

# pGCL for Isabelle

David Cock

May 26, 2024



# Contents

<b>1</b>	<b>Overview</b>	<b>1</b>
<b>2</b>	<b>Introduction to pGCL</b>	<b>3</b>
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 . . . . .	7
2.3.1	The State Space . . . . .	8
2.3.2	The Game . . . . .	8
2.3.3	A Brute Force Solution . . . . .	9
2.3.4	A Modular Approach . . . . .	10
<b>3</b>	<b>Semantic Structures</b>	<b>13</b>
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 . . . . .	21
3.1.7	Expectation Conjunction . . . . .	22
3.1.8	Rules Involving Conjunction. . . . .	23
3.1.9	Rules Involving Entailment and Conjunction Together . . . . .	25
3.2	Expectation Transformers . . . . .	26
3.2.1	Comparing Transformers . . . . .	28
3.2.2	Healthy Transformers . . . . .	32
3.2.3	Sublinearity . . . . .	37

3.2.4	Determinism	41
3.2.5	Modular Reasoning	44
3.2.6	Transforming Standard Expectations	44
3.3	Induction	45
3.3.1	The Lattice of Expectations	45
3.3.2	The Lattice of Transformers	48
3.3.3	Tail Recursion	51
<b>4</b>	<b>The pGCL Language</b>	<b>53</b>
4.1	A Shallow Embedding of pGCL in HOL	53
4.1.1	Core Primitives and Syntax	53
4.1.2	Unfolding rules for non-recursive primitives	56
4.2	Healthiness	58
4.2.1	The Healthiness of the Embedding	59
4.2.2	Healthiness for Loops	61
4.3	Continuity	64
4.3.1	Continuity of Primitives	65
4.3.2	Continuity of a Single Loop Step	68
4.4	Continuity and Induction for Loops	68
4.4.1	The Limit of Iterates	69
4.5	Sublinearity	71
4.5.1	Nonrecursive Primitives	71
4.5.2	Sublinearity for Loops	72
4.6	Determinism	74
4.6.1	Additivity	74
4.6.2	Maximality	75
4.6.3	Determinism	76
4.7	Well-Defined Programs.	77
4.7.1	Strict Implies Liberal	77
4.7.2	Sub-Distributivity of Conjunction	79
4.7.3	The Well-Defined Predicate.	82
4.8	The Loop Rules	84
4.8.1	Liberal and Strict Invariants.	84
4.8.2	Partial Correctness	85
4.8.3	Total Correctness	85
4.8.4	Unfolding	85
4.9	The Algebra of pGCL	86
4.9.1	Program Refinement	86
4.9.2	Simple Identities	87
4.9.3	Deterministic Programs are Maximal	90
4.9.4	The Algebraic Structure of Refinement	90
4.9.5	Data Refinement	91
4.9.6	The Algebra of Data Refinement	93
4.9.7	Structural Rules for Correspondence	94

4.9.8	Structural Rules for Data Refinement . . . . .	95
4.10	Structured Reasoning . . . . .	96
4.10.1	Syntactic Decomposition . . . . .	96
4.10.2	Algebraic Decomposition . . . . .	100
4.10.3	Hoare triples . . . . .	100
4.11	Loop Termination . . . . .	101
4.11.1	Trivial Termination . . . . .	101
4.11.2	Classical Termination . . . . .	101
4.11.3	Probabilistic Termination . . . . .	102
4.12	Automated Reasoning . . . . .	102
<b>Additional Material</b>		<b>105</b>
4.13	Miscellaneous Mathematics . . . . .	105
4.13.1	Truncated Subtraction . . . . .	107



# Chapter 1

## Overview

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

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

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





## Chapter 2

# Introduction to pGCL

### 2.1 Language Primitives

**theory** *Primitives* **imports** *../pGCL* **begin**

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

#### 2.1.1 The Basics

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

**datatype** *coin* = *Heads* | *Tails*

**record** *coins* =

$a :: \textit{coin}$

$b :: \textit{coin}$

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

**lemma**

$\textit{Apply} (\lambda s. s \langle a := \textit{Heads} \rangle) = (a := (\lambda s. \textit{Heads}))$

$\langle \textit{proof} \rangle$

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 *Apply* ( $\lambda s. s \langle a := b \ s \rangle$ ), 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  $\ll P \gg::'s \Rightarrow real$ , such that  $P s \longrightarrow \ll P \gg s = 1 \wedge \neg P s \longrightarrow \ll P \gg s = 0$ .

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  $\ll P \gg \Vdash \ll Q \gg$ ,  $\ll \ll ?P \gg \Vdash \ll ?Q \gg; ?P ?s \rrbracket \Longrightarrow ?Q ?s$ . Thus a valid specification of *Apply* ( $\lambda s. s(a := b s)$ ) is:

**lemma**

$$\bigwedge x. \ll \lambda s. b s = x \gg \Vdash wp (a := b) \ll \lambda s. a s = x \gg$$

*<proof>*

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

### 2.1.3 Probability

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

**lemma**

$$\ll \lambda s::real. s \neq 0 \gg \Vdash wp (Apply ((* 2) ;; Apply (\lambda s. s / s)) \ll \lambda s. s = 1 \gg$$

*<proof>*

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

**definition**

$flip-a :: real \Rightarrow coins\ prog$   
**where**  
 $flip-a\ p = a := (\lambda-. Heads) (\lambda s. p) \oplus a := (\lambda-. Tails)$

**lemma**  
 $(\lambda s. 1/2) = wp\ (flip-a\ (1/2)) \ll \lambda s. a\ s = Heads \gg$   
 $\langle proof \rangle$

### 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**  
 $\lambda s. 1/3 \Vdash wp\ (flip-a\ (2/3) \sqcap flip-a\ (1/3)) \ll \lambda s. a\ s = Heads \gg$   
 $\langle proof \rangle$

### 2.1.5 Properties of Expectations

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

**definition**  
 $flip-b :: real \Rightarrow coins\ prog$   
**where**  
 $flip-b\ p = b := (\lambda-. Heads) (\lambda s. p) \oplus b := (\lambda-. Tails)$

**lemma**  
 $(\lambda s. 1/4) = wp\ (flip-a\ (1/2) ;; flip-b\ (1/2))$   
 $\ll \lambda s. a\ s = Heads \wedge b\ s = Heads \gg$   
 $\langle proof \rangle$

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

**record**  $dice =$   
 $red :: nat$   
 $blue :: nat$

**definition**  $Puniform :: 'a\ set \Rightarrow ('a \Rightarrow real)$   
**where**  $Puniform\ S = (\lambda x. if\ x \in S\ then\ 1 / card\ S\ else\ 0)$

**lemma**  $Puniform-in:$   
 $x \in S \Longrightarrow Puniform\ S\ x = 1 / card\ S$   
 $\langle proof \rangle$

**lemma** *Puniform-out*:

$$x \notin S \implies \text{Puniform } S \ x = 0$$

*<proof>*

**lemma** *supp-Puniform*:

$$\text{finite } S \implies \text{supp } (\text{Puniform } S) = S$$

*<proof>*

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

**lemma**

$$(\lambda s. 7/2) = \text{wp } (\text{bind } v \text{ at } (\lambda s. \text{Puniform } \{1..6\} \ v) \text{ in } \text{red} := (\lambda \cdot. v)) \ \text{red}$$

*<proof>*

The expectations of independent variables add:

**lemma**

$$\begin{aligned} (\lambda s. 7) = \text{wp } (&(\text{bind } v \text{ at } (\lambda s. \text{Puniform } \{1..6\} \ v) \text{ in } \text{red} := (\lambda s. v)) \ ; \\ &(\text{bind } v \text{ at } (\lambda s. \text{Puniform } \{1..6\} \ v) \text{ in } \text{blue} := (\lambda s. v))) \\ &(\lambda s. \text{red } s + \text{blue } s) \end{aligned}$$

*<proof>*

**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**

$$\text{countdown} = \text{do } (\lambda x. 0 < x) \longrightarrow \text{Apply } (\lambda s. s - 1) \ \text{od}$$

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

**definition** *inv-count* **::** *int  $\Rightarrow$  bool*

**where**

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

Read *wp-inv G body I* as: *I* is an invariant of the loop  $\mu x. \text{body} ;; x \ll G \gg \oplus \text{Skip}$ , or  $\ll G \gg \&\& I \Vdash \text{wp body } I$ .

