiv-utrans-refl[iff]:
  equiv-utrans t t
⟨proof⟩

lemma le-trans-antisym:
  [ le-trans x y; le-trans y x ] ⇒ equiv-trans x y
⟨proof⟩

lemma le-utrans-antisym:
3.2. EXPECTATION TRANSFORMERS

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

(proof)

**Lemma** `equiv-trans-comm [ac-simps]`:
\[ \text{equiv-trans } t \; u \iff \text{equiv-trans } u \; t \]

(proof)

**Lemma** `equiv-utrans-comm [ac-simps]`:
\[ \text{equiv-utrans } t \; u \iff \text{equiv-utrans } u \; t \]

(proof)

**Lemma** `equiv-imp-le [intro]`:
\[ \text{equiv-trans } t \; u \implies \text{le-trans } t \; u \]

(proof)

**Lemma** `equivu-imp-le [intro]`:
\[ \text{equiv-utrans } t \; u \implies \text{le-utrans } t \; u \]

(proof)

**Lemma** `equiv-imp-le-alt`:
\[ \text{equiv-trans } t \; u \implies \text{le-trans } u \; t \]

(proof)

**Lemma** `equiv-uimp-le-alt`:
\[ \text{equiv-utrans } t \; u \implies \text{le-utrans } u \; t \]

(proof)

**Lemma** `le-trans-equiv-rsp [simp]`:
\[ \text{equiv-trans } t \; u \implies \text{le-trans } u \; v \iff \text{le-trans } v \; u \]

(proof)

**Lemma** `le-utrans-equiv-rsp [simp]`:
\[ \text{equiv-utrans } t \; u \implies \text{le-utrans } u \; v \iff \text{le-utrans } v \; u \]

(proof)

**Lemma** `equiv-trans-le-trans [trans]`:
\[ \text{equiv-trans } t \; u \; \text{le-trans } u \; v \implies \text{le-trans } t \; v \]

(proof)

**Lemma** `equiv-utrans-le-utrans [trans]`:
\[ \text{equiv-utrans } t \; u \; \text{le-utrans } u \; v \implies \text{le-utrans } t \; v \]

(proof)

**Lemma** `le-trans-equiv-rsp-right [simp]`:
\[ \text{equiv-trans } t \; u \implies \text{le-trans } v \; t \iff \text{le-trans } v \; u \]

(proof)

**Lemma** `le-utrans-equiv-rsp-right [simp]`:
\[ \text{equiv-utrans } t \; u \implies \text{le-utrans } v \; t \iff \text{le-utrans } v \; u \]
CHAPTER 3. SEMANTIC STRUCTURES

⟨proof⟩

lemma le-trans-equiv-trans[trans]:
\[ le-trans t u; equiv-trans u v \implies le-trans t v \]
⟨proof⟩

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

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

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

lemma equiv-trans-equiv-utrans[dest]:
equiv-trans t u =⇒ equiv-utrans t u
⟨proof⟩

3.2.2 Healthy Transformers

Feasibility

definition feasible :: (('a ⇒ real) ⇒ ('a ⇒ real)) ⇒ bool
where feasible t ←→ (∀ P b. bounded-by b P ∧ nneg P =⇒ bounded-by b (t P) ∧ nneg (t P))

A feasible transformer preserves non-negativity, and bounds. A feasible transformer always takes its argument ‘closer to 0’ (or leaves it where it is). Note that any particular value of the expectation may increase, but no element of the new expectation may exceed any bound on the old. This is thus a relatively weak condition.

lemma feasibleI[intro]:
\[ \land b P. [ \land bounded-by b P; nneg P] =⇒ bounded-by b (t P); \land b P. [ bounded-by b P; nneg P] =⇒ nneg (t P) \implies feasible t \]
⟨proof⟩

lemma feasible-boundedD[dest]:
\[ feasible t; bounded-by b P; nneg P \implies bounded-by b (t P) \]
⟨proof⟩

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

\[
\begin{align*}
\text{lemma feasible-sound[dest]:} \\
&[\text{feasible } t; \text{ sound } P ] \implies \text{sound } (t P) \\
\langle \text{proof} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{lemma feasible-pr-0[simp]:} \\
&\text{fixes } t::('s \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real} \\
&\text{assumes } ft: \text{feasible } t \\
&\text{shows } t (\lambda x. 0) = (\lambda x. 0) \\
\langle \text{proof} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{lemma feasible-id:} \\
&\text{feasible } (\lambda x. x) \\
\langle \text{proof} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{lemma feasible-bounded-by[dest]:} \\
&[\text{feasible } t; \text{ sound } P; \text{ bounded-by } b P ] \implies \text{bounded-by } b (t P) \\
\langle \text{proof} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{lemma feasible-fixes-top:} \\
&\text{feasible } t \implies t (\lambda s. 1) \leq (\lambda s. (1::\text{real})) \\
\langle \text{proof} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{lemma feasible-fixes-bot:} \\
&\text{assumes } ft: \text{feasible } t \\
&\text{shows } t (\lambda s. 0) = (\lambda s. 0) \\
\langle \text{proof} \rangle
\end{align*}
\]

\[
\begin{align*}
\text{lemma feasible-unitaryD[dest]:} \\
&\text{assumes } ft: \text{feasible } t \text{ and } uP: \text{unitary } P \\
&\text{shows } \text{unitary } (t P) \\
\langle \text{proof} \rangle
\end{align*}
\]

**Monotonicity**

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

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 \( Q \models t R \) means that \( Q \) is everywhere below \( t R \). For standard expectations (Section 3.1.5), this simply means that \( Q \) implies \( t R \), the weakest precondition of \( R \) under \( t \). Given another, monotonic, transformer \( u \), we have that \( u Q \models u (t R) \), or that the weakest precondition of \( Q \) under \( u \) entails that of \( R \) under the
composition $u \circ t$. If we additionally know that $P \vdash u Q$, then by transitivity we have $P \vdash u (t R)$. We thus derive a probabilistic form of the standard rule for sequential composition: 

\[ \text{[mono-trans } t; P \vdash u Q; Q \vdash t R \] \implies P \vdash u (t R). \]

**Lemma** mono-transI[doc]:

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

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

**Definition** scaling:

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

\[ \text{where} \]

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

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[doc]:

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

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

**Lemma** scalingD[doc]:

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

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

**Lemma** right-scalingD:

Assumes:

- st: \text{scaling } t
- sP: \text{sound } P
- nnc: 0 \leq c

Shows:

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

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

**Healthiness**

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

**definition**

$ health :: (('s \Rightarrow real) \Rightarrow ('s \Rightarrow real)) \Rightarrow bool$

**where**

$ healthy t \leftarrow feasible t \wedge mono-trans t \wedge scaling t$

**lemma** $healthyI[\text{intro}]:$

$[[ feasible t; mono-trans t; scaling t ]] \Rightarrow healthy t$

(proof)

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

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

$healthy t \Rightarrow mono-trans t$

(proof)

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

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

$healthy t \Rightarrow feasible t$

(proof)

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

$healthy t \Rightarrow scaling t$

(proof)

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

$[[ healthy t; bounded-by b P; nneg P ]] \Rightarrow bounded-by b (t P)$

(proof)

**lemma** $healthy-bounded-byD2:$

$[[ healthy t; bounded-by b P; sound P ]] \Rightarrow bounded-by b (t P)$

(proof)

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

$[[ healthy t; sound P ]] \Rightarrow bounded (t P)$

(proof)

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

$[[ healthy t; sound P ]] \Rightarrow nneg (t P)$

(proof)

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

$[[ healthy t; bounded-by b P; nneg P ]] \Rightarrow nneg (t P)$

(proof)

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

$[[ healthy t; sound P ]] \Rightarrow sound (t P)$

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

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

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

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

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

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

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

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

Lemma healthy-equiv1:
fixes t::('s ⇒ real) ⇒ 's ⇒ real and u
assumes equiv: equiv-trans t u
and healthy: healthy t
shows healthy u
(proof)

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

Lemma healthy-scale:
fixes t::('s ⇒ real) ⇒ 's ⇒ real
assumes ht: healthy t and nc: 0 ≤ c and bc: c ≤ 1
shows healthy (\lambda P s. c * t P s)
(proof)

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

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

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

**definition**

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

where

\[ \text{nearly-healthy} t \longleftrightarrow (\forall P. \ \text{unitary} P \rightarrow \text{unitary} (t P)) \land \]

\[ (\forall P Q. \ \text{unitary} P \rightarrow \text{unitary} Q \rightarrow P \vdash Q \rightarrow t P \vdash t Q) \]

**lemma** nearly-healthyI[intro]:

\[ [ \bigwedge P. \ \text{unitary} P \Rightarrow \text{unitary} (t P); \]

\[ \bigwedge P Q. [ \text{unitary} P; \ \text{unitary} Q; P \vdash Q \]\]

\[ \Rightarrow t P \vdash t Q ] \Rightarrow \text{nearly-healthy} t \]

(proof)

**lemma** nearly-healthy-monoD[dest]:

\[ [ \text{nearly-healthy} t; P \vdash Q; \ \text{unitary} P; \ \text{unitary} Q ] \Rightarrow t P \vdash t Q \]

(proof)

**lemma** nearly-healthy-unitaryD[dest]:

\[ [ \text{nearly-healthy} t; \ \text{unitary} P ] \Rightarrow \text{unitary} (t P) \]

(proof)

**lemma** healthy-nearly-healthy[dest]:

assumes \( \text{ht} \):

\( \text{healthy} t \)

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

(proof)

**lemmas** nearly-healthy-id[iff] =

\( \text{healthy-nearly-healthy}[\text{OF healthy-id, unfolded id-def}] \)

### 3.2.3 Sublinearity

As already mentioned, the core healthiness property (aside from feasibility and continuity) for transformers is sublinearity: The transformation of a quasi-linear combination of sound expectations is greater than the same combination applied to the transformation of the expectations themselves. The term \( x \odot y \) represents truncated subtraction i.e. \( \text{max} (x - y) (0::'a) \) (see Section 4.13.1).

**definition** sublinear ::

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

where

\[ \text{sublinear} t \longleftrightarrow (\forall a b c P Q s. (\text{sound} P \land \text{sound} Q \land 0 \leq a \land 0 \leq b \land 0 \leq c) \]

\[ \rightarrow \]

\[ a \ast t P s + b \ast t Q s \ominus c \]

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

**lemma** sublinearI[intro]:


CHAPTER 3. SEMANTIC STRUCTURES

Sub-additivity

\[
\begin{align*}
Q &= tP \cap uP \\
Q(x) &= Q(x) + Q(y) \\
Q(y) &= Q(x) + Q(y)
\end{align*}
\]

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

It is easier to see the relevance of sublinearity by breaking it into several component properties, as in the following sections.

Sub-additivity

\[
\text{definition } \text{sub-add} ::
\]

\[
(\forall P \ s \ . \ (\text{sound } P \ \& \ \text{sound } Q) \rightarrow t \ P \ s \ + \ t \ Q \ s \ \leq \ t \ (\lambda s'. \ P \ s' \ + \ Q \ s') \ s)
\]

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

Here \( P \) is an affine function (expectation) \( \text{real} \Rightarrow \text{real} \), restricted to some finite interval. In practice the state space (the left-hand type) is typically
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-addl[intro]:**

$$[\forall P \ Q \ s. [\text{sound } P; \text{sound } Q] \Rightarrow \hspace{1cm} t \ P \ s + t \ Q \ s \leq t \ (\lambda s'. \ P \ s' + Q \ s') \ s ] \Rightarrow \text{sub-add } t$$

(proof)

**lemma sub-addl2:**

$$[\forall P \ Q. [\text{sound } P; \text{sound } Q] \Rightarrow \hspace{1cm} \lambda s. \ t \ P \ s + t \ Q \ s \vdash t \ (\lambda s. \ P \ s + Q \ s)] \Rightarrow \text{sub-add } t$$

(proof)

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

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

(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 «N» 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: \( \forall x. \text{sound} (P x) \)
  shows \( (\lambda x. \sum y \in S. t (P y) x) \leq (\lambda x. \sum y \in S. P y x) \)
⟨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 \( (\sum y \in \{s. G s\}. P y \ast t \leftarrow \lambda z. z = y \ast s) + \\
(\sum y \in \{s. \neg G s\}. P y \ast t \leftarrow \lambda z. z = y \ast s) \leq t P s \)
⟨proof⟩

Sub-distributivity

definition sub-distrib ::
(\'(s ⇒ real) ⇒ \'(s ⇒ real)) ⇒ bool
where
  sub-distrib t ←→ (\forall P s. sound P → t P s ⊗ 1 ≤ t (\lambda s'. P s' ⊙ 1) s)

lemma sub-distribI[intro]:
[ \forall P s. sound P → t P s ⊗ 1 ≤ t (\lambda s'. P s' ⊙ 1) s ] → sub-distrib t
⟨proof⟩
3.2. EXPECTATION TRANSFORMERS

lemma sub-distribI2:
\[ \forall P. \text{sound } P \implies \lambda s. t \; P \; s \; \ominus \; 1 \vdash t \; (\lambda s'. \; P \; s' \; \ominus \; 1) \; s \leadsto \text{sub-distrib } t \]
(proof)

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

lemma equiv-sub-distrib:
fixes t ::= (\'s \rightarrow \text{real}) \rightarrow (\'s \rightarrow \text{real})
assumes eq: equiv-trans t u and sd: sub-distrib t
shows sub-distrib u
(proof)

Sublinearity implies sub-distributivity:

lemma sublinear-sub-distrib:
fixes t ::= (\'s \rightarrow \text{real}) \rightarrow (\'s \rightarrow \text{real})
assumes slt: sublinear t
shows sub-distrib t
(proof)

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

lemma sd-sa-sublinear:
fixes t ::= (\'s \rightarrow \text{real}) \rightarrow (\'s \rightarrow \text{real})
assumes sdt: sub-distrib t and sat: sub-add t and ht: healthy t
shows sublinear t
(proof)

Sub-conjunctivity

definition
sub-conj :: ((\'s \rightarrow \text{real}) \rightarrow (\'s \rightarrow \text{real}) \rightarrow \text{bool})
where
sub-conj t \equiv \forall P \; Q. \; (\text{sound } P \; \land \; \text{sound } Q) \longrightarrow t \; P \; \&\& \; t \; Q \vdash t \; (P \; \&\& \; Q)

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

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

lemma sub-conj-wp-twice:
\textbf{fixes} f::'s ⇒ (('s ⇒ real) ⇒ 's ⇒ real)
\textbf{assumes} all: ∀ s. sub-conj (f s)
\textbf{shows} sub-conj (λP s. f s P s)
\langle proof \rangle