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

*wp-inv* ( $\lambda x. 0 < x$ ) (*Apply* ( $\lambda s. s - 1$ ))  $\ll \text{inv-count} \gg$   
 $\langle \text{proof} \rangle$

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

**lemma** *term-countdown*:

$\ll \text{inv-count} \gg \Vdash \text{wp countdown } (\lambda s. 1)$   
 $\langle \text{proof} \rangle$

## 2.2.2 Probabilistic Termination

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

**type-synonym** *coin* = *bool*

**definition** *Heads* = *True*

**definition** *Tails* = *False*

**definition**

*flip* :: *coin prog*

**where**

*flip* = *Apply* ( $\lambda \cdot. \text{Heads}$ ) ( $\lambda s. 1/2$ )  $\oplus$  *Apply* ( $\lambda \cdot. \text{Tails}$ )

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**

*wait-for-heads* = *do* ( $(\neq) \text{Heads}$ )  $\longrightarrow$  *flip od*

Nonetheless, we can show termination .

**lemma** *wait-for-heads-term*:

$\lambda s. 1 \Vdash \text{wp wait-for-heads } (\lambda s. 1)$   
 $\langle \text{proof} \rangle$

**end**

## 2.3 The Monty Hall Problem

**theory** *Monty* **imports** *../pGCL begin*

We now tackle a more substantial example, allowing us to demonstrate the tools for compositional reasoning and the use of invariants in non-recursive programs.

Our example is the well-known Monty Hall puzzle in statistical inference [Selvin, 1975].

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

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

### 2.3.1 The State Space

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

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

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

```
definition player-wins :: game  $\Rightarrow$  bool
where player-wins g  $\equiv$  guess g = prize g
```

#### Invariants

We prove explicitly that only valid doors are ever chosen.

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

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

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

### 2.3.2 The Game

Hide the prize behind door  $D$ .

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

Choose door  $D$ .

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

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

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

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

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

Guess uniformly at random.

**definition** *make-guess* ::  $game\ prog$   
**where** *make-guess*  $\equiv guess-behind\ 1\ (\lambda s. 1/3) \oplus$   
 $guess-behind\ 2\ (\lambda s. 1/2) \oplus guess-behind\ 3$

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

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

Switch your guess to the other unopened door.

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

The complete game, either with or without switching guesses.

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

### 2.3.3 A Brute Force Solution

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

**lemma** *eval-win[simp]*:  
 $p = g \implies \langle player-wins \rangle (s \mid prize := p, guess := g, clue := c) = 1$   
 $\langle proof \rangle$

**lemma** *eval-loss[simp]*:  
 $p \neq g \implies \langle player-wins \rangle (s \mid prize := p, guess := g, clue := c) = 0$   
 $\langle proof \rangle$

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

**lemma** *wp-monty-noswitch*:  
 $(\lambda s. 1/3) = wp \text{ (monty False) } \llbracket \text{player-wins} \rrbracket$   
 $\langle \text{proof} \rangle$

**lemma** *swap-upd*:  
 $s(\lambda \text{ prize} := p, \text{ clue} := c, \text{ guess} := g) =$   
 $s(\lambda \text{ prize} := p, \text{ guess} := g, \text{ clue} := c)$   
 $\langle \text{proof} \rangle$

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

**lemma** *wp-monty-switch-bruteforce*:  
 $(\lambda s. 2/3) = wp \text{ (monty True) } \llbracket \text{player-wins} \rrbracket$   
 $\langle \text{proof} \rangle$

### 2.3.4 A Modular Approach

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

#### Healthiness

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

**lemma** *wd-hide-prize*:  
 $\text{well-def hide-prize}$   
 $\langle \text{proof} \rangle$

**lemma** *wd-make-guess*:  
 $\text{well-def make-guess}$   
 $\langle \text{proof} \rangle$

**lemma** *wd-reveal*:  
 $\text{well-def reveal}$   
 $\langle \text{proof} \rangle$

**lemma** *wd-switch-guess*:  
 $\text{well-def switch-guess}$   
 $\langle \text{proof} \rangle$

**lemmas** *monty-healthy* =  
 $\text{wd-switch-guess wd-reveal wd-make-guess wd-hide-prize}$



### Annotations

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

*hide-prize* chooses a valid door.

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

$$(\lambda s. I) \Vdash wp \text{ hide-prize } \langle \text{inv-prize} \rangle$$

*<proof>*

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

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

$$(\lambda s. 2/3 * \langle \lambda g. \text{inv-prize } g \rangle s) \Vdash$$

$$wp \text{ make-guess } \langle \lambda g. \text{guess } g \neq \text{prize } g \wedge \text{inv-prize } g \wedge \text{inv-guess } g \rangle$$

*<proof>*

**lemma** *last-one*:

**assumes**  $a \neq b$  **and**  $a \in \{1::\text{nat}, 2, 3\}$  **and**  $b \in \{1, 2, 3\}$   
**shows**  $\exists! c. \{1, 2, 3\} - \{b, a\} = \{c\}$   
*<proof>*

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

**lemma** *wp-reveal*:

$$\langle \lambda g. \text{guess } g \neq \text{prize } g \wedge \text{inv-prize } g \wedge \text{inv-guess } g \rangle \Vdash$$

$$wp \text{ reveal } \langle \lambda g. \text{guess } g \neq \text{prize } g \wedge$$

$$\text{clue } g \neq \text{prize } g \wedge$$

$$\text{clue } g \neq \text{guess } g \wedge$$

$$\text{inv-prize } g \wedge \text{inv-guess } g \wedge \text{inv-clue } g \rangle$$

(**is** ?X  $\Vdash$  wp reveal ?Y)  
*<proof>*

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

**lemma** *distinct-game*:

$$\llbracket \text{guess } g \neq \text{prize } g; \text{clue } g \neq \text{prize } g; \text{clue } g \neq \text{guess } g;$$

$$\text{inv-prize } g; \text{inv-guess } g; \text{inv-clue } g \rrbracket \implies$$

$$\{1, 2, 3\} = \{\text{guess } g, \text{prize } g, \text{clue } g\}$$

*<proof>*

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

**lemma** *wp-switch-guess*:

$$\langle \lambda g. \text{guess } g \neq \text{prize } g \wedge \text{clue } g \neq \text{prize } g \wedge \text{clue } g \neq \text{guess } g \wedge$$

$$\text{inv-prize } g \wedge \text{inv-guess } g \wedge \text{inv-clue } g \rangle \Vdash$$

$$wp \text{ switch-guess } \langle \text{player-wins} \rangle$$

*<proof>*

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

**lemma** *wp-monty-switch-modular*:  
 $(\lambda s. 2/3) \Vdash wp \text{ (monty True) } \langle \text{player-wins} \rangle$   
 $\langle \text{proof} \rangle$

### Using the VCG

**lemmas** *scaled-hide* = *wp-scale*[*OF wp-hide-prize, simplified*]  
**declare** *scaled-hide*[*pwp*] *wp-make-guess*[*pwp*] *wp-reveal*[*pwp*] *wp-switch-guess*[*pwp*]  
**declare** *wd-hide-prize*[*wd*] *wd-make-guess*[*wd*] *wd-reveal*[*wd*] *wd-switch-guess*[*wd*]

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

**lemma** *wp-monty-switch-vcg*:  
 $(\lambda s. 2/3) \Vdash wp \text{ (monty True) } \langle \text{player-wins} \rangle$   
 $\langle \text{proof} \rangle$

**end**

## Chapter 3

# Semantic Structures

### 3.1 Expectations

**theory** *Expectations* **imports** *Misc* **begin type-synonym** *'s expect = 's  $\Rightarrow$  real*

Expectations are a real-valued generalisation of boolean predicates: An expectation on state *s* is a function  $'s \Rightarrow real$ . 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:

<i>a</i>	<i>b</i>	$a \rightarrow b$	<i>x</i>	<i>y</i>	$x \leq y$
F	F	T	0	0	T
F	T	T	0	1	T
T	F	F	1	0	F
T	T	T	1	1	T

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

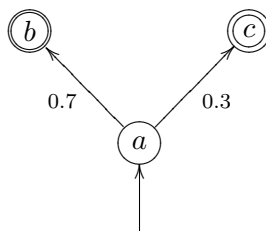


Figure 3.1: A probabilistic automaton

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

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

### 3.1.1 Bounded Functions

**definition** *bounded-by* ::  $(a \Rightarrow \text{real}) \Rightarrow \text{bool}$

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

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

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

$$\llbracket \bigwedge x. P x \leq b \rrbracket \Longrightarrow \text{bounded-by } b P$$

*<proof>*

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

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

*<proof>*

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

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

*<proof>*

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

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

*<proof>*

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

**definition** *bounded* ::  $(a \Rightarrow \text{real}) \Rightarrow \text{bool}$

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

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

**definition** *bound-of* ::  $(a \Rightarrow \text{real}) \Rightarrow \text{real}$

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

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

**assumes**  $bP$ : *bounded*  $P$   
**shows** *bdd-above* (*range*  $P$ )  
*<proof>*

The least upper bound has the usual properties:

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

**assumes**  $bP$ : *bounded-by*  $b P$   
**shows** *bound-of*  $P \leq b$   
 $\langle \text{proof} \rangle$

**lemma** *bounded-by-bound-of*[*intro!*]:  
**fixes**  $P::'a \Rightarrow \text{real}$   
**assumes**  $bP$ : *bounded*  $P$   
**shows** *bounded-by* (*bound-of*  $P$ )  $P$   
 $\langle \text{proof} \rangle$

**lemma** *bound-of-greater*[*intro*]:  
*bounded*  $P \Longrightarrow P x \leq \text{bound-of } P$   
 $\langle \text{proof} \rangle$

**lemma** *bounded-by-mono*:  
 $\llbracket \text{bounded-by } a P; a \leq b \rrbracket \Longrightarrow \text{bounded-by } b P$   
 $\langle \text{proof} \rangle$

**lemma** *bounded-by-imp-bounded*[*intro*]:  
*bounded-by*  $b P \Longrightarrow \text{bounded } P$   
 $\langle \text{proof} \rangle$

This is occasionally easier to apply:

**lemma** *bounded-by-bound-of-alt*:  
 $\llbracket \text{bounded } P; \text{bound-of } P = a \rrbracket \Longrightarrow \text{bounded-by } a P$   
 $\langle \text{proof} \rangle$

**lemma** *bounded-const*[*simp*]:  
*bounded*  $(\lambda x. c)$   
 $\langle \text{proof} \rangle$

**lemma** *bounded-by-const*[*intro*]:  
 $c \leq b \Longrightarrow \text{bounded-by } b (\lambda x. c)$   
 $\langle \text{proof} \rangle$

**lemma** *bounded-by-mono-alt*[*intro*]:  
 $\llbracket \text{bounded-by } b Q; P \leq Q \rrbracket \Longrightarrow \text{bounded-by } b P$   
 $\langle \text{proof} \rangle$

**lemma** *bound-of-const*[*simp, intro*]:  
*bound-of*  $(\lambda x. c) = (c::\text{real})$   
 $\langle \text{proof} \rangle$

**lemma** *bound-of-leI*:  
**assumes**  $\bigwedge x. P x \leq (c::\text{real})$   
**shows** *bound-of*  $P \leq c$   
 $\langle \text{proof} \rangle$

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

$\llbracket P \leq Q; \text{bounded } P; \text{bounded } Q \rrbracket \Longrightarrow \text{bound-of } P \leq \text{bound-of } Q$   
 $\langle \text{proof} \rangle$

**lemma** *bounded-by-o*[*intro,simp*]:  
 $\bigwedge b. \text{bounded-by } b P \Longrightarrow \text{bounded-by } b (P \circ f)$   
 $\langle \text{proof} \rangle$

**lemma** *le-bound-of*[*intro*]:  
 $\bigwedge x. \text{bounded } f \Longrightarrow f x \leq \text{bound-of } f$   
 $\langle \text{proof} \rangle$

### 3.1.2 Non-Negative Functions.

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

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

**lemma** *nnegI*[*intro*]:  
 $\llbracket \bigwedge x. 0 \leq P x \rrbracket \Longrightarrow nneg P$   
 $\langle \text{proof} \rangle$

**lemma** *nnegI2*[*intro*]:  
 $(\lambda s. 0) \leq P \Longrightarrow nneg P$   
 $\langle \text{proof} \rangle$

**lemma** *nnegD*[*dest*]:  
 $nneg P \Longrightarrow 0 \leq P x$   
 $\langle \text{proof} \rangle$

**lemma** *nnegD2*[*dest*]:  
 $nneg P \Longrightarrow (\lambda s. 0) \leq P$   
 $\langle \text{proof} \rangle$

**lemma** *nneg-bdd-below*[*intro*]:  
 $nneg P \Longrightarrow \text{bdd-below } (\text{range } P)$   
 $\langle \text{proof} \rangle$

**lemma** *nneg-const*[*iff*]:  
 $nneg (\lambda x. c) \longleftrightarrow 0 \leq c$   
 $\langle \text{proof} \rangle$

**lemma** *nneg-o*[*intro,simp*]:  
 $nneg P \Longrightarrow nneg (P \circ f)$   
 $\langle \text{proof} \rangle$

**lemma** *nneg-bound-nneg*[*intro*]:

$\llbracket \text{bounded } P; \text{nneg } P \rrbracket \Longrightarrow 0 \leq \text{bound-of } P$   
 $\langle \text{proof} \rangle$

**lemma** *nneg-bounded-by-nneg*[*dest*]:  
 $\llbracket \text{bounded-by } b P; \text{nneg } P \rrbracket \Longrightarrow 0 \leq (b::\text{real})$   
 $\langle \text{proof} \rangle$

**lemma** *bounded-by-nneg*[*dest*]:  
**fixes**  $P::'s \Rightarrow \text{real}$   
**shows**  $\llbracket \text{bounded-by } b P; \text{nneg } P \rrbracket \Longrightarrow 0 \leq b$   
 $\langle \text{proof} \rangle$

### 3.1.3 Sound Expectations

**definition** *sound* ::  $('s \Rightarrow \text{real}) \Rightarrow \text{bool}$   
**where**  $\text{sound } P \equiv \text{bounded } P \wedge \text{nneg } P$

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.  $\text{sound } P \Longrightarrow 0 \leq P s$ ) will usually follow by *blast*, *force* or *simp*.