Sublinearity implies sub-conjunctivity:
\textbf{lemma} sublinear-sub-conj:
\textbf{fixes} t:('s ⇒ real) ⇒ 's ⇒ real
\textbf{assumes} slt: sublinear t
\textbf{shows} sub-conj t
\langle proof \rangle

\textbf{Sublinearity under equivalence}

Sublinearity is preserved by equivalence.
\textbf{lemma} equiv-sublinear:
\[ \text{equiv-trans } t u; \text{sublinear } t \] =⇒ sublinear u
\langle proof \rangle

\textbf{3.2.4 Determinism}

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

\textbf{Additivity}

Full additivity is not generally satisfied. It holds for (sub-)probabilistic transformers however.
\textbf{definition}
\textit{additive} :: (('[a ⇒ real] ⇒ '[a ⇒ real]) ⇒ bool
\textbf{where}
\textit{additive} t ≡ ∀ P Q. (sound P ∧ sound Q) \textit{⇒} t (λs. P s + Q s) = (λs. t P s + t Q s)

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

\textbf{lemma} additiveI[intro]:
\[ \text{λP Q s. } \text{[sound } P; \text{sound } Q \] =⇒ t (λs. P s + Q s) s = t P s + t Q s \] =⇒ additive t
\langle proof \rangle

Additivity is strictly stronger than sub-additivity.
\textbf{lemma} additive-sub-add:
\textit{additive } t =⇒ \textit{sub-add } t
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 $sPz$: $\forall z. \text{sound} (P z)$
- **shows** $t (\lambda x. \sum_{y \in S}. P y x) = (\lambda x. \sum_{y \in S}. t (P y) x)$

**proof**

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

**lemma** additive-delta-split:

- **fixes** $t$::('s::finite $\Rightarrow$ real) $\Rightarrow$ 's $\Rightarrow$ real
- **assumes** additive: additive $t$
  and $ht$: healthy $t$
  and $sP$: sound $P$
- **shows** $t P x = (\sum_{y \in \text{UNIV}}. P y * t « \lambda z. z = y » x) + (\sum_{y \in \{s. \neg G s\}}. P y * t « \lambda z. z = y » x)$

**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 $\Rightarrow$ real) $\Rightarrow$ 's $\Rightarrow$ real
- **assumes** additive: additive $t$
  and $ht$: healthy $t$
  and $sP$: sound $P$
- **shows** $t P x = (\sum_{y \in \{s. G s\}}. P y * t « \lambda z. z = y » x) + (\sum_{y \in \{s. \neg G s\}}. P y * t « \lambda z. z = y » x)$

**proof**

Maximality

**definition**

- maximal :: (('a $\Rightarrow$ real) $\Rightarrow$ 'a $\Rightarrow$ real) $\Rightarrow$ bool

**where**

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

**lemma** maximalI[intro]:

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

**proof**

**lemma** maximalD[dest]:

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

**Definition**
\[
\text{determ} :: (('a ⇒ real) ⇒ 'a ⇒ real) ⇒ bool
\]

where
\[
determ t ≡ \text{additive } t ∧ \text{maximal } t
\]

**Lemma**
\[
determI[intro]: \begin{cases} \text{additive } t; \text{maximal } t \end{cases} ⇒ \text{determ } t
\]

**Lemma**
\[
determ-additiveD[intro]: \text{determ } t ⇒ \text{additive } t
\]

**Lemma**
\[
determ-maximalD[intro]: \text{determ } t ⇒ \text{maximal } t
\]

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

**Lemma**
\[
determ-negate: \begin{cases} \text{assumes } \text{determ: } \text{determ } t \end{cases} \text{shows } t \overset{P}{\rightarrow} s + t \overset{\neg P}{\rightarrow} s = 1
\]

### 3.2.5 Modular Reasoning

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

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

**Lemma**
\[
\text{entails-combine}: \begin{cases} \text{assumes } wp1: P \models t R \\
\text{and } wp2: Q \models t S \\
\text{and } sc: \text{sub-conj } t \\
\text{and } sR: \text{sound } R \\
\text{and } sS: \text{sound } S \\
\text{shows } P \&\& Q \models t (R \&\& S) \end{cases}
\]

These allow mismatched results to be composed

**Lemma**
\[
\text{entails-strengthen-post}: \begin{cases} P \models t Q; \text{healthy } t; \text{sound } R; Q \models R; \text{sound } Q \end{cases} ⇒ P \models t R
\]
3.2. EXPECTATION TRANSFORMERS

lemma entails-weaken-pre:
\[ \begin{array}{l}
[ Q \vdash t \Rightarrow R; \ P \vdash Q ] \implies P \vdash t \ R \\
\end{array}
\]
(proof)

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

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

3.2.6 Transforming Standard Expectations

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

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

lemma use-premise:
\begin{itemize}
    \item \textbf{assumes} h: healthy t \text{ and } wP: \forall s . P s \implies 1 \leq t \ Q s
    \item \textbf{shows} \ (\lambda s . c * P s) \vdash t \ (\lambda s . c * Q s)
\end{itemize}
(proof)

The other direction works too.

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

Predicate conjunction behaves as expected:

lemma conj-post:
\[ \begin{array}{l}
[ P \vdash t \ « \lambda s . Q s \land R s »; \text{healthy t} ] \implies P \vdash t \ « Q s »
\end{array}
\]
(proof)

Similar to [\text{healthy ?t}; \forall s . ?P s \implies 1 \leq ?t \ « \ ?Q s »] \implies « ?P » \vdash ?t « ?Q », but more general.

lemma entails-peconj- assumption:
\begin{itemize}
    \item \textbf{assumes} f: \text{feasible t and } wP: \forall s . P s \implies Q s \leq t \ R s
    \item \textbf{and} uQ: \text{unitary Q and } uR: \text{unitary R}
    \item \textbf{shows} \ (\forall s . P s \land Q s) \vdash t \ R
\end{itemize}
(proof)
end
3.3 Induction

theory Induction
  imports Expectations Transformers
begin

3.3.1 The Lattice of Expectations

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

Finding a lower bound is easy (it’s $\lambda$. 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 $\forall \ P. \ sound \ P \rightarrow \ healthy \ (t \ P)$ and $\forall Q. \ sound \ Q \rightarrow \ healthy \ (\lambda P. \ t \ P \ Q)$. Thus by feasibility, $t \ P \ Q$ must be bounded by $bound-of \ P$. 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 $\lambda$. $bound-of \ P$.

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

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

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

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

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

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

(proof)

lemma Sup-exp-sound:
  assumes sS: ∀P. P ∈ S → sound P
  and bS: ∀P. P ∈ S → bounded-by b P
  shows sound (Sup-exp S)
⟨proof⟩

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

lemma lfp-exp-lowerbound:
  [ t P ≤ P; sound P ] ⇒ lfp-exp t ≤ P
⟨proof⟩

lemma lfp-exp-greatest:
  [ [ P. t P ≤ P; sound P ] → Q ≤ P; sound Q; t R ⊢ R; sound R ] → Q ≤ lfp-exp t
⟨proof⟩

lemma feasible-lfp-exp-sound:
  feasible t ⇒ sound (lfp-exp t)
⟨proof⟩

lemma lfp-exp-sound:
  assumes fR: t R ⊢ R and sR: sound R
  shows sound (lfp-exp t)
⟨proof⟩

lemma lfp-exp-bound:
  (∀P. unitary P → unitary (t P)) → bounded-by 1 (lfp-exp t)
⟨proof⟩

lemma lfp-exp-unitary:
  (∀P. unitary P → unitary (t P)) → unitary (lfp-exp t)
⟨proof⟩

lemma lfp-exp-lemma2:
  fixes t::'s trans
  assumes st: ∀P. sound P → sound (t P)
  and mt: mono-trans t
  and fR: t R ⊢ R and sR: sound R
  shows t (lfp-exp t) ≤ lfp-exp t
⟨proof⟩

lemma lfp-exp-lemma3:
  assumes st: ∀P. sound P → sound (t P)
  and mt: mono-trans t
  and fR: t R ⊢ R and sR: sound R
shows \( lfp\text{-}exp \ t \leq t \ (lfp\text{-}exp \ t) \)

proof

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

proof

definition \( gfp\text{-}exp :: \{ s \text{ trans} \implies \{ s \text{ expect} \} \} \)
where
\[ gfp\text{-}exp \ t = \text{Sup-exp} \ \{ P. \text{unitary} P \land P \leq t \ P \} \]

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

proof

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

proof

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

proof

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

proof

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

proof

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

proof

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

proof

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

⟨proof⟩

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 \( \lambda - \). \( \theta :: \cdot \), and the top element is the greatest allowed by feasibility: \( \lambda P - \) \( \cdot \). bound-of \( P \).

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

lemma Inf-trans-lower:
\[ \forall t \in S; \forall u \in S. \forall P. \text{sound} \; P \longrightarrow \text{sound} \; (u \; P) \] \( \Rightarrow \) le-trans \( (\text{Inf-trans} \; S) \; t \)
⟨proof⟩

lemma Inf-trans-greatest:
\[ \forall S \neq \{\}; \forall t \in S. \forall P. \text{le-trans} \; u \; t \] \( \Rightarrow \) le-trans \( u \) \( (\text{Inf-trans} \; S) \)
⟨proof⟩

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

lemma Sup-trans-upper:
\[ \forall t \in S; \forall u \in S. \forall P. \text{unitary} \; P \longrightarrow \text{unitary} \; (u \; P) \] \( \Rightarrow \) le-utrans \( t \) \( (\text{Sup-trans} \; S) \)
⟨proof⟩

lemma Sup-trans-upper2:
\[ \forall t \in S; \forall u \in S. \forall P. (\text{nneg} \; P \land \text{bounded-by} \; b \; P) \longrightarrow (\text{nneg} \; (u \; P) \land \text{bounded-by} \; b \; (u \; P)); \]
\[ \text{nneg} \; P; \text{bounded-by} \; b \; P \] \( \Rightarrow \) \( t \; P \Updownarrow \; \text{Sup-trans} \; S \; P \)
⟨proof⟩

lemma Sup-trans-least:
\[ \forall t \in S; \text{le-utrans} \; t \; u; \forall P. \text{unitary} \; P \longrightarrow \text{unitary} \; (u \; P) \] \( \Rightarrow \) le-utrans \( (\text{Sup-trans} \; S) \; t \) \( u \)
⟨proof⟩

lemma Sup-trans-least2:
\[ \forall t \in S; \forall P. \text{nneg} \; P \longrightarrow \text{bounded-by} \; b \; P \longrightarrow t \; P \Updownarrow \; u \; P; \]
\[ \forall u \in S. \forall P. (\text{nneg} \; P \land \text{bounded-by} \; b \; P) \longrightarrow (\text{nneg} \; (u \; P) \land \text{bounded-by} \; b \; (u \; P)); \]
\[ \text{nneg} \; P; \text{bounded-by} \; b \; P; \forall P. \; \text{nneg} \; P; \text{bounded-by} \; b \; P \] \( \Rightarrow \) \( \text{nneg} \; (u \; P) \]
\( \Rightarrow \) Sup-trans \( S \; P \Updownarrow \; u \; P \)
⟨proof⟩

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

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

definition lfp-trans :: (’s trans ⇒ ’s trans) ⇒ ’s trans
where lfp-trans $T$ = Inf-trans { $t$. ($\forall P$. sound $P$ ⟷ sound $(t P)$) ∧ le-trans $(T t)$ }

lemma lfp-trans-lowerbound:
$\begin{align*}
\text{le-trans $(T t)$ } & \implies \text{le-trans $(lfp-trans T t)$ } \\
\text{proof }
\end{align*}$

lemma lfp-trans-greatest:
$\begin{align*}
\text{le-trans $(T t)$ } & \implies \text{le-trans $(lfp-trans T t)$ } \\
\text{proof }
\end{align*}$

lemma lfp-trans-sound:
fixes $P Q$: ’s expect
assumes $sP$: sound $P$
and $fv$: le-trans $(T v)$ $v$
and $sv$: $\forall P$. sound $P$ ⟷ sound $(v P)$
shows sound $(lfp-trans T P)$
(proof)

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

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

lemma lfp-trans-lemma3:
fixes $v$: ’s trans
assumes mono: \( \forall t u. \quad \text{le-trans } t u; \land P. \quad \text{sound } P \Rightarrow \text{sound } (t P); \land P. \quad \text{sound } P \Rightarrow \text{sound } (u P) \Rightarrow \text{le-trans } (T t) (T u) \)

and sT: \( \forall t P. \quad [ \land Q. \quad \text{sound } Q \Rightarrow \text{sound } (t Q); \land \text{sound } P ] \Rightarrow \text{sound } (T t P) \)

and fT: \( \text{le-trans } (T v) v \)

and sv: \( \land P. \quad \text{sound } P \Rightarrow \text{sound } (v P) \)

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

⟨proof⟩


lemma lfp-trans-unfold:

fixes \( P::\text{‘s expect} \)

assumes mono: \( \forall t u. \quad \text{le-trans } t u; \land P. \quad \text{sound } P \Rightarrow \text{sound } (t P); \land P. \quad \text{sound } P \Rightarrow \text{sound } (u P) \Rightarrow \text{le-trans } (T t) (T u) \)

and sT: \( \forall t P. \quad [ \land Q. \quad \text{sound } Q \Rightarrow \text{sound } (t Q); \land \text{sound } P ] \Rightarrow \text{sound } (T t P) \)

and fT: \( \text{le-trans } (T v) v \)

and sv: \( \land P. \quad \text{sound } P \Rightarrow \text{sound } (v P) \)

shows \( \text{equiv-trans } (lfp-trans T) (T (lfp-trans T)) \)

⟨proof⟩


definition gfp-trans :: \( (\text{‘s trans } \Rightarrow \text{‘s trans}) \Rightarrow \text{‘s trans} \)

where \( gfp-trans T = \text{Sup-trans } \{ t. \quad (\forall P. \quad \text{unitary } P \Rightarrow \text{unitary } (t P)) \land \text{le-trans } t (T t) \} \)

lemma gfp-trans-upperbound:

\[ \lnot \text{le-trans } t (T t); \land P. \quad \text{unitary } P \Rightarrow \text{unitary } (t P) \] \Rightarrow \text{le-trans } t (gfp-trans T) \n
⟨proof⟩


lemma gfp-trans-least:

\[ \lnot \text{le-trans } t (T t); \land P. \quad \text{unitary } P \Rightarrow \text{unitary } (t P) \] \Rightarrow \text{le-trans } t u; \land P. \quad \text{unitary } P \Rightarrow \text{unitary } (u P) \Rightarrow \text{le-trans } (gfp-trans T) u \n
⟨proof⟩


lemma gfp-trans-unitary:

fixes \( P::\text{‘s expect} \)