**lemma** *soundI*:  
 $\llbracket \text{bounded } P; \text{nneg } P \rrbracket \Longrightarrow \text{sound } P$   
 $\langle \text{proof} \rangle$

**lemma** *soundI2*[*intro*]:  
 $\llbracket \text{bounded-by } b P; \text{nneg } P \rrbracket \Longrightarrow \text{sound } P$   
 $\langle \text{proof} \rangle$

**lemma** *sound-bounded*[*dest*]:  
 $\text{sound } P \Longrightarrow \text{bounded } P$   
 $\langle \text{proof} \rangle$

**lemma** *sound-nneg*[*dest*]:  
 $\text{sound } P \Longrightarrow \text{nneg } P$   
 $\langle \text{proof} \rangle$

**lemma** *bound-of-sound*[*intro*]:  
**assumes**  $sP: \text{sound } P$   
**shows**  $0 \leq \text{bound-of } P$   
 $\langle \text{proof} \rangle$

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

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

**lemma** *mult-sound*:

**assumes**  $sP$ : *sound*  $P$  **and**  $sQ$ : *sound*  $Q$

**shows** *sound*  $(\lambda s. P s * Q s)$

$\langle$ *proof* $\rangle$

**lemma** *div-sound*:

**assumes**  $sP$ : *sound*  $P$  **and**  $cpos$ :  $0 < c$

**shows** *sound*  $(\lambda s. P s / c)$

$\langle$ *proof* $\rangle$

**lemma** *tminus-sound*:

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

**shows** *sound*  $(\lambda s. P s \ominus c)$

$\langle$ *proof* $\rangle$

**lemma** *const-sound*:

$0 \leq c \implies$  *sound*  $(\lambda s. c)$

$\langle$ *proof* $\rangle$

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

*sound*  $P \implies$  *sound*  $(P o f)$

$\langle$ *proof* $\rangle$

**lemma** *sc-bounded-by*[*intro,simp*]:

$\llbracket$  *sound*  $P$ ;  $0 \leq c$   $\rrbracket \implies$  *bounded-by*  $(c * \text{bound-of } P)$   $(\lambda x. c * P x)$

$\langle$ *proof* $\rangle$

**lemma** *sc-bounded*[*intro,simp*]:

**assumes**  $sP$ : *sound*  $P$  **and**  $pos$ :  $0 \leq c$

**shows** *bounded*  $(\lambda x. c * P x)$

$\langle$ *proof* $\rangle$

**lemma** *sc-bound*[*simp*]:

**assumes**  $sP$ : *sound*  $P$

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

**shows**  $c * \text{bound-of } P = \text{bound-of } (\lambda x. c * P x)$

$\langle$ *proof* $\rangle$

**lemma** *sc-sound*:

$\llbracket$  *sound*  $P$ ;  $0 \leq c$   $\rrbracket \implies$  *sound*  $(\lambda s. c * P s)$

$\langle$ *proof* $\rangle$

**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*  $(a * b)$   $(\lambda s. P s * Q s)$

$\langle$ *proof* $\rangle$

**lemma** *bounded-by-add*:



**fixes**  $P :: 's \Rightarrow \text{real}$  **and**  $Q$   
**assumes**  $bP$ : *bounded-by*  $a P$   
**and**  $bQ$ : *bounded-by*  $b Q$   
**shows** *bounded-by*  $(a + b)$   $(\lambda s. P s + Q s)$   
 $\langle \text{proof} \rangle$

**lemma** *sound-unit*[*intro!,simp*]:  
*sound*  $(\lambda s. 1)$   
 $\langle \text{proof} \rangle$

**lemma** *unit-mult*[*intro*]:  
**assumes**  $sP$ : *sound*  $P$  **and**  $bP$ : *bounded-by*  $1 P$   
**and**  $sQ$ : *sound*  $Q$  **and**  $bQ$ : *bounded-by*  $1 Q$   
**shows** *bounded-by*  $1$   $(\lambda s. P s * Q s)$   
 $\langle \text{proof} \rangle$

**lemma** *sum-sound*:  
**assumes**  $sP$ :  $\forall x \in S. \text{sound } (P x)$   
**shows** *sound*  $(\lambda s. \sum x \in S. P x s)$   
 $\langle \text{proof} \rangle$

### 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*  $:: 's \text{ expect} \Rightarrow \text{bool}$   
**where** *unitary*  $P \longleftrightarrow \text{sound } P \wedge \text{bounded-by } 1 P$

**lemma** *unitaryI*[*intro*]:  
 $\llbracket \text{sound } P; \text{bounded-by } 1 P \rrbracket \Longrightarrow \text{unitary } P$   
 $\langle \text{proof} \rangle$

**lemma** *unitaryI2*:  
 $\llbracket \text{nneg } P; \text{bounded-by } 1 P \rrbracket \Longrightarrow \text{unitary } P$   
 $\langle \text{proof} \rangle$

**lemma** *unitary-sound*[*dest*]:  
*unitary*  $P \Longrightarrow \text{sound } P$   
 $\langle \text{proof} \rangle$

**lemma** *unitary-bound*[*dest*]:  
*unitary*  $P \Longrightarrow \text{bounded-by } 1 P$   
 $\langle \text{proof} \rangle$

### 3.1.5 Standard Expectations

**definition**  
*embed-bool*  $:: ('s \Rightarrow \text{bool}) \Rightarrow 's \Rightarrow \text{real}$  ( $\ll - \gg 1000$ )  
**where**

$$\langle P \rangle \equiv (\lambda s. \text{if } P \text{ } s \text{ then } 1 \text{ else } 0)$$

Standard expectations are the embeddings of boolean predicates, mapping *False* to 0 and *True* to 1. We write  $\langle P \rangle$  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]:

$$\text{nneg } \langle P \rangle \\ \langle \text{proof} \rangle$$

**lemma** *embed-bool-bounded-by-1*[simp,intro]:

$$\text{bounded-by } 1 \langle P \rangle \\ \langle \text{proof} \rangle$$

**lemma** *embed-bool-bounded*[simp,intro]:

$$\text{bounded } \langle P \rangle \\ \langle \text{proof} \rangle$$

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

**lemma** *embed-bool-idem*:

$$\langle P \rangle s * \langle P \rangle s = \langle P \rangle s \\ \langle \text{proof} \rangle$$

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

$$P s \implies \langle P \rangle s = 1 \\ \langle \text{proof} \rangle$$

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

$$\neg P s \implies \langle P \rangle s = 0 \\ \langle \text{proof} \rangle$$

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

$$0 \leq \langle G \rangle s \\ \langle \text{proof} \rangle$$

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

$$\langle G \rangle s \leq 1 \\ \langle \text{proof} \rangle$$

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

$$0 \leq 1 - \langle G \rangle s \\ \langle \text{proof} \rangle$$

**lemma** *expect-1-I*:

$$P x \implies 1 \leq \langle P \rangle x \\ \langle \text{proof} \rangle$$

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

*sound*  $\langle P \rangle$   
 $\langle proof \rangle$

**lemma** *embed-o[simp]*:  
 $\langle P \rangle o f = \langle P o f \rangle$   
 $\langle proof \rangle$

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

**definition** *negate* ::  $(s \Rightarrow bool) \Rightarrow s \Rightarrow bool$  ( $\mathcal{N}$ )  
**where** *negate*  $P = (\lambda s. \neg P s)$

**lemma** *negateI*:  
 $\neg P s \Longrightarrow \mathcal{N} P s$   
 $\langle proof \rangle$

**lemma** *embed-split*:  
 $f s = \langle P \rangle s * f s + \langle \mathcal{N} P \rangle s * f s$   
 $\langle proof \rangle$

**lemma** *negate-embed*:  
 $\langle \mathcal{N} P \rangle s = 1 - \langle P \rangle s$   
 $\langle proof \rangle$

**lemma** *eval-nembed-true[simp]*:  
 $P s \Longrightarrow \langle \mathcal{N} P \rangle s = 0$   
 $\langle proof \rangle$

**lemma** *eval-nembed-false[simp]*:  
 $\neg P s \Longrightarrow \langle \mathcal{N} P \rangle s = 1$   
 $\langle proof \rangle$