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

shows \( \text{unitary } (gfp-trans T P) \)

⟨proof⟩


lemma gfp-trans-lemma2:

assems mono: \( \forall t u. \quad \text{le-trans } t u; \land P. \quad \text{unitary } P \Rightarrow \text{unitary } (t P); \land P. \quad \text{unitary } P \Rightarrow \text{unitary } (u P) \Rightarrow \text{le-trans } (T t) (T u) \)

and hT: \( \forall t P. \quad [ \land Q. \quad \text{unitary } Q \Rightarrow \text{unitary } (t Q); \land \text{unitary } P ] \Rightarrow \text{unitary } (T t P) \)

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

⟨proof⟩
3.3.3 Tail Recursion

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

**Lemma gfp-trans-unfold:**

**Assumes**
- $\forall t. [\text{le-utrans } t \ u; \forall P. \text{unitary } P \Rightarrow \text{unitary } (t \ P)]$
- $\forall P. \text{unitary } P \Rightarrow \text{unitary } (u \ P)] \Rightarrow \text{le-utrans } (T \ t) \ (T \ u)$

**And** $hT: \forall t. P. [\forall Q. \text{unitary } Q \Rightarrow \text{unitary } (t \ Q); \text{unitary } P \Rightarrow \text{unitary } (T \ t \ P)]$

**Shows** $\text{le-utrans } (T \ (\text{gfp-trans } T)) \ (\text{gfp-trans } T)$

(Proof)

**Lemma gfp-trans-lemma3:**

**Assumes**
- $\forall t. [\text{le-utrans } t \ u; \forall P. \text{unitary } P \Rightarrow \text{unitary } (t \ P); \forall P. \text{unitary } P \Rightarrow \text{unitary } (u \ P)] \Rightarrow \text{le-utrans } (T \ t) \ (T \ u)$

**And** $hT: \forall t. P. [\forall Q. \text{unitary } Q \Rightarrow \text{unitary } (t \ Q); \text{unitary } P \Rightarrow \text{unitary } (T \ t \ P)]$

**Shows** $\text{le-utrans } (T \ (\text{gfp-trans } T)) \ (\text{gfp-trans } T)$

(Proof)

**3.3.3 Tail Recursion**

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

**Lemma lfp-pulldown:**

**Fixes** $P: \text{'}s \ \text{expect}$

**Assumes**
- $\forall u. \text{unitary } P \Rightarrow T \ u \ P = t \ P (u \ P)$
- $\forall t. [\forall Q. \text{unitary } Q \Rightarrow \text{unitary } (t \ Q); \text{unitary } P \Rightarrow \text{unitary } (T \ t \ P)]$

**And** $fT: \forall P. [\forall Q. \text{unitary } Q \Rightarrow \text{unitary } (t \ P Q)]$

**And** $ft: \forall P. \text{unitary } P \Rightarrow \text{unitary } (t \ P Q)$

**And** $mt: \forall P. \text{unitary } P \Rightarrow \text{unitary } (t \ P R)$

**And** $uP: \text{unitary } P$

**And** $\text{monoT}: \forall t. u. [\text{le-utrans } t \ u; \forall P. \text{unitary } P \Rightarrow \text{unitary } (t \ P)]$

**Shows** $\text{gfp-trans } T \ P = \text{gfp-exp } (t \ P) \ (\text{is } ?X \ P = ?Y \ P)$

(Proof)

**Lemma lfp-pulldown:**

**Fixes** $P: \text{'}s \ \text{trans}$ and $T: \text{'}s \ \text{trans}$

**Assumes**
- $\forall u. \text{sound } P \Rightarrow T \ u \ P = t \ P (u \ P)$
- $\forall t. P. \text{sound } P \Rightarrow \text{sound } (t \ P Q)$
- $\forall t. P. \text{sound } P \Rightarrow \text{mono-trans } (t \ P)$
- $\forall t. P. \text{sound } P \Rightarrow \text{sound } (T \ t \ P)$
- $\forall t. [\forall Q. \text{sound } Q \Rightarrow \text{sound } (t \ P Q); \text{sound } P \Rightarrow \text{sound } (u \ P)] \Rightarrow \text{le-utrans } (T \ t \ P)$
- $\forall t. [\forall Q. \text{sound } Q \Rightarrow \text{sound } (T \ t \ P); \text{sound } P \Rightarrow \text{sound } (T \ t \ P)]$

**And** $fv: \text{le-utrans } (T \ v) v$

**And** $sv: \forall P. \text{sound } P \Rightarrow \text{sound } (v \ P)$

**And** $sp: \text{sound } P$
shows \( \text{lfp-trans } T P = \text{lfp-exp } (t P) \) (is \( ?X P = ?Y P \))

\[ \text{Inf-utrans } :: \; 's \; \text{trans set } \Rightarrow \; 's \; \text{trans} \]

where \( \text{Inf-utrans } S = (\text{if } S = \{\} \; \text{then } \lambda P \; s. \; 1 \; \text{else } \text{Inf-trans } S) \)

lemma \( \text{Inf-utrans-lower} \):
\[
\begin{align*}
& \; \{ t \in S; \forall t \in S. \; \forall P. \; \text{unitary } P \rightarrow \text{unitary } (t P) \} \implies \text{le-utrans } (\text{Inf-utrans } S) \; t \\
\end{align*}
\]
(proof)

lemma \( \text{Inf-utrans-greatest} \):
\[
\begin{align*}
& \; \{ \forall P. \; \text{unitary } P \implies \text{unitary } (t P); \forall u \in S. \; \text{le-utrans } t \; u \} \implies \text{le-utrans } t \\
& \; (\text{Inf-utrans } S) \\
\end{align*}
\]
(proof)

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, \( \lambda P. 0 :: 'c \), or always succeeds, \( \lambda P. 1 :: 'c \).

definition Abort :: 's prog
where
Abort ≡ \( \lambda ab P s. \) if \( ab \) then 0 else 1

Skip does nothing at all.

definition Skip :: 's prog
where
Skip ≡ \( \lambda P. P \)

Apply lifts a state transformer into the space of programs.

definition Apply :: ('s ⇒ 's) ⇒ 's prog
where
Apply \( f \) ≡ \( \lambda ab P s. P f s \)

Seq is sequential composition.

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

\( PC \) is probabilistic choice between programs.

definition PC :: 's prog ⇒ ('s ⇒ real) ⇒ 's prog ⇒ 's prog
(\( \text{infix} \oplus [58,57,57] \) 59)
where
PC \( a \; P \; b \equiv \lambda ab Q s. \; P \; s \; * \; a \; ab \; Q \; s \; + \; (1 \; - \; P \; s) \; * \; b \; ab \; Q \; s \)
**CHAPTER 4. THE PGCL LANGUAGE**

*DC* is demonic choice between programs.

**definition**

\[ DC :: 's prog \Rightarrow 's prog \Rightarrow 's prog (- \coprod - [58,57] 57) \]

**where**

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

*AC* is angelic choice between programs.

**definition**

\[ AC :: 's prog \Rightarrow 's prog \Rightarrow 's prog (- \coprod - [58,57] 57) \]

**where**

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

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

**definition**

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

**where**

\[ Embed t = (\lambda ab. t) \]

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

**definition**

\[ Mu :: ('s prog \Rightarrow 's prog) \Rightarrow 's prog (\text{binder } \mu 50) \]

**where**

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

*repeat* expresses finite repetition

**primrec**

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

**where**

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

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

**definition**

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

**where**

\[ SetDC f S \equiv \lambda ab P s. \inf ((\lambda a. f a \; ab \; P \; s) \; S \; s) \]

**syntax**

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

**translations**

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

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

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

**definition**

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

**where**

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

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

**definition**

\[ Bind :: ('s \Rightarrow 'a) \Rightarrow ('a \Rightarrow 's prog) \Rightarrow 's prog \]
where
\[ \text{Bind } g \ f \ ab \equiv \lambda P \ s. \ \text{let } a = g \ s \ \text{in } f \ a \ ab \ P \ s \]

This gives us something like let syntax

**syntax** -Bind :: pttrn => ('s => 'a) => 's prog => 's prog
(- is in [55,55,55])

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

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

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

**syntax** -PBind :: pttrn => ('s => real) => 's prog => 's prog
(bind - at - in [55,55,55])

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

**syntax** -DBind :: pttrn => ('s => 'a set) => 's prog => 's prog
(bind - from - in [55,55,55])

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

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

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

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

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

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

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

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

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

**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 ≡ pr True
definition

\[ \text{wlp} :: \mathit{prog} \Rightarrow \mathit{trans} \]

where

\[ \text{wlp pr} \equiv \text{pr False} \]

If-Then-Else as a degenerate probabilistic choice.

abbreviation\(^{\text{(input)}}\)

\[ \text{if-then-else} :: [\mathit{trans} \Rightarrow \mathit{bool} \land \mathit{prog}] \Rightarrow \mathit{prog} \]

\[(\text{If - Then - Else - 58})\]

where

\[ \text{If P Then a Else b} \equiv a \cdot P \otimes b \]

Syntax for loops

abbreviation

\[ \text{do-while} :: [\mathit{trans} \Rightarrow \mathit{bool} \land \mathit{prog}] \Rightarrow \mathit{prog} \]

\[(\text{do - \rightarrow// (4 -) //od})\]

where

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

4.1.2 Unfolding rules for non-recursive primitives

lemma \textit{eval-wp-Abort}:

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

\langle proof \rangle

lemma \textit{eval-wlp-Abort}:

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

\langle proof \rangle

lemma \textit{eval-wp-Skip}:

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

\langle proof \rangle

lemma \textit{eval-wlp-Skip}:

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

\langle proof \rangle

lemma \textit{eval-wp-Apply}:

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

\langle proof \rangle

lemma \textit{eval-wlp-Apply}:

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

\langle proof \rangle

lemma \textit{eval-wp-Seq}:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Use simp add:wp_eval to fully unfold a program fragment

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

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

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

Use these as simp rules to clear out Skips

lemmas 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:**
\[
\text{healthy } (\text{wp } \text{Abort})
\]
\[
\text{proof}
\]

**lemma nearly-healthy-wlp-Abort:**
\[
\text{nearly-healthy } (\text{wlp } \text{Abort})
\]
\[
\text{proof}
\]

**lemma healthy-wp-Skip:**
\[
\text{healthy } (\text{wp } \text{Skip})
\]
\[
\text{proof}
\]

**lemma nearly-healthy-wlp-Skip:**
\[
\text{nearly-healthy } (\text{wlp } \text{Skip})
\]
\[
\text{proof}
\]

**lemma healthy-wp-Seq:**
\[
\text{fixes } t::\text{\`s prog and } u
\]
\[
\text{assumes } \text{ht: healthy (wp t) and hu: healthy (wp u)}
\]
\[
\text{shows } \text{healthy (wp (t ;; u))}
\]
\[
\text{proof}
\]

**lemma nearly-healthy-wlp-Seq:**
\[
\text{fixes } t::\text{\`s prog and } u
\]
\[
\text{assumes } \text{ht: nearly-healthy (wlp t) and hu: nearly-healthy (wlp u)}
\]
\[
\text{shows } \text{nearly-healthy (wlp (t ;; u))}
\]
\[
\text{proof}
\]

**lemma healthy-wp-PC:**
\[
\text{fixes } f::\text{\`s prog}
\]
\[
\text{assumes } \text{kf: healthy (wp f) and kg: healthy (wp g) and uP: unitary P}
\]
\[
\text{shows } \text{healthy (wp (f } \oplus \text{ g))}
\]
\[
\text{proof}
\]

**lemma nearly-healthy-wlp-PC:**
\[
\text{fixes } f::\text{\`s prog}
\]
\[
\text{assumes } \text{kf: nearly-healthy (wlp f) and kg: nearly-healthy (wlp g) and uP: unitary P}
\]
\[
\text{shows } \text{nearly-healthy (wlp (f } \oplus \text{ g))}
\]
\[
\text{proof}
\]

**lemma healthy-wp-DC:**
\[
\text{fixes } f::\text{\`s prog}
\]
assumes $hf$: healthy ($wp \ f$) and $hg$: healthy ($wp \ g$)
shows healthy ($wp \ (f \bigcap g)$)
⟨proof⟩

**lemma** nearly-healthy-wlp-DC:
fixes $f$::′s prog
assumes $hf$: nearly-healthy ($wlp \ f$) and $hg$: nearly-healthy ($wlp \ g$)
shows nearly-healthy ($wlp \ (f \bigcap g)$)
⟨proof⟩

**lemma** healthy-wp-AC:
fixes $f$::′s prog
assumes $hf$: healthy ($wp \ f$) and $hg$: healthy ($wp \ g$)
shows healthy ($wp \ (f \bigcup g)$)
⟨proof⟩

**lemma** nearly-healthy-wlp-AC:
fixes $f$::′s prog
assumes $hf$: nearly-healthy ($wlp \ f$) and $hg$: nearly-healthy ($wlp \ g$)
shows nearly-healthy ($wlp \ (f \bigcup g)$)
⟨proof⟩

**lemma** healthy-wp-Embed:
healthy $t$ $\implies$ healthy ($wp \ (Embed \ t)$)
⟨proof⟩

**lemma** nearly-healthy-wlp-Embed:
nearly-healthy $t$ $\implies$ nearly-healthy ($wlp \ (Embed \ t)$)
⟨proof⟩

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

**lemma** nearly-healthy-wlp-repeat:
assumes $h-a$: nearly-healthy ($wlp \ a$)
shows nearly-healthy ($wlp \ (repeat \ n \ a)$) (is ?X n)
⟨proof⟩

**lemma** healthy-wp-SetDC:
fixes prog::′b ⇒ ′a prog and $S$::′a ⇒ ′b set
assumes healthy: $\forall x. x \in S \implies$ healthy ($wp \ (prog \ x)$) and nonempty: $\exists x. x \in S$
shows healthy ($wp \ (SetDC \ prog \ S)$) (is healthy ?T)
⟨proof⟩
4.2. HEALTHINESS

**Lemma nearly-healthy-wlp-SetDC:**

- **Fixes** $\text{prog} :: 'b \\Rightarrow 'a \text{ prog}$ and $\text{S} :: 'a \\Rightarrow 'b \text{ set}$
- **Assumes** healthy: $\forall x. x \in S \implies \text{nearly-healthy (wlp (prog x))}$
- **And** nonempty: $\exists x. x \in S$
- **Shows** nearly-healthy (wlp (SetDC prog S)) (is nearly-healthy ?T)