**lemma** *negate-Not[simp]*:  
 $\mathcal{N} Not = (\lambda x. x)$   
 $\langle proof \rangle$

**lemma** *negate-negate[simp]*:  
 $\mathcal{N} (\mathcal{N} P) = P$   
 $\langle proof \rangle$

**lemma** *embed-bool-cancel*:  
 $\langle G \rangle s * \langle \mathcal{N} G \rangle s = 0$   
 $\langle proof \rangle$

### 3.1.6 Entailment

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

**abbreviation** *entails* ::  $(s \Rightarrow real) \Rightarrow (s \Rightarrow real) \Rightarrow bool$  (-  $\Vdash$  - 50)

where  $P \Vdash Q \equiv P \leq Q$

**lemma** *entailsI*[*intro*]:

$$\llbracket \bigwedge s. P s \leq Q s \rrbracket \Longrightarrow P \Vdash Q$$

*<proof>*

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

$$P \Vdash Q \Longrightarrow P s \leq Q s$$

*<proof>*

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

$$P = Q \Longrightarrow P \Vdash Q$$

*<proof>*

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

$$\llbracket P \Vdash Q; Q \Vdash R \rrbracket \Longrightarrow P \Vdash R$$

*<proof>*

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

**lemma** *implies-entails*:

$$\llbracket \bigwedge s. P s \Longrightarrow Q s \rrbracket \Longrightarrow \langle P \rangle \Vdash \langle Q \rangle$$

*<proof>*

**lemma** *entails-implies*:

$$\bigwedge s. \llbracket \langle P \rangle \Vdash \langle Q \rangle; P s \rrbracket \Longrightarrow Q s$$

*<proof>*

### 3.1.7 Expectation Conjunction

**definition**

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

where

$$p \cdot\& q \equiv p + q \ominus I$$

**definition**

$$exp-conj :: ('s \Rightarrow real) \Rightarrow ('s \Rightarrow real) \Rightarrow ('s \Rightarrow real) \text{ (infixl } \&\& \text{ 71)}$$

where  $a \&\& b \equiv \lambda s. (a s \cdot\& b s)$

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

**lemma** *pconj-lzero*[*intro,simp*]:

$$b \leq I \Longrightarrow 0 \cdot\& b = 0$$

*<proof>*

**lemma** *pconj-rzero*[*intro,simp*]:

$$b \leq I \Longrightarrow b \cdot\& 0 = 0$$

*<proof>*

**lemma** *pconj-lone*[*intro,simp*]:

$$0 \leq b \implies 1 \text{ .\& } b = b$$

*<proof>*

**lemma** *pconj-rone*[*intro,simp*]:

$$0 \leq b \implies b \text{ .\& } 1 = b$$

*<proof>*

**lemma** *pconj-bconj*:

$$\langle\langle a \rangle\rangle s \text{ .\& } \langle\langle b \rangle\rangle s = \langle\lambda s. a \ s \ \wedge \ b \ s\rangle s$$

*<proof>*

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

$$a \text{ .\& } b = b \text{ .\& } a$$

*<proof>*

**lemma** *pconj-assoc*:

$$\llbracket 0 \leq a; a \leq 1; 0 \leq b; b \leq 1; 0 \leq c; c \leq 1 \rrbracket \implies$$

$$a \text{ .\& } (b \text{ .\& } c) = (a \text{ .\& } b) \text{ .\& } c$$

*<proof>*

**lemma** *pconj-mono*:

$$\llbracket a \leq b; c \leq d \rrbracket \implies a \text{ .\& } c \leq b \text{ .\& } d$$

*<proof>*

**lemma** *pconj-nneg*[*intro,simp*]:

$$0 \leq a \text{ .\& } b$$

*<proof>*

**lemma** *min-pconj*:

$$(\min a \ b) \text{ .\& } (\min c \ d) \leq \min (a \text{ .\& } c) (b \text{ .\& } d)$$

*<proof>*

**lemma** *pconj-less-one*[*simp*]:

$$a + b < 1 \implies a \text{ .\& } b = 0$$

*<proof>*

**lemma** *pconj-ge-one*[*simp*]:

$$1 \leq a + b \implies a \text{ .\& } b = a + b - 1$$

*<proof>*

**lemma** *pconj-idem*[*simp*]:

$$\langle\langle P \rangle\rangle s \text{ .\& } \langle\langle P \rangle\rangle s = \langle\langle P \rangle\rangle s$$

*<proof>*

### 3.1.8 Rules Involving Conjunction.

**lemma** *exp-conj-mono-left*:

$P \Vdash Q \implies P \&\& R \Vdash Q \&\& R$   
 $\langle proof \rangle$

**lemma** *exp-conj-mono-right*:  
 $Q \Vdash R \implies P \&\& Q \Vdash P \&\& R$   
 $\langle proof \rangle$

**lemma** *exp-conj-comm[ac-simps]*:  
 $a \&\& b = b \&\& a$   
 $\langle proof \rangle$

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

**lemma** *exp-conj-o-distrib[simp]*:  
 $(P \&\& Q) \text{ of } = (P \text{ of } ) \&\& (Q \text{ of } )$   
 $\langle proof \rangle$

**lemma** *exp-conj-assoc*:  
**assumes** *unitary P and unitary Q and unitary R*  
**shows**  $P \&\& (Q \&\& R) = (P \&\& Q) \&\& R$   
 $\langle proof \rangle$

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

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

**lemma** *exp-conj-idem[simp]*:  
 $\langle P \rangle \&\& \langle P \rangle = \langle P \rangle$   
 $\langle proof \rangle$

**lemma** *exp-conj-nneg[intro,simp]*:  
 $(\lambda s. 0) \leq P \&\& Q$   
 $\langle proof \rangle$

**lemma** *exp-conj-sound[intro,simp]*:  
**assumes**  $s\text{-}P$ : *sound P*  
**and**  $s\text{-}Q$ : *sound Q*  
**shows** *sound (P && Q)*  
 $\langle proof \rangle$

**lemma** *exp-conj-rzero[simp]*:

*bounded-by 1*  $P \implies P \ \&\& \ (\lambda s. 0) = (\lambda s. 0)$   
 ⟨*proof*⟩

**lemma** *exp-conj-1-right*[*simp*]:  
**assumes** *nn*: *nneg*  $A$   
**shows**  $A \ \&\& \ (\lambda-. I) = A$   
 ⟨*proof*⟩

**lemma** *exp-conj-std-split*:  
 $\llbracket \lambda s. P \ s \wedge Q \ s \rrbracket = \llbracket P \rrbracket \ \&\& \ \llbracket Q \rrbracket$   
 ⟨*proof*⟩

### 3.1.9 Rules Involving Entailment and Conjunction Together

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

**lemma** *entails-frame*:  
**assumes** *ePR*:  $P \Vdash R$   
**and** *eQS*:  $Q \Vdash S$   
**shows**  $P \ \&\& \ Q \Vdash R \ \&\& \ S$   
 ⟨*proof*⟩

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

**lemma** *pentails-cases*:  
**assumes** *PQe*:  $\bigwedge x. P \ x \Vdash Q \ x$   
**and** *exhaust*:  $\bigwedge s. \exists x. P \ (x \ s) \ s = I$   
**and** *framed*:  $\bigwedge x. P \ x \ \&\& \ R \ \Vdash \ Q \ x \ \&\& \ S$   
**and** *sR*: *sound*  $R$  **and** *sS*: *sound*  $S$   
**and** *bQ*:  $\bigwedge x. \text{bounded-by } 1 \ (Q \ x)$   
**shows**  $R \Vdash S$   
 ⟨*proof*⟩

**lemma** *unitary-bot*[*iff*]:  
*unitary*  $(\lambda s. 0::\text{real})$   
 ⟨*proof*⟩

**lemma** *unitary-top*[*iff*]:  
*unitary*  $(\lambda s. 1::\text{real})$   
 ⟨*proof*⟩

**lemma** *unitary-embed*[*iff*]:  
*unitary*  $\llbracket P \rrbracket$   
 ⟨*proof*⟩

**lemma** *unitary-const*[*iff*]:  
 $\llbracket 0 \leq c; c \leq 1 \rrbracket \implies \text{unitary } (\lambda s. c)$   
 ⟨*proof*⟩