**Proof**

**Lemma healthy-wp-SetPC:**

- **Fixes** $\text{p} :: 's \\Rightarrow 'a \\Rightarrow \text{real}$ and $\text{f} :: 'a \\Rightarrow 's \text{ prog}$
- **Assumes** healthy: $\forall a. a \in \text{supp} (p \ s) \implies \text{healthy (wp (f a))}$
- **And** sound: $\forall s. \text{sound (p \ s)}$
- **And** sub-dist: $\forall s. (\sum a \in \text{supp} (p \ s). p \ s \ a) \leq 1$
- **Shows** healthy (wp (SetPC f p)) (is healthy ?X)

**Proof**

**Lemma nearly-healthy-wlp-SetPC:**

- **Fixes** $\text{p} :: 's \\Rightarrow 'a \\Rightarrow \text{real}$ and $\text{f} :: 'a \\Rightarrow 's \text{ prog}$
- **Assumes** healthy: $\forall a. a \in \text{supp} (p \ s) \implies \text{nearly-healthy (wlp (f a))}$
- **And** sound: $\forall s. \text{sound (p \ s)}$
- **And** sub-dist: $\forall s. (\sum a \in \text{supp} (p \ s). p \ s \ a) \leq 1$
- **Shows** nearly-healthy (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** $\text{f} :: 's \\Rightarrow 'a$
- **Assumes** $\text{hsub: } \forall s. \text{healthy (wp (p (f s)))}$
- **Shows** healthy (wp (Bind f p))

**Proof**

**Lemma nearly-healthy-wlp-Bind:**

- **Fixes** $\text{f} :: 's \\Rightarrow 'a$
- **Assumes** $\text{hsub: } \forall s. \text{nearly-healthy (wlp (p (f s)))}$
- **Shows** nearly-healthy (wlp (Bind f p))

**Proof**

**4.2.2 Healthiness for Loops**

**Lemma wp-loop-step-mono:**

- **Fixes** $\text{t u} :: 's \text{ trans}$
assumes $hb$: healthy ($wp$ body)
    and $le$: le-trans $t$ $u$
    and $ht$: $\forall P. \text{sound } P \Rightarrow \text{sound } (t P)$
    and $hu$: $\forall P. \text{sound } P \Rightarrow \text{sound } (u P)$
shows le-trans ($wp$ (body ;; Embed $t \leftarrow G$ ⊕ Skip))
    ($wp$ (body ;; Embed $u \leftarrow G$ ⊕ Skip))
⟨proof⟩

lemma $wlp$-loop-step-mono:
    fixes $t$ $u$::′s trans
    assumes $mb$: nearly-healthy ($wlp$ body)
        and $le$: le-utrans $t$ $u$
        and $ht$: $\forall P. \text{unitary } P \Rightarrow \text{unitary } (t P)$
        and $hu$: $\forall P. \text{unitary } P \Rightarrow \text{unitary } (u P)$
shows le-utrans ($wlp$ (body ;; Embed $t \leftarrow G$ ⊕ Skip))
    ($wlp$ (body ;; Embed $u \leftarrow G$ ⊕ Skip))
⟨proof⟩

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

lemma lfp-loop-fp:
    assumes $hb$: healthy ($wp$ body)
        and $sP$: sound $P$
shows $\lambda s. \langle G \rangle s^* wp$ body ($\lambda s. \text{bound-of } P$) $s + \langle N G \rangle s^* P s \vdash \lambda s. \text{bound-of } P$
⟨proof⟩

lemma lfp-loop-greatest:
    fixes P::′s expect
    assumes $hb$: $\forall R. \lambda s. \langle G \rangle s^* wp$ body $R s + \langle N G \rangle s^* P s \vdash R \Rightarrow \text{sound } R$
    and $hb$: healthy ($wp$ body)
        and $sP$: sound $P$
        and $sQ$: sound $Q$
shows $Q \vdash \text{lfp-exp } (\lambda Q s. \langle G \rangle s^* wp$ body $Q s + \langle N G \rangle s^* P s)$
⟨proof⟩

lemma lfp-loop-sound:
    fixes P::′s expect
    assumes $hb$: healthy ($wp$ body)
        and $sP$: sound $P$
shows sound ($\text{lfp-exp } (\lambda Q s. \langle G \rangle s^* wp$ body $Q s + \langle N G \rangle s^* P s)$)
⟨proof⟩

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

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

lemma wp-loop-step-sound:
fixes \(t\) : \('s\) trans
assumes \(\text{hb}: \text{healthy} \; (\text{wp body})\)
and \(\text{ht}: \bigwedge P. \text{sound} \; P \rightarrow \text{sound} \; (t \; P)\)
and \(sP: \text{sound} \; P\)
shows 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\cite{McIver and Morgan, 2004, §7, p. 198, footnote 23}.

lemma wp-loop-loop-1:
fixes \(\text{body} : \;'s\) prog
assumes \(sP: \text{sound} \; P\)
and \(\text{hb}: \text{healthy} \; (\text{wp body})\)
shows wlp \((\text{do } G \rightarrow \text{body od} \; P) = \text{gfp-exp} (\lambda Q \; s. «G» \; s * \text{wp body} \; Q \; s + «N G» \; s * P \; s)\)
\((\text{is } ?X = \text{gfp-exp} (\forall Y \; P))\)
⟨proof⟩

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

Likewise, we can rewrite strict loops.

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

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

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

lemma healthy-wp-loop:
fixes \(\text{body} : \;'s\) prog
assumes $hb$: healthy ($wp$ body)
shows healthy ($wp$ (do $G$ $\longrightarrow$ 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 $\Rightarrow$ bool
where bd-cts $t$ = $(\forall M. (\forall i. (M i \vdash M (Suc i)) \land sound (M i)) \longrightarrow$
$(\exists b. \forall i. bounded-by b (M i)) \longrightarrow$
$t (\text{Sup-exp} (\text{range} M)) = \text{Sup-exp} (\text{range} (t o M))$

lemma bd-ctsD:
$[\text{bd-cts} \; t; \land i. M i \vdash M (Suc i); \land i. sound (M i); \land i. bounded-by b (M i)] \Longrightarrow$
$t (\text{Sup-exp} (\text{range} M)) = \text{Sup-exp} (\text{range} (t o M))$
(proof)

lemma bd-ctsI:
$(\land b. M. (\land i. M i \vdash M (Suc i)) \Rightarrow (\land i. sound (M i)) \Rightarrow (\land i. bounded-by b (M i)) \Rightarrow$
$t (\text{Sup-exp} (\text{range} M)) = \text{Sup-exp} (\text{range} (t o M))) \Rightarrow \text{bd-cts} \; t$
(proof)

A generalised property for transformers of transformers.

definition bd-cts-tr :: ($'$s trans $\Rightarrow$ $'$s trans) $\Rightarrow$ bool
where bd-cts-tr $T$ = $(\forall M. (\forall i. \text{le-trans} (M i) (M (Suc i)) \land \text{feasible} (M i)) \longrightarrow$
4.3. CONTINUITY

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

**Lemma bd-cts-trD:**

\[ \begin{array}{l}
\[ \text{bd-cts-tr } T; \land i. \text{le-trans } (M i) (M (\text{Suc } i)); \land i. \text{feasible } (M i) \] \implies \\
\text{equiv-trans } (T (\text{Sup-trans } (M \cdot UNIV))) (\text{Sup-trans } ((T \circ M) \cdot UNIV))
\end{array} \]

**Lemma bd-cts-trI:**

\[ \begin{array}{l}
(\land M. (\land i. \text{le-trans } (M i) (M (\text{Suc } i))) \implies (\land i. \text{feasible } (M i)) \implies \\
\text{equiv-trans } (T (\text{Sup-trans } (M \cdot UNIV))) (\text{Sup-trans } ((T \circ M) \cdot UNIV))
\end{array} \implies \text{bd-cts-tr } T \]

4.3.1 Continuity of Primitives

**Lemma cts-wp-Abort:**

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

**Lemma cts-wp-Skip:**

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

**Lemma cts-wp-Apply:**

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

**Lemma cts-wp-Bind:**

\[ \text{fixes } a::'a \Rightarrow 's \text{ prog} \]

\[ \text{assumes } ca: \land s. \text{bd-cts } (\wp (a \ f s)) \]

\[ \text{shows } \text{bd-cts } (\wp (\text{Bind } f a)) \]

The first nontrivial proof. We transform the suprema into limits, and appeal to the continuity of the underlying operation (here infimum). This is typical of the remainder of the nonrecursive elements.

**Lemma cts-wp-DC:**

\[ \text{fixes } a \ b::'s \text{ prog} \]

\[ \text{assumes } ca: \text{bd-cts } (\wp a) \]

\[ \text{and } cb: \text{bd-cts } (\wp b) \]

\[ \text{and } ha: \text{healthy } (\wp a) \]

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

\[ \text{shows } \text{bd-cts } (\wp (a \bigcap b)) \]

**Lemma cts-wp-Seq:**

\[ \text{fixes } a \ b::'s \text{ prog} \]

\[ \text{assumes } ca: \text{bd-cts } (\wp a) \]
and cb: bd-cts (wp b)
and hb: healthy (wp b)
shows bd-cts (wp (a ;; b))

⟨proof⟩

lemmas 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 (probab-}
listic choice can be extended infinitely, but we have not done so). The
proofs for both are inductive, and rely on the above results on binary oper-
ators.

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

lemmas 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⟩

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

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

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

lemmas 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: ∀x. x ∈ (supp p) ⇒ bd-cts (wp (a x))
and ha: ∀x. x ∈ (supp p) ⇒ healthy (wp (a x))
and up: unitary p
and sump: sum p (supp p) ≤ 1
and fsupp: finite (supp p)
shows bd-cts (wp (SetPC a (λ- p)))
⟨proof⟩

lemma cts-wp-SetPC:
fixes a::'a ⇒ 's prog
assumes ca: ∀x s. x ∈ (supp (p s)) ⇒ bd-cts (wp (a x))
and ha: ∀x s. x ∈ (supp (p s)) ⇒ healthy (wp (a x))
and up: ∀s. unitary (p s)
and sump: ∀s. sum (p s) (supp (p s)) ≤ 1
and fsupp: ∀s. finite (supp (p s))
shows bd-cts (wp (SetPC a p))
⟨proof⟩

lemma wp-SetDC-Bind:
SetDC a S = Bind S (λS. SetDC a (λ-. S))
⟨proof⟩

lemma SetDC-finite-insert:
assumes fS: finite S
and neS: S ≠ {}
shows SetDC a (λ-. insert x S) = a x ∩ SetDC a (λ-. S)
⟨proof⟩

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

lemma cts-wp-SetDC-const:
fixes a::'a ⇒ 's prog
assumes ca: ∀x. x ∈ S ⇒ bd-cts (wp (a x))
and ha: ∀x. x ∈ S ⇒ healthy (wp (a x))
and fs: finite S
and neS: S ≠ {}
shows bd-cts (wp (SetDC a (λ-. S)))
⟨proof⟩

lemma cts-wp-SetDC:
fixes a::'a ⇒ 's prog
assumes ca: ∀x s. x ∈ S s ⇒ bd-cts (wp (a x))
and ha: ∀x s. x ∈ S s ⇒ healthy (wp (a x))
and fs: ∀s. finite (S s)
and neS: S s ≠ {}

CHAPTER 4. THE PGCL LANGUAGE

shows bd-cts (wp (SetDC a S))
\langle proof \rangle

lemma cts-wp-repeat:
  bd-cts (wp a) \Rightarrow healthy (wp a) \Rightarrow bd-cts (wp (repeat n a))
\langle proof \rangle

lemma cts-wp-Embed:
  bd-cts t \Rightarrow bd-cts (wp (Embed t))
\langle proof \rangle

4.3.2 Continuity of a Single Loop Step

A single loop iteration is continuous, in the more general sense defined above for transformer transformers.

lemma cts-wp-loopstep:
  fixes body::’s prog
  assumes hb: healthy (wp body)
  and cb: bd-cts (wp body)
  shows bd-cts-tr (\lambda x. wp (body ; Embed x \oplus Skip)) (is bd-cts-tr ?F)
\langle proof \rangle

end

4.4 Continuity and Induction for Loops

theory LoopInduction imports Healthiness Continuity begin

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

A loop step is monotonic.

lemma wp-loop-step-mono-trans:
  fixes body::’s prog
  assumes sP: sound s
  and hb: healthy (wp body)
  shows mono-trans (\lambda Q s. \langle G s \oplus Skip \rangle) (\langle N s \rangle)
\langle proof \rangle

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

lemma lfp-wp-loop-unfold:
  fixes body::’s prog
  assumes hb: healthy (wp body)
  and sP: sound s
\langle proof \rangle
4.4. CONTINUITY AND INDUCTION FOR LOOPS

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

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

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

4.4.1 The Limit of Iterates

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

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

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

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

iterates body G 0 = (λP s. 0)

⟨proof⟩

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

iterates body G (Suc i) = wp (body ;; Embed (iterates body G i) « G ⊕ Skip)

⟨proof⟩

All iterates are healthy.

**lemma** iterates-healthy:

healthy (wp body) ⇒ healthy (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**

The iterates themselves are all continuous.

**Lemma** `cts-iterates`:
- **Finds** `body·`'s prog
- **Assumes** `hb`: healthy (`wp body`)
  - and `cb`: bd-cts (`wp body`)
- **Shows** `bd-cts (iterates body G i)`

Therefore so is the loop itself.

**Lemma** `cts-wp-loop`:
- **Finds** `body·`'s prog
- **Assumes** `hb`: healthy (`wp body`)
  - and `cb`: bd-cts (`wp body`)
- **Shows** `bd-cts (wp do G → body od)`

**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`

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)`

**Lemma** `sublinear-wp-Abort`:
sublinear (wp Abort)
⟨proof⟩

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

lemma sublinear-wp-Seq:
fixes x::'s prog
assumes slx: sublinear (wp x) and sly: sublinear (wp y)
and hx: healthy (wp x) and hy: healthy (wp y)
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 d y))
⟨proof⟩

As for continuity, we insist on a finite support.