**lemma** *unitary-mult*:

**assumes**  $uA$ : unitary  $A$  **and**  $uB$ : unitary  $B$

**shows** unitary  $(\lambda s. A s * B s)$

$\langle proof \rangle$

**lemma** *exp-conj-unitary*:

$\llbracket \text{unitary } P; \text{unitary } Q \rrbracket \implies \text{unitary } (P \&\& Q)$

$\langle proof \rangle$

**lemma** *unitary-comp* $[simp]$ :

unitary  $P \implies \text{unitary } (P \circ f)$

$\langle proof \rangle$

**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*

*minus-sound sc-sound exp-conj-sound sum-sound*

**end**

## 3.2 Expectation Transformers

**theory** *Transformers* **imports** *Expectations* **begin** **type-synonym**  $'s \text{ trans} = 's \text{ expect} \implies 's \text{ expect}$

Transformers are functions from expectations to expectations i.e.  $('s \implies \text{real}) \implies 's \implies \text{real}$ .

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

Consider again the automaton of [Figure 3.1](#). Here, the effect of executing the automaton from its initial state ( $a$ ) until it reaches some final state ( $b$  or  $c$ ) is to *transform* the expectation on final states ( $P$ ), into one on initial states, giving the *expected* value of the function on termination. Here, the transformation is linear:  $P_{\text{prior}}(a) = 0.7 * P_{\text{post}}(b) + 0.3 * P_{\text{post}}(c)$ , but this need not be the case.

Consider the automaton of [Figure 3.2](#). Here, we have extended that of [Figure 3.1](#) with two additional states,  $d$  and  $e$ , and a pair of silent (unlabelled) transitions.



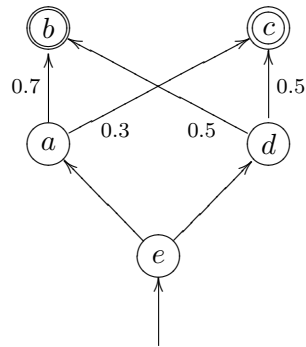


Figure 3.2: A nondeterministic-probabilistic automaton.

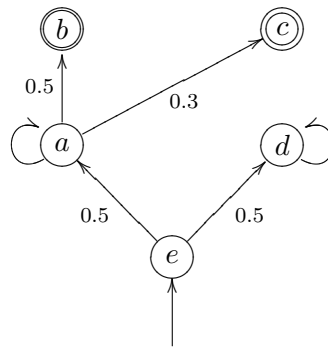


Figure 3.3: A diverging automaton.

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

Finally, [Figure 3.3](#) shows the other way in which strict sublinearity arises: divergence. This automaton transitions with probability 0.5 to state  $d$ , from which it never escapes. Once there, the probability of reaching any terminating state is zero, and thus the probability of terminating from the initial state ( $e$ ) is no higher than 0.5. If it instead takes the edge to state  $a$ , we again see a self loop, and thus in theory an infinite trace. In this case, however, every time the automaton reaches

state  $a$ , with probability  $0.5 + 0.3 = 0.8$ , it transitions to a terminating state. An infinite trace of transitions  $a \rightarrow a \rightarrow \dots$  thus has probability 0, and the automaton terminates with probability 1. We formalise such probabilistic termination arguments in [Section 4.11](#).

Having reached  $a$ , the automaton will proceed to  $b$  with probability  $0.5 * (1 / (0.5 + 0.3)) = 0.625$ , 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,  $P_{\text{prior}}(e) = 0.3125 * P_{\text{post}}(b) + 0.1875 * P_{\text{post}}(c)$ . 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 ( $\perp = (\lambda s. 0)$ ,  $\top = (\lambda s. b)$ ), and is closed under the action of feasible transformers, including  $\sqcap$  and  $\sqcup$ , which are themselves feasible. We are thus able to define both least and greatest fixed points on this set, and thus give semantics to recursive programs built from feasible components.

### 3.2.1 Comparing Transformers

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

**definition**

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

**where**

$le\text{-}trans \ t \ u \equiv \forall P. \text{ sound } P \longrightarrow t \ P \leq u \ P$

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

**definition**

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

**where**

$$le\text{-}utrans\ t\ u \longleftrightarrow (\forall P. unitary\ P \longrightarrow t\ P \leq u\ P)$$

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

$$\llbracket \bigwedge P. sound\ P \Longrightarrow t\ P \leq u\ P \rrbracket \Longrightarrow le\text{-}trans\ t\ u$$

*<proof>*

**lemma** *le-utransI[intro]*:

$$\llbracket \bigwedge P. unitary\ P \Longrightarrow t\ P \leq u\ P \rrbracket \Longrightarrow le\text{-}utrans\ t\ u$$

*<proof>*

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

$$\llbracket le\text{-}trans\ t\ u; sound\ P \rrbracket \Longrightarrow t\ P \leq u\ P$$

*<proof>*

**lemma** *le-utransD[dest]*:

$$\llbracket le\text{-}utrans\ t\ u; unitary\ P \rrbracket \Longrightarrow t\ P \leq u\ P$$

*<proof>*

**lemma** *le-trans-trans[trans]*:

$$\llbracket le\text{-}trans\ x\ y; le\text{-}trans\ y\ z \rrbracket \Longrightarrow le\text{-}trans\ x\ z$$

*<proof>*

**lemma** *le-utrans-trans[trans]*:

$$\llbracket le\text{-}utrans\ x\ y; le\text{-}utrans\ y\ z \rrbracket \Longrightarrow le\text{-}utrans\ x\ z$$

*<proof>*

**lemma** *le-trans-refl[iff]*:

$$le\text{-}trans\ x\ x$$

*<proof>*

**lemma** *le-utrans-refl[iff]*:

$$le\text{-}utrans\ x\ x$$

*<proof>*

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

$$le\text{-}trans\ t\ u \Longrightarrow le\text{-}utrans\ t\ u$$

*<proof>*

**definition**

$$l\text{-}trans :: 's\ trans \Rightarrow 's\ trans \Rightarrow bool$$

**where**

$$l\text{-}trans\ t\ u \longleftrightarrow le\text{-}trans\ t\ u \wedge \neg le\text{-}trans\ u\ t$$

Transformer equivalence is induced by comparison:

**definition**

$$equiv\text{-}trans :: 's\ trans \Rightarrow 's\ trans \Rightarrow bool$$

**where**

$$equiv\text{-}trans\ t\ u \longleftrightarrow le\text{-}trans\ t\ u \wedge le\text{-}trans\ u\ t$$

**definition**

$equiv\text{-}utrans :: 's\ trans \Rightarrow 's\ trans \Rightarrow bool$

**where**

$equiv\text{-}utrans\ t\ u \longleftrightarrow le\text{-}utrans\ t\ u \wedge le\text{-}utrans\ u\ t$

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

$\llbracket \bigwedge P. sound\ P \Longrightarrow t\ P = u\ P \rrbracket \Longrightarrow equiv\text{-}trans\ t\ u$   
 $\langle proof \rangle$

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

$\llbracket \bigwedge P. sound\ P \Longrightarrow t\ P = u\ P \rrbracket \Longrightarrow equiv\text{-}utrans\ t\ u$   
 $\langle proof \rangle$

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

$\llbracket equiv\text{-}trans\ t\ u; sound\ P \rrbracket \Longrightarrow t\ P = u\ P$   
 $\langle proof \rangle$

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

$\llbracket equiv\text{-}utrans\ t\ u; unitary\ P \rrbracket \Longrightarrow t\ P = u\ P$   
 $\langle proof \rangle$

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

$equiv\text{-}trans\ t\ t$   
 $\langle proof \rangle$

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

$equiv\text{-}utrans\ t\ t$   
 $\langle proof \rangle$

**lemma**  $le\text{-}trans\text{-}antisym$ :

$\llbracket le\text{-}trans\ x\ y; le\text{-}trans\ y\ x \rrbracket \Longrightarrow equiv\text{-}trans\ x\ y$   
 $\langle proof \rangle$

**lemma**  $le\text{-}utrans\text{-}antisym$ :

$\llbracket le\text{-}utrans\ x\ y; le\text{-}utrans\ y\ x \rrbracket \Longrightarrow equiv\text{-}utrans\ x\ y$   
 $\langle proof \rangle$

**lemma**  $equiv\text{-}trans\text{-}comm[ac\text{-}simps]$ :

$equiv\text{-}trans\ t\ u \longleftrightarrow equiv\text{-}trans\ u\ t$   
 $\langle proof \rangle$

**lemma**  $equiv\text{-}utrans\text{-}comm[ac\text{-}simps]$ :

$equiv\text{-}utrans\ t\ u \longleftrightarrow equiv\text{-}utrans\ u\ t$   
 $\langle proof \rangle$

**lemma**  $equiv\text{-}imp\text{-}le[intro]$ :

$equiv\text{-}trans\ t\ u \Longrightarrow le\text{-}trans\ t\ u$   
 $\langle proof \rangle$

**lemma** *equivu-imp-le*[*intro*]:  
 $equiv\text{-}utrans\ t\ u \implies le\text{-}utrans\ t\ u$   
 $\langle proof \rangle$

**lemma** *equiv-imp-le-alt*:  
 $equiv\text{-}trans\ t\ u \implies le\text{-}trans\ u\ t$   
 $\langle proof \rangle$

**lemma** *equiv-uimp-le-alt*:  
 $equiv\text{-}utrans\ t\ u \implies le\text{-}utrans\ u\ t$   
 $\langle proof \rangle$

**lemma** *le-trans-equiv-rsp*[*simp*]:  
 $equiv\text{-}trans\ t\ u \implies le\text{-}trans\ t\ v \longleftrightarrow le\text{-}trans\ u\ v$   
 $\langle proof \rangle$

**lemma** *le-utrans-equiv-rsp*[*simp*]:  
 $equiv\text{-}utrans\ t\ u \implies le\text{-}utrans\ t\ v \longleftrightarrow le\text{-}utrans\ u\ v$   
 $\langle proof \rangle$

**lemma** *equiv-trans-le-trans*[*trans*]:  
 $\llbracket equiv\text{-}trans\ t\ u; le\text{-}trans\ u\ v \rrbracket \implies le\text{-}trans\ t\ v$   
 $\langle proof \rangle$

**lemma** *equiv-utrans-le-utrans*[*trans*]:  
 $\llbracket equiv\text{-}utrans\ t\ u; le\text{-}utrans\ u\ v \rrbracket \implies le\text{-}utrans\ t\ v$   
 $\langle proof \rangle$

**lemma** *le-trans-equiv-rsp-right*[*simp*]:  
 $equiv\text{-}trans\ t\ u \implies le\text{-}trans\ v\ t \longleftrightarrow le\text{-}trans\ v\ u$   
 $\langle proof \rangle$

**lemma** *le-utrans-equiv-rsp-right*[*simp*]:  
 $equiv\text{-}utrans\ t\ u \implies le\text{-}utrans\ v\ t \longleftrightarrow le\text{-}utrans\ v\ u$   
 $\langle proof \rangle$

**lemma** *le-trans-equiv-trans*[*trans*]:  
 $\llbracket le\text{-}trans\ t\ u; equiv\text{-}trans\ u\ v \rrbracket \implies le\text{-}trans\ t\ v$   
 $\langle proof \rangle$

**lemma** *le-utrans-equiv-utrans*[*trans*]:  
 $\llbracket le\text{-}utrans\ t\ u; equiv\text{-}utrans\ u\ v \rrbracket \implies le\text{-}utrans\ t\ v$   
 $\langle proof \rangle$

**lemma** *equiv-trans-trans*[*trans*]:  
**assumes** *xy*:  $equiv\text{-}trans\ x\ y$   
**and** *yz*:  $equiv\text{-}trans\ y\ z$   
**shows**  $equiv\text{-}trans\ x\ z$

$\langle proof \rangle$

**lemma** *equiv-utrans-trans*[*trans*]:

**assumes** *xy*: *equiv-utrans x y*

**and** *yz*: *equiv-utrans y z*

**shows** *equiv-utrans x z*

$\langle proof \rangle$

**lemma** *equiv-trans-equiv-utrans*[*dest*]:

*equiv-trans t u*  $\implies$  *equiv-utrans t u*

$\langle proof \rangle$

### 3.2.2 Healthy Transformers

#### Feasibility

**definition** *feasible* ::  $((a \Rightarrow real) \Rightarrow (a \Rightarrow real)) \Rightarrow bool$

**where** *feasible t*  $\longleftrightarrow (\forall P b. \text{bounded-by } b P \wedge \text{nneg } P \longrightarrow$   
 $\text{bounded-by } b (t P) \wedge \text{nneg } (t P))$

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

**lemma** *feasibleI*[*intro*]:

$\llbracket \bigwedge b P. \llbracket \text{bounded-by } b P; \text{nneg } P \rrbracket \implies \text{bounded-by } b (t P);$

$\llbracket \bigwedge b P. \llbracket \text{bounded-by } b P; \text{nneg } P \rrbracket \implies \text{nneg } (t P) \rrbracket \implies \text{feasible } t$

$\langle proof \rangle$

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

$\llbracket \text{feasible } t; \text{bounded-by } b P; \text{nneg } P \rrbracket \implies \text{bounded-by } b (t P)$

$\langle proof \rangle$

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

$\llbracket \text{feasible } t; \text{bounded-by } b P; \text{nneg } P \rrbracket \implies \text{nneg } (t P)$

$\langle proof \rangle$

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

$\llbracket \text{feasible } t; \text{sound } P \rrbracket \implies \text{sound } (t P)$

$\langle proof \rangle$

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

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

**assumes** *ft*: *feasible t*

**shows**  $t (\lambda x. 0) = (\lambda x. 0)$

$\langle proof \rangle$

**lemma** *feasible-id*:

*feasible* ( $\lambda x. x$ )  
 $\langle \text{proof} \rangle$

**lemma** *feasible-bounded-by*[*dest*]:  
 $\llbracket \text{feasible } t; \text{sound } P; \text{bounded-by } b \ P \rrbracket \implies \text{bounded-by } b \ (t \ P)$   
 $\langle \text{proof} \rangle$

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

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

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

## Monotonicity

**definition**  
 $\text{mono-trans} :: ((s \Rightarrow \text{real}) \Rightarrow (s \Rightarrow \text{real})) \Rightarrow \text{bool}$   
**where**  
 $\text{mono-trans } t \equiv \forall P \ Q. (\text{sound } P \wedge \text{sound } Q \wedge P \leq Q) \longrightarrow t \ P \leq t \ Q$

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

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

**lemma** *mono-transI*[*intro*]:  
 $\llbracket \bigwedge P \ Q. \llbracket \text{sound } P; \text{sound } Q; P \leq Q \rrbracket \implies t \ P \leq t \ Q \rrbracket \implies \text{mono-trans } t$   
 $\langle \text{proof} \rangle$

**lemma** *mono-transD*[*dest*]:  
 $\llbracket \text{mono-trans } t; \text{sound } P; \text{sound } Q; P \leq Q \rrbracket \implies t \ P \leq t \ Q$   
 $\langle \text{proof} \rangle$

## Scaling

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

### definition

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

### where

$scaling\ t \equiv \forall P\ c\ x. sound\ P \wedge 0 \leq c \longrightarrow c * t\ P\ x = t\ (\lambda x. c * P\ x)\ x$