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

lemma sublinear-wp-Embed:
sublinear t \Rightarrow sublinear (wp (Embed t))
⟨proof⟩
4.5. SUBLINEARITY

**Lemma sublinear-wp-repeat:**
\[
[\text{sublinear}(\text{wp } p); \text{healthy}(\text{wp } p)] \implies \text{sublinear}(\text{wp}(\text{repeat } n \ p))
\]
(proof)

**Lemma sublinear-wp-Bind:**
\[
[\forall s. \text{sublinear}(\text{wp}(a(f s)))] \implies \text{sublinear}(\text{wp}(\text{Bind } f \ a))
\]
(proof)

### 4.5.2 Sublinearity for Loops

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

**Lemma sub-distrib-wp-loop:**
fixes body::'s prog
assumes sdb: sub-distrib (wp body)
and hb: healthy (wp body)
and nhb: nearly-healthy (wlp body)
shows sub-distrib (wp (do G ::= body od))
(proof)

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

**Lemma sublinear-iterates:**
assumes hb: healthy (wp body)
and sb: sublinear (wp body)
shows sublinear (iterates body G i)
(proof)

From this, sub-additivity follows for the limit (i.e. the loop), by appealing to the property at all steps.

**Lemma sub-add-wp-loop:**
fixes body::'s prog
assumes sb: sublinear (wp body)
and cb: bd-cts (wp body)
and hwp: healthy (wp body)
shows sub-add (wp (do G ::= body od))
(proof)

**Lemma sublinear-wp-loop:**
fixes body::'s prog
assumes hb: healthy (wp body)
and nhb: nearly-healthy (wlp body)
and sb: sublinear (wp body)
and cb: bd-cts (wp body)
shows sublinear (wp (do G ::= body od))
(proof)

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

end

4.6 Determinism

theory Determinism imports WellDefined begin

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

4.6.1 Additivity

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

⟨proof⟩

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

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

lemma additive-wp-Seq:
  fixes a::'s prog
  assumes adda: additive (wp a)
  and addb: additive (wp b)
  and wb: well-def b
  shows additive (wp (a ;; b))
⟨proof⟩

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

DC is not additive.

**Lemma** additive-wp-SetPC:
\[ \left( \forall x. x \in \text{supp}(p) \Rightarrow \text{additive}(wp(a x)) \right) \land \text{finite}({\text{supp}(p)}) \implies \text{additive}(wp(\text{SetPC} a p)) \]

(proof)

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

(proof)

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

(proof)

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

(proof)

**Lemmas** fa-intros =
additive-wp-Abort additive-wp-Skip
additive-wp-Apply additive-wp-Seq
additive-wp-PC additive-wp-SetPC
additive-wp-Bind additive-wp-Embed
additive-wp-repeat

4.6.2 Maximality

**Lemma** max-wp-Skip:
\[ \text{maximal}(wp\;\text{Skip}) \]

(proof)

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

(proof)

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

(proof)

**Lemma** max-wp-PC:
\[ \left[ \text{maximal}(wp\;a); \text{maximal}(wp\;b) \right] \implies \text{maximal}(wp(a\oplus b)) \]

(proof)

**Lemma** max-wp-DC:
\[ \left[ \text{maximal}(wp\;a); \text{maximal}(wp\;b) \right] \implies \text{maximal}(wp(a\sqcap b)) \]

(proof)
lemma max-wp-SetPC:
\[
\begin{align*}
\text{fixes } p : 'a \Rightarrow 's \text{ prog} \\
\text{assumes } \wp : \text{maximal } (wp (p a)) \\
\text{and } \text{ne} : \text{maximal } (wp (\text{SetPC } p P)) \\
\end{align*}
\]
\[
\text{shows} \quad \text{maximal } (wp (\text{SetPC } p P))
\]
\langle proof \rangle

lemma max-wp-SetDC:
\[
\begin{align*}
\text{fixes } p : 'a \Rightarrow 's \text{ prog} \\
\text{assumes } \wp : \text{maximal } (wp (p a)) \\
\text{and } \text{ne} : \text{maximal } (wp (\text{SetDC } p S)) \\
\end{align*}
\]
\langle proof \rangle

lemma max-wp-Embed:
\[
\begin{align*}
\text{maximal } (wp a) \quad \text{maximal } (wp (\text{Embed } t)) \\
\end{align*}
\]
\langle proof \rangle

lemma max-wp-repeat:
\[
\begin{align*}
\text{maximal } (wp a) \quad \text{maximal } (wp (\text{repeat } n a)) \\
\end{align*}
\]
\langle proof \rangle

lemma max-wp-Bind:
\[
\begin{align*}
\text{assumes } ma : \text{maximal } (wp (a (f s))) \\
\text{shows} \quad \text{maximal } (wp (\text{Bind } f a)) \\
\end{align*}
\]
\langle proof \rangle

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:
\[
\begin{align*}
\text{assumes } ht : \text{healthy } t \\
\text{and } \text{trm} : \lambda s. 1 \vdash t (\lambda s. 1) \\
\text{shows} \quad \text{maximal } t \\
\end{align*}
\]
\langle proof \rangle

4.6.3 Determinism

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

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

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

**lemma** det-wp-PC:
\[
determ (wp a) \implies determ (wp b) \implies determ (wp (a \oplus b))
\]
\(<proof>\)

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

**lemma** det-wp-Bind:
\[
(\forall x. determ (wp (a (f x)))) \implies determ (wp (Bind f a))
\]
\(<proof>\)

**lemma** det-wp-Embed:
\[
determ t \implies determ (wp (Embed t))
\]
\(<proof>\)

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

**lemmas** determ-intros =
- det-wp-Skip det-wp-Apply
- det-wp-Seq det-wp-PC
- det-wp-SetPC det-wp-Bind
- det-wp-Embed det-wp-repeat

end

4.7 Well-Defined Programs.

theory WellDefined imports
  Healthiness
  Sublinearity
  LoopInduction
begin

The definition of a well-defined program collects the various notions of healthiness and well-behavedness that we have so far established: healthiness of the strict and liberal transformers, continuity and sublinearity of the strict transformers, and two new properties. These are that the strict transformer always lies below the liberal one (i.e. that it is at least as strict,
recalling the standard embedding of a predicate), and that expectation con-
junction is distributed between then 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 interpre-
tations ($wp$ and $wlp$).

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

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

lemma \( \text{wp-under-wlpI} \) [intro]:
$$[ \forall P. \text{unitary } P \Rightarrow wp \text{ prog } P \vdash wlp \text{ prog } P ] \Rightarrow \text{wp-under-wlp prog}$$

⟨proof⟩

lemma \( \text{wp-under-wlpD} \) [dest]:
$$[ \text{wp-under-wlp prog}; \text{unitary } P ] \Rightarrow wp \text{ prog } P \vdash wlp \text{ prog } P$$

⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

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

⟨proof⟩

lemma \( \text{wp-under-wlp-Seq} \):
assumes \( h-wlp-a \): nearly-healthy \((wlp \ a)\)
and \( h-wp-b \): healthy \((wp \ b)\)
and \( h-wlp-b \): nearly-healthy \((wlp \ b)\)
and \( wp-u-a \): \((wp-under-wlp \ a)\)
and \( wp-u-b \): \((wp-under-wlp \ b)\)
shows \((wp-under-wlp \ (a \;; \ b))\)

⟨proof⟩

lemma \( \text{wp-under-wlp-PC} \):
assumes \( h-wp-a \): healthy \((wp \ a)\)
4.7. WELL-DEFINED PROGRAMS.

and h-wlp-a: nearly-healthy (wlp a)
and h-wp-b: healthy (wp b)
and h-wlp-b: nearly-healthy (wlp b)
and wp-a-a: wp-under-wlp a
and wp-a-b: wp-under-wlp b
and uP: unitary P
shows wp-under-wlp (a p ⊕ b)

⟨proof⟩

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

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

lemma wp-under-wlp-SetDC:
  assumes wp-u-f: \( \forall s \ a \ a \in S s \Rightarrow wp-under-wlp (f a) \)
          and hf: \( \forall s \ a \ a \in S s \Rightarrow healthy (wp (f a)) \)
          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 ] →
  wp-under-wlp (repeat n a)
⟨proof⟩

lemma wp-under-wlp-Bind:
  [ \( \forall s . \ wp-under-wlp (a (f s)) \) ] → 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

\textnormal{sub-distrib-pconj} :: 's prog \Rightarrow \text{bool}

where

\textnormal{sub-distrib-pconj} \text{ prog} \equiv

\forall P\ Q. \text{unitary } P \implies \text{unitary } Q \implies

\text{wlp } \text{ prog} P \&\& \text{ wp } \text{ prog} Q \implies \text{ wp } \text{ prog} (P \&\& Q)

lemma \textnormal{sub-distrib-pconjI} [\text{intro}]:

\forall P\ Q. [ \text{unitary } P; \text{unitary } Q ] \Rightarrow \text{wlp } \text{ prog} P \&\& \text{ wp } \text{ prog} Q \implies \text{ wp } \text{ prog} (P \&\& Q)

⟨\text{proof}⟩

lemma \textnormal{sub-distrib-pconjD} [\text{dest}]:

\forall P\ Q. [ \text{sub-distrib-pconj } \text{ prog}; \text{unitary } P; \text{unitary } Q ] \Rightarrow \text{wlp } \text{ prog} P \&\& \text{ wp } \text{ prog} Q \implies \text{ wp } \text{ prog} (P \&\& Q)

⟨\text{proof}⟩

lemma \textnormal{sdp-Abort}:

\text{sub-distrib-pconj } \text{ Abort}

⟨\text{proof}⟩

lemma \textnormal{sdp-Skip}:

\text{sub-distrib-pconj } \text{ Skip}

⟨\text{proof}⟩

lemma \textnormal{sdp-Seq}:

\text{fixes } a \text{ and } b

\text{assumes } \text{sdp-a: } \text{sub-distrib-pconj } a

\text{and } \text{sdp-b: } \text{sub-distrib-pconj } b

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

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

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

\text{shows } \text{sub-distrib-pconj } (a ;; b)

⟨\text{proof}⟩

lemma \textnormal{sdp-Apply}:

\text{sub-distrib-pconj } (\text{Apply } f)

⟨\text{proof}⟩
4.7. WELL-DEFINED PROGRAMS.

**Lemma sdp-DC:**

**Fixes** $a :: s \text{ prog and } b$

**Assumes**
- $sdp-a$: sub-distrib-pconj $a$
- $sdp-b$: sub-distrib-pconj $b$
- $h-wp-a$: healthy $(wp a)$
- $h-wp-b$: healthy $(wp b)$
- $h-wlp-b$: nearly-healthy $(wlp b)$

**Shows** sub-distrib-pconj $(a \sqcup b)$

(Proof)

**Lemma sdp-PC:**

**Fixes** $a :: s \text{ prog and } b$

**Assumes**
- $sdp-a$: sub-distrib-pconj $a$
- $sdp-b$: sub-distrib-pconj $b$
- $h-wp-a$: healthy $(wp a)$
- $h-wp-b$: healthy $(wp b)$
- $uP$: unitary $P$

**Shows** sub-distrib-pconj $(a P \oplus b)$

(Proof)

**Lemma sdp-Embed:**

\[
\left[ \left[ \forall P Q. \left[ \forall \text{ unitary } P \right] \Rightarrow t P \land t Q \Rightarrow \left[ P \land Q \right] \right] \Rightarrow \text{sub-distrib-pconj} \left( \text{Embed } t \right) \right]
\]

(Proof)

**Lemma sdp-repeat:**

**Fixes** $a :: s \text{ prog}$

**Assumes**
- $sdp-a$: sub-distrib-pconj $a$
- $hwp$: healthy $(wp a)$ and $hwlp$: nearly-healthy $(wlp a)$

**Shows** sub-distrib-pconj $(\text{repeat } n a) \ (\text{is } ?X n)$

(Proof)

**Lemma sdp-SetPC:**

**Fixes** $p :: a \Rightarrow s \text{ prog}$

**Assumes**
- $sdp$: $\forall s. a \in \text{supp} (P s) \Rightarrow \text{sub-distrib-pconj} (p a)$
- $fin$: $\forall s. \text{finite} (\text{supp} (P s))$
- $nnp$: $\forall s. a. 0 \leq P s a$
- $sub$: $\forall s. \text{sum} (P s) (\text{supp} (P s)) \leq 1$

**Shows** sub-distrib-pconj $(\text{SetPC } p P)$

(Proof)

**Lemma sdp-SetDC:**

**Fixes** $p :: a \Rightarrow s \text{ prog}$

**Assumes**
- $sdp$: $\forall s. a \in S s \Rightarrow \text{sub-distrib-pconj} (p a)$
- $hwp$: $\forall s. a \in S s \Rightarrow \text{healthy} (wp (p a))$
- $hwlp$: $\forall s. a \in S s \Rightarrow \text{nearly-healthy} (wlp (p a))$
- $nc$: $\forall s. S s \neq \{\}$
shows sub-distrib-pconj (SetDC p S)
(proof)

lemma sdp-Bind:
[ \[ s, \text{sub-distrib-pconj} (p (f s)) \] \] \implies \text{sub-distrib-pconj} (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 \rightarrow 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 \Rightarrow bool
where
well-def prog \equiv \text{healthy} (wp prog) \land \text{nearly-healthy} (wlp prog)
\land \text{wp-under-wlp} prog \land \text{sub-distrib-pconj} prog
\land \text{sublinear} (wp prog) \land \text{bd-cts} (wp prog)

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

lemma well-def-wp-healthy[dest]:
well-def prog \implies \text{healthy} (wp prog)
(proof)

lemma well-def-wlp-nearly-healthy[dest]:
well-def prog \implies \text{nearly-healthy} (wlp prog)
(proof)

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

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

\textbf{lemma well-def-sdp\{dest\}:}
\[
\text{well-def prog} \implies \text{sub-distrib-pconj prog}
\]
\[\langle \text{proof} \rangle\]

\textbf{lemma well-def-wp-sublinear\{dest\}:}
\[
\text{well-def prog} \implies \text{sublinear (wp prog)}
\]
\[\langle \text{proof} \rangle\]

\textbf{lemma well-def-wp-cts\{dest\}:}
\[
\text{well-def prog} \implies \text{bd-cts (wp prog)}
\]
\[\langle \text{proof} \rangle\]

\textbf{lemmas wd-dests =}
\[
\text{well-def-wp-healthy well-def-wlp-nearly-healthy}
\]
\[
\text{well-def-wp-under well-def-sdp}
\]
\[
\text{well-def-wp-sublinear well-def-wp-cts}
\]

\textbf{lemma wd-Abort:}
\[
\text{well-def Abort}
\]
\[\langle \text{proof} \rangle\]

\textbf{lemma wd-Skip:}
\[
\text{well-def Skip}
\]
\[\langle \text{proof} \rangle\]

\textbf{lemma wd-Apply:}
\[
\text{well-def (Apply f)}
\]
\[\langle \text{proof} \rangle\]

\textbf{lemma wd-Seq:}
\[
[ \text{well-def a; well-def b} ] \implies \text{well-def (a ;; b)}
\]
\[\langle \text{proof} \rangle\]

\textbf{lemma wd-PC:}
\[
[ \text{well-def a; well-def b; unitary P} ] \implies \text{well-def (a p□ b)}
\]
\[\langle \text{proof} \rangle\]

\textbf{lemma wd-DC:}
\[
[ \text{well-def a; well-def b} ] \implies \text{well-def (a ▼ b)}
\]
\[\langle \text{proof} \rangle\]

\textbf{lemma wd-SetDC:}
\[
[ \forall x s. x \in S s \implies \text{well-def (a x)}; \forall s. S s \neq \emptyset; \forall s. \text{finite (S s)} ] \implies \text{well-def (SetDC a S)}
\]
\[\langle \text{proof} \rangle\]
**Lemma** *wd-SetPC*:

\[
\begin{align*}
\forall x \in (\text{supp} (p s)) & \implies \text{well-def} (a x) ;
\forall s. \text{unitary} (p s) ;
\forall s. \text{finite} (\text{supp} (p s)) ;
\forall s. \text{sum} (p s) (\text{supp} (p s)) \leq 1 & \implies \text{well-def} (\text{SetPC} a p)
\end{align*}
\]

**Lemma** *wd-Embed*:

**Fixes** \(t::'s\) trans

**Assumes** *ht: healthy t and st: sublinear t and ct: bd-cts t

**Shows** \(\text{well-def} (\text{Embed} t)\)

**Lemma** *wd-repeat*:

\(\text{well-def} a \implies \text{well-def} (\text{repeat} n a)\)

**Lemma** *wd-Bind*:

\[
\begin{align*}
\forall s. \text{well-def} (a (f s)) & \implies \text{well-def} (\text{Bind} f a)
\end{align*}
\]

**Lemma** *wd-loop*:

\(\text{well-def body} \implies \text{well-def} (\text{do} G \rightarrow \text{body} od)\)

**Lemmas** *wd-intros =

*wd-Abort wd-Skip wd-Apply
wd-Embed wd-Seq wd-PC
wd-DC wd-SetPC wd-SetDC
wd-Bind wd-repeat wd-loop

**End**

### 4.8 The Loop Rules

**Theory** *Loops imports WellDefined begin*

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

#### 4.8.1 Liberal and Strict Invariants.

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

**Definition**

\(\text{wp-inv} :: (s \rightarrow \text{bool}) \rightarrow (s \rightarrow \text{real}) \rightarrow \text{bool} \)

**Where**

\(\text{wp-inv} G \text{ body} I \iff (\forall s. \langle G \rangle s * I s \leq \text{wp body} I s)\)
4.8. THE LOOP RULES

**Lemma** $wp$-$invI$:

$$\forall I. (\forall s. \langle G \rangle_s \ast I_s \leq wp\ body\ I_s) \implies wp\-inv\ G\ body\ I$$

**Definition**

**wlp-inv** :: $\langle s \Rightarrow bool \rangle \Rightarrow \langle s \Rightarrow real \rangle \Rightarrow bool$

**Where**

$$wlp-inv\ G\ body\ I \iff (\forall s. \langle G \rangle_s \ast I_s \leq wlp\ body\ I_s)$$

**Lemma** $wlp$-$invI$:

$$\forall I. (\forall s. \langle G \rangle_s \ast I_s \leq wlp\ body\ I_s) \implies wlp-inv\ G\ body\ I$$

**Lemma** $wlp$-$invD$:

$$wlp-inv\ G\ body\ I \implies \langle N G \rangle_s \ast I_s \leq wlp\ body\ I_s$$

For standard invariants, the multiplication reduces to conjunction.

**Lemma** $wp$-$inv$-$stdD$:

- **Assumes** $inv$: $wp$-$inv\ G\ body\ I$
- **And** $hb$: healthy ($wp$ $body$)
- **Shows** $\langle G \rangle \& \& \langle I \rangle \vdash wp\ body\ I$

**4.8.2 Partial Correctness**


**Lemma** $wlp$-$Loop$:

- **Assumes** $wd$: $well-def\ body$
- **And** $uI$: unitary $I$
- **And** $inv$: $wlp$-$inv\ G\ body\ I$
- **Shows** $I \leq wlp\ do\ G \longrightarrow body\ od\ (\lambda s. \langle N G \rangle_s \ast I_s)$$\ (is\ I \leq wlp\ do\ G \longrightarrow body\ od\ ?P)$

**4.8.3 Total Correctness**

The first total correctness lemma for loops which terminate with probability 1[McIver and Morgan, 2004, Lemma 7.3.1, §7, p. 186].

**Lemma** $wp$-$Loop$:

- **Assumes** $wd$: $well-def\ body$
- **And** $inv$: $wlp$-$inv\ G\ body\ I$
- **And** $unit$: unitary $I$
- **Shows** $I \& \& wp\ (do\ G \longrightarrow body\ od)\ (\lambda s. 1) \vdash wp\ (do\ G \longrightarrow body\ od)\ (\lambda s. \langle N G \rangle_s \ast I_s)$$\ (is\ I \& \& ?T \vdash wp\ ?loop\ ?X)$
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)

(\textit{proof})

**lemma** wp-loop-nguard:

\[ \text{healthy (wp body); sound P; } \neg G s \implies wp \text{ do } G \rightarrow \text{body od } P s = P s \]

(\textit{proof})

**lemma** wp-loop-guard:

\[ \text{healthy (wp body); sound P; } G s \implies wp \text{ do } G \rightarrow \text{body od } P s = wp \text{ (body ;; do } G \rightarrow \text{body od} \text{) } P s \]

(\textit{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 \(a \sqcap b\) and \(a \sqcup b\) as the meet and join operators, respectively. We also take advantage of the algebraic structure to establish a framework for the modular decomposition of proofs.

4.9.1 Program Refinement

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

**definition**

\[ \text{refines :: 's prog } \Rightarrow \text{'s prog } \Rightarrow \text{ bool (infix } \sqsubseteq \text{ 70) } \]

**where**

\[ \text{prog } \sqsubseteq \text{ prog'} \equiv \forall P. \text{ sound P } \rightarrow \text{ wp prog P } \vdash \text{ wp prog'} P \]

**lemma** refines [intro]:

\[ \forall P. \text{ sound P } \rightarrow \text{ wp prog P } \vdash \text{ wp prog'} P \Rightarrow \text{ prog } \sqsubseteq \text{ prog'} \]

(\textit{proof})
4.9. THE ALGEBRA OF PGCL

**lemma** refinesD[dest]:
\[[ \text{prog} \sqsubseteq \text{prog}'; \text{sound} P ] \implies \text{wp prog} P \models \text{wp prog}' P\]

(\textit{proof})

The equivalence relation below will turn out to be that induced by refinement. It is also the application of \textit{equiv-trans} to the weakest precondition.

**definition**
\textit{pequiv} :: 's prog \Rightarrow 's prog \Rightarrow bool (\textit{infix} \simeq 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]:
\[[ \forall P. \text{sound} P \implies \text{wp prog} P = \text{wp prog}' P ] \implies \text{prog} \simeq \text{prog}'

(\textit{proof})

**lemma** pequivD[dest,simp]:
\[[ \text{prog} \simeq \text{prog}'; \text{sound} P ] \implies \text{wp prog} P = \text{wp prog}' P

(\textit{proof})

**lemma** pequiv-equiv-trans:
a \simeq b \iff \text{equiv-trans} (\text{wp a}) (\text{wp b})

(\textit{proof})

4.9.2 Simple Identities

The following identities involve only the primitive operations as defined in Section 4.1.1, and refinement as defined above.

**Laws following from the basic arithmetic of the operators separately**

**lemma** DC-comm[ac-simps]:
a \sqcap b = b \sqcap a

(\textit{proof})

**lemma** DC-assoc[ac-simps]:
a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c

(\textit{proof})

**lemma** DC-idem:
a \sqcap a = a

(\textit{proof})

**lemma** AC-comm[ac-simps]:
a \sqcup b = b \sqcup a

(\textit{proof})

**lemma** AC-assoc[ac-simps]:
\[ a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c \]

**Lemma AC-idem:**
\[ a \sqcup a = a \]

**Lemma PC-quasi-comm:**
\[ a \mathbin{\mathrel{\sqcap}} b = b (\lambda s. 1 - p \, s) \mathbin{\mathrel{\sqcap}} a \]

**Lemma PC-idem:**
\[ a \mathbin{\mathrel{\sqcap}} a = a \]

**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:**
\[ (a \sqcap b) \subseteq a \]

**Lemma right-refines-DC:**
\[ (a \sqcap b) \subseteq b \]

**Lemma DC-refines:**
\[ \text{fixes } a::'s \text{ prog and } b \text{ and } c \]
\[ \text{assumes } rab: a \subseteq b \text{ and } rac: a \subseteq c \]
\[ \text{shows } a \subseteq (b \sqcap c) \]

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

**Laws relating angelic choice and refinement**

**Lemma left-refines-AC:**
\[ a \subseteq (a \sqcup b) \]
4.9. THE ALGEBRA OF PGCL

**lemma** right-refines-AC:
\[ b \sqsubseteq (a \sqcup b) \]
⟨proof⟩

**lemma** AC-refines:
\begin{align*}
\text{fixes } & a::'s \text{ prog and } b \text{ and } c \\
\text{assumes } & rac: a \subseteq c \text{ and } rbc: b \subseteq c \\
\text{shows } & (a \sqcup b) \subseteq c
\end{align*}
⟨proof⟩

**lemma** AC-mono:
\begin{align*}
\text{fixes } & a::'s \text{ prog} \\
\text{assumes } & rab: a \sqsubseteq b \text{ and } rcd: c \sqsubseteq d \\
\text{shows } & (a \sqcup c) \subseteq (b \sqcup d)
\end{align*}
⟨proof⟩

Laws depending on the arithmetic of \( a \odot b \) and \( a \cap b \) together

**lemma** DC-refines-PC:
\begin{align*}
\text{assumes } & \text{unit: unitary } p \\
\text{shows } & (a \cap b) \subseteq (a \odot b)
\end{align*}
⟨proof⟩

Laws depending on the arithmetic of \( a \odo b \) and \( a \cup b \) together

**lemma** PC-refines-AC:
\begin{align*}
\text{assumes } & \text{unit: unitary } p \\
\text{shows } & (a \odo b) \subseteq (a \cup b)
\end{align*}
⟨proof⟩

Laws depending on the arithmetic of \( a \cup b \) and \( a \cap b \) together

**lemma** DC-refines-AC:
\[ (a \cap b) \subseteq (a \cup b) \]
⟨proof⟩

**lemma** pr-trans[trans]:
\begin{align*}
\text{fixes } & A::'a \text{ prog} \\
\text{assumes } & \text{prAB: } A \subseteq B \\
& \text{prBC: } B \subseteq C \\
\text{shows } & A \subseteq C
\end{align*}
⟨proof⟩

**lemma** pequiv-refl[intro,simp]:
\[ a \equiv a \]
⟨proof⟩
chapter 4. the pgcl language

lemma pequiv-comm[ac-simps]:
\[ a \simeq b \iff b \simeq a \]
⟨proof⟩

lemma pequiv-pr[dest]:
\[ a \simeq b \implies a \sqsubseteq b \]
⟨proof⟩

lemma pequiv-trans[intro,trans]:
\[ [ a \simeq b ; b \simeq c ] \implies a \simeq c \]
⟨proof⟩

lemma pequiv-pr-trans[intro,trans]:
\[ [ a \simeq b ; b \sqsubseteq c ] \implies a \sqsubseteq c \]
⟨proof⟩

lemma pr-pequiv-trans[intro,trans]:
\[ [ a \sqsubseteq b ; b \simeq c ] \implies a \simeq c \]
⟨proof⟩

Refinement induces equivalence by antisymmetry:

lemma pequiv-antisym:
\[ [ a \sqsubseteq b ; b \sqsubseteq a ] \implies a \simeq b \]
⟨proof⟩

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

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

4.9.3 Deterministic Programs are Maximal

Any sub-additive refinement of a deterministic program is in fact an equivalence. Deterministic programs are thus maximal (under the refinement order) among sub-additive programs.

lemma refines-determ:
fixes a::'s prog
assumes da: determ (wp a)
and wa: well-def a
and wb: well-def b
and dr: a \sqsubseteq b
shows a \simeq b
⟨proof⟩

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

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

\texttt{quotient-type \textqt{\prime s program} =}
\[
\textqt{\prime s prog / partial} : \Lambda a b.\ a \simeq b \land \text{well-def} a \land \text{well-def} b
\]

\texttt{instantiation program :: (type) semilattice-inf begin}

\texttt{lift-definition}
\[
\texttt{less-eq-program :: \textqt{\prime a program} \Rightarrow \textqt{\prime a program} \Rightarrow \textit{bool} is refines}
\]

\texttt{proof}

\texttt{lift-definition}
\[
\texttt{less-program :: \textqt{\prime a program} \Rightarrow \textqt{\prime a program} \Rightarrow \textit{bool} is}
\]
\[
\Lambda a b.\ a \subseteq b \land \neg b \subseteq a
\]

\texttt{proof}

\texttt{lift-definition}
\[
\texttt{inf-program :: \textqt{\prime a program} \Rightarrow \textqt{\prime a program} \Rightarrow \textqt{\prime a program} is DC}
\]

\texttt{proof}

\texttt{instance}

\texttt{proof}

end

\texttt{instantiation program :: (type) bot begin}

\texttt{lift-definition}
\[
\texttt{bot-program :: \textqt{\prime a program} is Abort}
\]

\texttt{proof}

\texttt{instance \texttt{proof}}

\texttt{end}

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

\texttt{lift-definition}
\[
\texttt{pdeterm :: \textqt{\prime s program} \Rightarrow \textit{bool} is}
\]
\[
\Lambda a.\ \text{determ} (wp a)
\]

\texttt{proof}

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

\texttt{proof}
4.9.5 Data Refinement

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

definition
drefines :: ('b ⇒ 'a) ⇒ ('b ⇒ bool) ⇒ 'a prog ⇒ 'b prog ⇒ bool

where
drefines \( \varphi \) G A B ≡ ∀ P Q. (unitary P ∧ unitary Q ∧ (P ⊢ wp A Q)) →→ («G» && (P o \( \varphi \)) ⊢ wp B (Q o \( \varphi \)))

lemma drefinesD[dest]:
[ drefines \( \varphi \) G A B; unitary P; unitary Q; P ⊢ wp A Q ] →→ «G» && (P o \( \varphi \)) ⊢ wp B (Q o \( \varphi \))
⟨proof⟩

We can alternatively use G as an assumption:

lemma drefinesD2:
assumes dr: drefines \( \varphi \) G A B
and uP: unitary P
and uQ: unitary Q
and wpA: P ⊢ wp A Q
and G: G s
shows (P o \( \varphi \)) s ≤ wp B (Q o \( \varphi \)) s
⟨proof⟩

This additional form is sometimes useful:

lemma drefinesD3:
assumes dr: drefines \( \varphi \) G a b
and G: G s
and uQ: unitary Q
and wa: well-def a
shows wp a Q (\( \varphi \) s) ≤ wp b (Q o \( \varphi \)) s
⟨proof⟩

lemma drefinesI[intro]:
[ \( \varphi \) G A B; unitary P; unitary Q; P ⊢ wp A Q ] →→ «G» && (P o \( \varphi \)) ⊢ wp B (Q o \( \varphi \))
⟨proof⟩

Use G as an assumption, when showing refinement:

lemma drefinesI2:
fixes A::'a prog
and B::'b prog
and \( \varphi ::'b ⇒ 'a \)
and G::'b ⇒ bool
assumes wB: well-def B
and withAs:
\[ P \ Q \ s. \ [ unitary P; \ unitary Q; \ G s; \ P \vdash \ wp A Q ] \implies (P \circ \varphi) s \leq wp B (Q \circ \varphi) s \]
shows drefines \( \varphi \) G A B

(proof)

lemma dr-strengthen-guard:
fixes a::'s prog and b::'t prog
assumes fg: \( \forall Q. \ unitary Q \implies \{ G \} && (wp A Q \circ \varphi) = \{ G \} && wp B (Q \circ \varphi) \)

shows drefines \( \varphi \) F a b

(proof)

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

lemma pcorresI2[intro]:
fixes A::'a prog and B::'b prog
assumes withG: \( \forall Q. \ unitary Q \implies \{ G \} && (wp A Q o \varphi) = \{ G \} && wp B (Q o \varphi) \)

and wA: well-def A
and wB: well-def B
shows pcorres \( \varphi \) G A B

(proof)

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

lemma pcorresD2:
assumes pc: pcorres \( \varphi \) G A B
and uQ: unitary Q
and wA: well-def A and wB: well-def B
and G: G s
shows wp A Q (\( \varphi \) s) = wp B (Q o \( \varphi \) s)

(proof)
4.9.6 The Algebra of Data Refinement

Program refinement implies a trivial data refinement:

**lemma** refines-drefines:

```plaintext
fixes a::'s prog
assumes rab: a ⊑ b and wb: well-def b
shows drefines (λs. s) G a b
```

Data refinement is transitive:

**lemma** dr-trans[trans]:

```plaintext
fixes A::'a prog and B::'b prog and C::'c prog
assumes drAB: drefines ϕ G A B
and drBC: drefines ϕ' G' B C
and Gimp: \( s \rightarrow G (ϕ' s) \)
shows drefines (ϕ o ϕ') G' A C
```

Data refinement composes with program refinement:

**lemma** pr-dr-trans[trans]:

```plaintext
assumes prAB: A ⊑ B
and drBC: drefines ϕ G B C
shows drefines ϕ G A C
```

**lemma** dr-pr-trans[trans]:

```plaintext
assumes drAB: drefines ϕ G A B
assumes prBC: B ⊑ C
shows drefines ϕ G A C
```

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

**lemma** dr-refl:

```plaintext
assumes wa: well-def a
and comm: \( Q. \) unitary Q \( \rightarrow wp a Q o ϕ \vdash wp a (Q o ϕ) \)
shows drefines ϕ G a a
```

Correspondence implies data refinement

**lemma** pcorres-drefine:

```plaintext
assumes corres: pcorres ϕ G A C
and wc: well-def C
shows drefines ϕ G A C
```

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` `ϕ` `G` `a` `b`

**shows** `pcorres` `ϕ` `G` `a` `b`

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

⟨proof⟩

4.9.7 Structural Rules for Correspondence

**lemma** `pcorres-Skip`:

`pcorres` `ϕ` `G` `Skip` `Skip`

⟨proof⟩

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 `ϕ::'c ⇒ 'b`

**assumes** `pcAB`: `pcorres` `ϕ` `G` `A` `B`

and `pcCD`: `pcorres` `ϕ` `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`: `〈Q. unitary Q ⇒ «I» && wp B Q = wp B («H» && Q)〉`

and `p1p3`: `〈s. G s ⇒ I s〉`

**shows** `pcorres` `ϕ` `G` `(A;;C)` `(B;;D)`

⟨proof⟩

4.9.8 Structural Rules for Data Refinement

**lemma** `dr-Skip`:

**fixes** `ϕ::'c ⇒ 'b`

**shows** `drefines` `ϕ` `G` `Skip` `Skip`

⟨proof⟩

**lemma** `dr-Abort`:

**fixes** `ϕ::'c ⇒ 'b`

**shows** `drefines` `ϕ` `G` `Abort` `Abort`

⟨proof⟩

**lemma** `dr-Apply`:

**fixes** `ϕ::'c ⇒ 'b`

**assumes** `commutes`: `f o ϕ = ϕ o g`

**shows** `drefines` `ϕ` `G` `(Apply f)` `(Apply g)`
CHAPTER 4. THE PGCL LANGUAGE

lemma \textit{dr-Seq}:
\begin{align*}
\text{assumes } & \text{drAB: drefines } \varphi \ P \ A \ B \\
\text{and } & \text{drBC: drefines } \varphi \ Q \ C \ D \\
\text{and } & \text{wpB: } «P» \vdash \text{wp } B \ «Q» \\
\text{and } & \text{wB: well-def } B \\
\text{and } & \text{wC: well-def } C \\
\text{and } & \text{wD: well-def } D \\
\text{shows } & \text{drefines } \varphi \ P (A; C) (B; D)
\end{align*}
\langle \text{proof} \rangle

lemma \textit{dr-repeat}:
\begin{align*}
\text{fixes } & \varphi :: \ 'a \Rightarrow \ 'b \\
\text{assumes } & \text{dr-ab: drefines } \varphi \ G \ a \ b \\
\text{and } & \text{Gpr: } «G» \vdash \text{wp } b \ «G» \\
\text{and } & \text{wa: well-def } a \\
\text{and } & \text{wb: well-def } b \\
\text{shows } & \text{drefines } \varphi \ G \ (\text{repeat } n \ a) (\text{repeat } n \ b) (\text{is } ?X n)
\end{align*}
\langle \text{proof} \rangle

end

4.10 Structured Reasoning

theory \textit{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. \ 0) & \vdash \text{wp } \text{Abort } Q
\end{align*}
\langle \text{proof} \rangle

lemma \textit{wlp-Abort}:
\begin{align*}
(\lambda s. \ 1) & \vdash \text{wlp } \text{Abort } Q
\end{align*}
\langle \text{proof} \rangle

lemma \textit{wp-Skip}:
\begin{align*}
P & \vdash \text{wp } \text{Skip } P
\end{align*}
\langle \text{proof} \rangle

lemma \textit{wlp-Skip}:
\begin{align*}
P & \vdash \text{wlp } \text{Skip } P
\end{align*}
4.10. STRUCTURED REASONING

(⟨proof⟩)

lemma wp-Apply:
\[ Q \circ f \vdash wp \left( \text{Apply } f \right) Q \]
⟨⟨proof⟩⟩

lemma wlp-Apply:
\[ Q \circ f \vdash wlp \left( \text{Apply } f \right) Q \]
⟨⟨proof⟩⟩

lemma wp-Seq:
assumes ent-a: \( P \vdash wp a Q \)
and ent-b: \( Q \vdash wp b R \)
and wa: well-def a
and wb: well-def b
and s-Q: sound Q
and s-R: sound R
shows \( P \vdash wp \left( a \;;; b \right) R \)
⟨⟨proof⟩⟩

lemma wlp-Seq:
assumes ent-a: \( P \vdash wlp a Q \)
and ent-b: \( Q \vdash wlp b R \)
and wa: well-def a
and wb: well-def b
and u-Q: unitary Q
and u-R: unitary R
shows \( P \vdash wlp \left( a \;;; b \right) R \)
⟨⟨proof⟩⟩

lemma wp-PC:
\[ (\lambda s. P s \ast wp a Q s + (1 - P s) \ast wp b Q s) \vdash wp \left( a \oplus b \right) Q \]
⟨⟨proof⟩⟩

lemma wlp-PC:
\[ (\lambda s. P s \ast wlp a Q s + (1 - P s) \ast wlp b Q s) \vdash wlp \left( a \oplus b \right) Q \]
⟨⟨proof⟩⟩

A simpler rule for when the probability does not depend on the state.

lemma PC-fixed:
assumes wp-a: \( P \vdash a ab R \)
and wp-b: \( Q \vdash b ab R \)
and np: \( 0 \leq p \) and bp: \( p \leq 1 \)
shows \( (\lambda s. p \ast P s + (1 - p) \ast Q s) \vdash (a \left( \lambda s. p \right) \oplus b) ab R \)
⟨⟨proof⟩⟩

lemma wp-PC-fixed:
\[ [ P \vdash wp a R; Q \vdash wp b R; 0 \leq p; p \leq 1 ] \implies 
(\lambda s. p \ast P s + (1 - p) \ast Q s) \vdash wp \left( a \left( \lambda s. p \right) \oplus b \right) R \]
\[ \text{lemma wlp-PC-fixed:} \]
\[
\begin{align*}
\{ P \vdash \text{wlp } a \ R; \quad Q \vdash \text{wlp } b \ R; \quad 0 \leq p; \quad p \leq 1 \} \implies \\
(\lambda s. p * P s + (1 - p) * Q s) \vdash \text{wlp } (a \langle \lambda s. p \rangle \oplus b) \ R
\end{align*}
\]
\[
\langle \text{proof} \rangle
\]

\[ \text{lemma wp-DC:} \]
\[
(\lambda s. \text{min } (\text{wp } a \ Q s) \ (\text{wp } b \ Q s)) \vdash \text{wp } (a \sqcap b) \ Q
\]
\[
\langle \text{proof} \rangle
\]

\[ \text{lemma wlp-DC:} \]
\[
(\lambda s. \text{min } (\text{wlp } a \ Q s) \ (\text{wlp } b \ Q s)) \vdash \text{wlp } (a \sqcap b) \ Q
\]
\[
\langle \text{proof} \rangle
\]

Combining annotations for both branches:

\[ \text{lemma DC-split:} \]
\[
\begin{align*}
\text{fixes } a:: \text{'s prog} \text{ and } b \\
\text{assumes wpa: } P \vdash a \ ab \ R \\
\text{and wpb: } Q \vdash b \ ab \ R \\
\text{shows } (\lambda s. \text{min } (P s) \ (Q s)) \vdash (a \sqcap b) \ ab \ R
\end{align*}
\]
\[
\langle \text{proof} \rangle
\]

\[ \text{lemma wp-DC-split:} \]
\[
\begin{align*}
\{ P \vdash \text{wp } \text{prog } R; \quad Q \vdash \text{wp } \text{prog}' R \} \implies \\
(\lambda s. \text{min } (P s) \ (Q s)) \vdash \text{wp } (\text{prog } \sqcap \text{prog}') \ R
\end{align*}
\]
\[
\langle \text{proof} \rangle
\]

\[ \text{lemma wlp-DC-split:} \]
\[
\begin{align*}
\{ P \vdash \text{wlp } \text{prog } R; \quad Q \vdash \text{wlp } \text{prog}' R \} \implies \\
(\lambda s. \text{min } (P s) \ (Q s)) \vdash \text{wlp } (\text{prog } \sqcap \text{prog}') \ R
\end{align*}
\]
\[
\langle \text{proof} \rangle
\]

\[ \text{lemma wp-DC-split-same:} \]
\[
\begin{align*}
\{ P \vdash \text{wp } \text{prog } Q; \quad P \vdash \text{wp } \text{prog}' Q \} \implies P \vdash \text{wp } (\text{prog } \sqcap \text{prog}') \ Q
\end{align*}
\]
\[
\langle \text{proof} \rangle
\]

\[ \text{lemma wlp-DC-split-same:} \]
\[
\begin{align*}
\{ P \vdash \text{wlp } \text{prog } Q; \quad P \vdash \text{wlp } \text{prog}' Q \} \implies P \vdash \text{wlp } (\text{prog } \sqcap \text{prog}') \ Q
\end{align*}
\]
\[
\langle \text{proof} \rangle
\]

\[ \text{lemma SetPC-split:} \]
\[
\begin{align*}
\text{fixes } f:: \text{'}x \Rightarrow \text{'}y \text{ prog} \\
\text{and } p:: \text{'}y \Rightarrow \text{'}x \Rightarrow \text{real} \\
\text{assumes rec: } \bigwedge x \ s. \ x \in \text{supp } (p s) \implies P x \vdash f x \ ab \ Q \\
\text{and nnp: } \bigwedge s. \ \text{nneg } (p s) \\
\text{shows } (\lambda s. \sum x \in \text{supp } (p s). \ p s \ x * P x s) \vdash \text{SetPC } f \ p \ ab \ Q
\end{align*}
\]
\[
\langle \text{proof} \rangle
\]
lemma wp-SetPC-split:
\[ \begin{align*}
\forall x. x \in \text{supp} (p \downarrow s) \Rightarrow P x \vdash \text{wp} (f x) Q;
\end{align*} \]
\[ \begin{align*}
(\lambda s. \sum_{x \in \text{supp} (p \downarrow s)} p s x \cdot P x s) \vdash \text{wp} (\text{SetPC} f p) Q
\end{align*} \]
(proof)

lemma wlp-SetPC-split:
\[ \begin{align*}
\forall x. x \in \text{supp} (p \downarrow s) \Rightarrow P x \vdash \text{wlp} (f x) Q;
\end{align*} \]
\[ \begin{align*}
(\lambda s. \sum_{x \in \text{supp} (p \downarrow s)} p s x \cdot P x s) \vdash \text{wlp} (\text{SetPC} f p) Q
\end{align*} \]
(proof)

lemma wp-SetDC-split:
\[ \begin{align*}
\forall s. x \in S s \Rightarrow P x \vdash \text{wp} (f x) Q;
\end{align*} \]
\[ \begin{align*}
P \vdash \text{wp} (\text{SetDC} f S) Q
\end{align*} \]
(proof)

lemma wlp-SetDC-split:
\[ \begin{align*}
\forall s. x \in S s \Rightarrow P x \vdash \text{wlp} (f x) Q;
\end{align*} \]
\[ \begin{align*}
P \vdash \text{wlp} (\text{SetDC} f S) Q
\end{align*} \]
(proof)

lemma wp-SetDC:
\[ \begin{align*}
\text{assumes wp: } \forall s. x \in S s \Rightarrow P x \vdash \text{wp} (f x) Q
\end{align*} \]
\[ \begin{align*}
\text{and } \text{ne: } \forall s. S s \neq \{\} \Rightarrow
\end{align*} \]
\[ \begin{align*}
(\lambda x. \text{Inf} ((\lambda x. P x s) \cdot S s)) \vdash \text{wp} (\text{SetDC} f S) Q
\end{align*} \]
(proof)

lemma wlp-SetDC:
\[ \begin{align*}
\text{assumes wp: } \forall s. x \in S s \Rightarrow P x \vdash \text{wlp} (f x) Q
\end{align*} \]
\[ \begin{align*}
\text{and } \text{ne: } \forall s. S s \neq \{\} \Rightarrow
\end{align*} \]
\[ \begin{align*}
(\lambda x. \text{sound} (P x) \text{Inf} ((\lambda x. P x s) \cdot S s)) \vdash \text{wlp} (\text{SetDC} f S) Q
\end{align*} \]
(proof)

lemma wp-Embed:
\[ \begin{align*}
P \vdash t Q \Rightarrow P \vdash \text{wp} (\text{Embed} t) Q
\end{align*} \]
(proof)

lemma wlp-Embed:
\[ \begin{align*}
P \vdash t Q \Rightarrow P \vdash \text{wlp} (\text{Embed} t) Q
\end{align*} \]
(proof)

lemma wp-Bind:
\[ \begin{align*}
[ \begin{align*}
\forall s. P s \leq \text{wp} (a (f s)) Q s
\end{align*} \] \Rightarrow
\end{align*} \]
\[ \begin{align*}
P \vdash \text{wp} (\text{Bind} f a) Q
\end{align*} \]
(proof)

lemma wlp-Bind:
\[ \begin{align*}
[ \begin{align*}
\forall s. P s \leq \text{wlp} (a (f s)) Q s
\end{align*} \] \Rightarrow
\end{align*} \]
\[ \begin{align*}
P \vdash \text{wlp} (\text{Bind} f a) Q
\end{align*} \]
(proof)
\textbf{lemma} \textit{wp-repeat}: \\
\hspace{1cm} \begin{array}{l}
P \vdash \text{wp } a \ Q; \\
\phantom{P \vdash \text{wp } a \ Q; } Q \vdash \text{wp } (\text{repeat } n \ a) \ R; \\
\phantom{P \vdash \text{wp } a \ Q; } \text{well-def } a; \text{ sound } Q; \text{ sound } R \\ 
\Rightarrow P \vdash \text{wp } (\text{repeat } \text{Suc } n \ a) \ R \\
\end{array} \langle \text{proof} \rangle

\textbf{lemma} \textit{wlp-repeat}: \\
\hspace{1cm} \begin{array}{l}
P \vdash \text{wlp } a \ Q; \\
\phantom{P \vdash \text{wlp } a \ Q; } Q \vdash \text{wlp } (\text{repeat } n \ a) \ R; \\
\phantom{P \vdash \text{wlp } a \ Q; } \text{well-def } a; \text{ unitary } Q; \text{ unitary } R \\ 
\Rightarrow P \vdash \text{wlp } (\text{repeat } \text{Suc } n \ a) \ R \\
\end{array} \langle \text{proof} \rangle

Note that the loop rules presented in section Section 4.8 are of the same form, and would belong here, had they not already been stated.

The following rules are specialisations of those for general transformers, and are easier for the unifier to match.

\textbf{lemmas} \textit{wp-strengthen-post=} \\
\hspace{1cm} \textit{entails-strengthen-post}\left[ \textit{where } t=\text{wp } a \text{ for } a \right]

\textbf{lemma} \textit{wlp-strengthen-post}: \\
\hspace{1cm} P \vdash \text{wlp } a \ Q \Rightarrow \text{nearly-healthy } (\text{wlp } a) \Rightarrow \text{unitary } R \Rightarrow Q \vdash R \Rightarrow \text{unitary } Q \\
\Rightarrow \begin{array}{l}
P \vdash \text{wlp } a \ R \\
\end{array} \langle \text{proof} \rangle

\textbf{lemmas} \textit{wp-weaken-pre=} \\
\hspace{1cm} \textit{entails-weaken-pre}\left[ \textit{where } t=\text{wp } a \text{ for } a \right]

\textbf{lemmas} \textit{wlp-weaken-pre=} \\
\hspace{1cm} \textit{entails-weaken-pre}\left[ \textit{where } t=\text{wlp } a \text{ for } a \right]

\textbf{lemmas} \textit{wp-scale=} \\
\hspace{1cm} \textit{entails-scale}\left[ \textit{where } t=\text{wp } a \text{ for } a, \text{OF - well-def-wp-healthy} \right]

\section{4.10.2 Algebraic Decomposition}

Refinement is a powerful tool for decomposition, belied by the simplicity of the rule. This is an \textit{axiomatic} formulation of refinement (all annotations of the \textit{a} are annotations of \textit{b}), rather than an operational version (all traces of \textit{b} are traces of \textit{a}).

\textbf{lemma} \textit{wp-refines}: \\
\hspace{1cm} \begin{array}{l}
\hspace{1cm} a \sqsubseteq b; P \vdash \text{wp } a \ Q; \text{ sound } Q \\ 
\hspace{1cm} \Rightarrow P \vdash \text{wp } b \ Q \\
\end{array} \langle \text{proof} \rangle

\textbf{lemmas} \textit{wp-drefines} = drefinesD
4.10.3 Hoare triples

The Hoare triple, or validity predicate, is logically equivalent to the weakest-precondition entailment form. The benefit is that it allows us to define transitivity rules for computational (also/finally) reasoning.

**Definition**

\[
wp\text{-valid} :: (\lambda a. \text{real}) \implies \lambda \text{prog} \implies (\lambda a. \text{real}) \implies \text{bool}
\]

\[
\{ |- \} - \{ |- \} p
\]

**Where**

\[
wp\text{-valid} \ P \ \text{prog} \ Q \equiv P \vdash wp \ \text{prog} \ Q
\]

**Lemma** \(wp\text{-validI}:

\[
P \vdash wp \ \text{prog} \ Q \implies \{P\} \ \text{prog} \ \{Q\} p
\]

**Lemma** \(wp\text{-validD}:

\[
\{P\} \ \text{prog} \ \{Q\} p \implies P \vdash wp \ \text{prog} \ Q
\]

**Lemma** \(valid\text{-Seq}:

\[
[ [ \{P\} \ a \ \{Q\} p; \ \{Q\} b \ \{R\} p; \ \text{well-def} \ a; \ \text{well-def} \ b; \ \text{sound} \ Q; \ \text{sound} \ R ] ] \implies
\]

\[
\{P\} \ a ;; \ b \ \{R\} p
\]

We make it available to the computational reasoner:

**Declare** \(valid\text{-Seq}[\text{trans}]

**End**

4.11 Loop Termination

**Theory** Terminations imports Embedding StructuredReasoning Loops begin

Termination for loops can be shown by classical means (using a variant, or a measure function), or by probabilistic means: We only need that the loop terminates with probability one.

4.11.1 Trivial Termination

A maximal transformer (program) doesn’t affect termination. This is essentially saying that such a program doesn’t abort (or diverge).

**Lemma** maximal-Seq-term:

**Fixes** \(r::s \ \text{prog} \ \text{and} \ s::s \ \text{prog}

**Assumes** \(mr::\text{maximal} (wp \ r)

\text{and} \ ws::\text{well-def} \ s

\text{and} \ ts::(\lambda s. \ 1) \vdash wp \ s \ (\lambda s. I)

**Shows** (\lambda s. \ 1) \vdash wp \ (r ;; s) \ (\lambda s. I)

**Proof**
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)

### 4.11.2 Classical Termination

The first non-trivial termination result is quite standard: If we can provide a natural-number-valued measure, that decreases on every iteration, and implies termination on reaching zero, the loop terminates.

**lemma** loop-term-nat-measure-noinv:
  **fixes** m :: ′s ⇒ nat and body :: ′s prog
  **assumes** wb: well-def body
  and guard: ∀s. m s = 0 → ¬ G s
  and variant: ∀n. «λs. m s = Suc n» ⊢ wp body «λs. m s = n»
  **shows** λs. 1 ⊢ wp do G → body od (λs. 1)

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 do G → body od (λs. 1)

### 4.11.3 Probabilistic Termination

Any loop that has a non-zero chance of terminating after each step terminates with probability 1.

**lemma** termination-0-1:
  **fixes** body :: ′s prog
  **assumes** wb: well-def body
  — The loop terminates in one step with nonzero probability
  and onestep: (λs. p) ⊢ wp body «N G»
  and nzp: 0 < p
  — The body is maximal i.e. it terminates absolutely.
  and mb: maximal (wp body)
  **shows** λs. 1 ⊢ wp do G → body od (λs. 1)

end
4.12 Automated Reasoning

theory Automation imports StructuredReasoning
begin

This theory serves as a container for automated reasoning tactics for pGCL, implemented in ML. At present, there is a basic verification condition generator (VCG).

named-theorems wd
  theorems to automatically establish well-definedness
named-theorems pwp-core
  core probabilistic wp rules, for evaluating primitive terms
named-theorems pwp
  user-supplied probabilistic wp rules
named-theorems pwp
  user-supplied probabilistic wlp rules

⟨ML⟩

declare wd-intros[wd]

lemmas core-wp-rules =
  wp-Skip    wlp-Skip
  wp-Abort   wlp-Abort
  wp-Apply   wlp-Apply
  wp-Seq     wlp-Seq
  wp-DC-split wlp-DC-split
  wp-PC-fixed wlp-PC-fixed
  wp-SetDC   wlp-SetDC
  wp-SetPC-split wlp-SetPC-split

declare core-wp-rules[pwp-core]

end
4.13 Miscellaneous Mathematics

theory Misc
imports
  HOL−Analysis.Analysis
begin

lemma sum-UNIV:
  fixes S::'a::finite set
  assumes complete: \( \forall x. x \notin S \Rightarrow f x = 0 \)
  shows \( \sum f S = \sum f \text{UNIV} \)
  ⟨proof⟩

lemma cInf-mono:
  fixes A::'a::conditionally-complete-lattice set
  assumes lower: \( \forall b. b \in B \Rightarrow \exists a \in A. a \leq b \)
  and bounded: \( \forall a. a \in A \Rightarrow c \leq a \)
  and ne: B \neq \{\}
  shows \( \inf A \leq \inf B \)
  ⟨proof⟩

lemma max-distrib:
  fixes c::real
  assumes nn: \( 0 \leq c \)
  shows \( c \cdot \max a b = \max (c \cdot a) (c \cdot b) \)
  ⟨proof⟩

lemma mult-div-mono-left:
  fixes c::real
  assumes nnc: \( 0 \leq c \) and nzc: \( c \neq 0 \)
  and inv: \( a \leq \text{inverse} \ c \cdot b \)
  shows \( c \cdot a \leq b \)
  ⟨proof⟩

lemma mult-div-mono-right:
  fixes c::real
  assumes nnc: \( 0 \leq c \) and nzc: \( c \neq 0 \)
  and inv: \( \text{inverse} \ c \cdot a \leq b \)
  shows \( a \leq c \cdot b \)
  ⟨proof⟩

⟨proof⟩

107
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 (λ i. 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 = (λ y. if y=x then 0 else p y / (1 - p x))

lemma supp-dist-remove:
4.13. MISCELLANEOUS MATHEMATICS

\[ p x \neq 0 \implies p x \neq 1 \implies \text{supp} (\text{dist-remove} p x) = \text{supp} p - \{x\} \]

\langle proof \rangle

\text{lemma supp-empty:} \\
\text{supp} f = \{\} \implies f x = 0 \\
\langle proof \rangle

\text{lemma nsupp-zero:} \\
x \notin \text{supp} f \implies f x = 0 \\
\langle proof \rangle

\text{lemma sum-supp:} \\
\text{fixes} f :: 'a::finite \Rightarrow \text{real} \\
\text{shows} \sum f (\text{supp} f) = \sum f \text{UNIV} \\
\langle proof \rangle

4.13.1 Truncated Subtraction

\text{definition} \\
t minus :: real \Rightarrow real \Rightarrow real (infixl \odot 60) \\
\text{where} \\
x \odot y = \max (x - y) \ 0

\text{lemma minus-le-tminus[intro,simp]:} \\
a - b \leq a \odot b \\
\langle proof \rangle

\text{lemma tminus-cancel-1:} \\
0 \leq a \implies a + 1 \odot 1 = a \\
\langle proof \rangle

\text{lemma tminus-zero-imp-le:} \\
x \odot y \leq 0 \implies x \leq y \\
\langle proof \rangle

\text{lemma tminus-zero[simp]:} \\
0 \leq x \implies x \odot 0 = x \\
\langle proof \rangle

\text{lemma tminus-left-mono:} \\
a \leq b \implies a \odot c \leq b \odot c \\
\langle proof \rangle

\text{lemma tminus-less:} \\
[ 0 \leq a; 0 \leq b ] \implies a \odot b \leq a \\
\langle proof \rangle

\text{lemma tminus-left-distrib:} \\
\text{assumes} \ nna: 0 \leq a
shows \( a * (b \ominus c) = a * b \ominus a * c \)
\langle proof \rangle

lemma tminus-le[simp]:
\( b \leq a \implies a \ominus b = a - b \)
\langle proof \rangle

lemma tminus-le-alt[simp]:
\( a \leq b \implies a \ominus b = 0 \)
\langle proof \rangle

lemma tminus-nle[simp]:
\( \neg b \leq a \implies a \ominus b = 0 \)
\langle proof \rangle

lemma tminus-add-mono:
\( (a+b) \ominus (c+d) \leq (a\ominus c) + (b\ominus d) \)
\langle proof \rangle

lemma tminus-sum-mono:
\textbf{assumes} \( fS : \text{finite} \ S \)
\textbf{shows} \( \text{sum} \ f \ S \ominus \text{sum} \ g \ S \leq \text{sum} \ (\lambda x. \ f \ x \ominus g \ x) \ S \)
\( \quad \text{(is ?X S)} \)
\langle proof \rangle

lemma tminus-nneg[simp,intro]:
\( \varnothing \leq a \ominus b \)
\langle proof \rangle

lemma tminus-right-antimono:
\textbf{assumes} \( clb : c \leq b \)
\textbf{shows} \( a \ominus b \leq a \ominus c \)
\langle proof \rangle

lemma min-tminus-distrib:
\( \min \ a \ b \ominus c = \min \ (a \ominus c) \ (b \ominus c) \)
\langle proof \rangle

end
Bibliography