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

### lemma *scalingI*[intro]:

$\llbracket \bigwedge P\ c\ x. \llbracket sound\ P; 0 \leq c \rrbracket \implies c * t\ P\ x = t\ (\lambda x. c * P\ x)\ x \rrbracket \implies scaling\ t$   
 $\langle proof \rangle$

### lemma *scalingD*[dest]:

$\llbracket scaling\ t; sound\ P; 0 \leq c \rrbracket \implies c * t\ P\ x = t\ (\lambda x. c * P\ x)\ x$   
 $\langle proof \rangle$

### lemma *right-scalingD*:

**assumes**  $st: scaling\ t$

**and**  $sP: sound\ P$

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

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

$\langle proof \rangle$

## Healthiness

Healthy transformers are feasible and monotonic, and respect scaling

### definition

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

### where

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

### lemma *healthyI*[intro]:

$\llbracket feasible\ t; mono-trans\ t; scaling\ t \rrbracket \implies healthy\ t$   
 $\langle proof \rangle$

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

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

$healthy\ t \implies mono-trans\ t$

$\langle proof \rangle$



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

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

*healthy t*  $\implies$  *feasible t*  
 $\langle$ *proof* $\rangle$

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

*healthy t*  $\implies$  *scaling t*  
 $\langle$ *proof* $\rangle$

**lemma** *healthy-bounded-byD*[*intro*]:

$\llbracket$  *healthy t*; *bounded-by b P*; *nneg P*  $\rrbracket \implies$  *bounded-by b (t P)*  
 $\langle$ *proof* $\rangle$

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

$\llbracket$  *healthy t*; *bounded-by b P*; *sound P*  $\rrbracket \implies$  *bounded-by b (t P)*  
 $\langle$ *proof* $\rangle$

**lemma** *healthy-boundedD*[*dest,simp*]:

$\llbracket$  *healthy t*; *sound P*  $\rrbracket \implies$  *bounded (t P)*  
 $\langle$ *proof* $\rangle$

**lemma** *healthy-nnegD*[*dest,simp*]:

$\llbracket$  *healthy t*; *sound P*  $\rrbracket \implies$  *nneg (t P)*  
 $\langle$ *proof* $\rangle$

**lemma** *healthy-nnegD2*[*dest,simp*]:

$\llbracket$  *healthy t*; *bounded-by b P*; *nneg P*  $\rrbracket \implies$  *nneg (t P)*  
 $\langle$ *proof* $\rangle$

**lemma** *healthy-sound*[*intro*]:

$\llbracket$  *healthy t*; *sound P*  $\rrbracket \implies$  *sound (t P)*  
 $\langle$ *proof* $\rangle$

**lemma** *healthy-unitary*[*intro*]:

$\llbracket$  *healthy t*; *unitary P*  $\rrbracket \implies$  *unitary (t P)*  
 $\langle$ *proof* $\rangle$

**lemma** *healthy-id*[*simp,intro!*]:

*healthy id*  
 $\langle$ *proof* $\rangle$

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

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

**lemma** *le-trans-bot*[*intro,simp*]:

*healthy t*  $\implies$  *le-trans* ( $\lambda P$  *s. 0*) *t*  
 $\langle$ *proof* $\rangle$

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

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

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

**lemma** *healthy-equivI*:  
**fixes**  $t::('s \Rightarrow real) \Rightarrow 's \Rightarrow real$  **and**  $u$   
**assumes** *equiv*:  $equiv\text{-trans}\ t\ u$   
**and** *healthy*:  $healthy\ t$   
**shows**  $healthy\ u$   
 $\langle proof \rangle$

**lemma** *healthy-equiv*:  
 $equiv\text{-trans}\ t\ u \implies healthy\ t \longleftrightarrow healthy\ u$   
 $\langle proof \rangle$

**lemma** *healthy-scale*:  
**fixes**  $t::('s \Rightarrow real) \Rightarrow 's \Rightarrow real$   
**assumes** *ht*:  $healthy\ t$  **and** *nc*:  $0 \leq c$  **and** *bc*:  $c \leq 1$   
**shows**  $healthy\ (\lambda P\ s.\ c * t\ P\ s)$   
 $\langle proof \rangle$

**lemma** *healthy-top*[*iff*]:  
 $healthy\ (\lambda P\ s.\ bound\text{-of}\ P)$   
 $\langle proof \rangle$

**lemma** *healthy-bot*[*iff*]:  
 $healthy\ (\lambda P\ s.\ 0)$   
 $\langle proof \rangle$

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

**definition**

$nearly\text{-healthy} :: (('s \Rightarrow real) \Rightarrow ('s \Rightarrow real)) \Rightarrow bool$   
**where**  
 $nearly\text{-healthy}\ t \longleftrightarrow (\forall P.\ unitary\ P \longrightarrow unitary\ (t\ P)) \wedge$   
 $(\forall P\ Q.\ unitary\ P \longrightarrow unitary\ Q \longrightarrow P \Vdash Q \longrightarrow t\ P \Vdash t\ Q)$

**lemma** *nearly-healthyI*[*intro*]:  
 $\llbracket \bigwedge P.\ unitary\ P \implies unitary\ (t\ P);$   
 $\bigwedge P\ Q.\ \llbracket unitary\ P; unitary\ Q; P \Vdash Q \rrbracket \implies t\ P \Vdash t\ Q \rrbracket \implies nearly\text{-healthy}\ t$   
 $\langle proof \rangle$

**lemma** *nearly-healthy-monoD*[*dest*]:  
 $\llbracket \text{nearly-healthy } t; P \Vdash Q; \text{unitary } P; \text{unitary } Q \rrbracket \Longrightarrow t P \Vdash t Q$   
 $\langle \text{proof} \rangle$

**lemma** *nearly-healthy-unitaryD*[*dest*]:  
 $\llbracket \text{nearly-healthy } t; \text{unitary } P \rrbracket \Longrightarrow \text{unitary } (t P)$   
 $\langle \text{proof} \rangle$

**lemma** *healthy-nearly-healthy*[*dest*]:  
**assumes** *ht*: *healthy* *t*  
**shows** *nearly-healthy* *t*  
 $\langle \text{proof} \rangle$

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

### 3.2.3 Sublinearity

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

**definition** *sublinear* ::  
 $((s \Rightarrow \text{real}) \Rightarrow (s \Rightarrow \text{real})) \Rightarrow \text{bool}$   
**where**  
 $\text{sublinear } t \iff (\forall a b c P Q s. (\text{sound } P \wedge \text{sound } Q \wedge 0 \leq a \wedge 0 \leq b \wedge 0 \leq c) \longrightarrow$   
 $a * t P s + b * t Q s \ominus c$   
 $\leq t (\lambda s'. a * P s' + b * Q s' \ominus c) s)$

**lemma** *sublinearI*[*intro*]:  
 $\llbracket \bigwedge a b c P Q s. \llbracket \text{sound } P; \text{sound } Q; 0 \leq a; 0 \leq b; 0 \leq c \rrbracket \Longrightarrow$   
 $a * t P s + b * t Q s \ominus c \leq$   
 $t (\lambda s'. a * P s' + b * Q s' \ominus c) s \rrbracket \Longrightarrow \text{sublinear } t$   
 $\langle \text{proof} \rangle$

**lemma** *sublinearD*[*dest*]:  
 $\llbracket \text{sublinear } t; \text{sound } P; \text{sound } Q; 0 \leq a; 0 \leq b; 0 \leq c \rrbracket \Longrightarrow$   
 $a * t P s + b * t Q s \ominus c \leq$   
 $t (\lambda s'. a * P s' + b * Q s' \ominus c) s$   
 $\langle \text{proof} \rangle$

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

#### Sub-additivity

**definition** *sub-add* ::

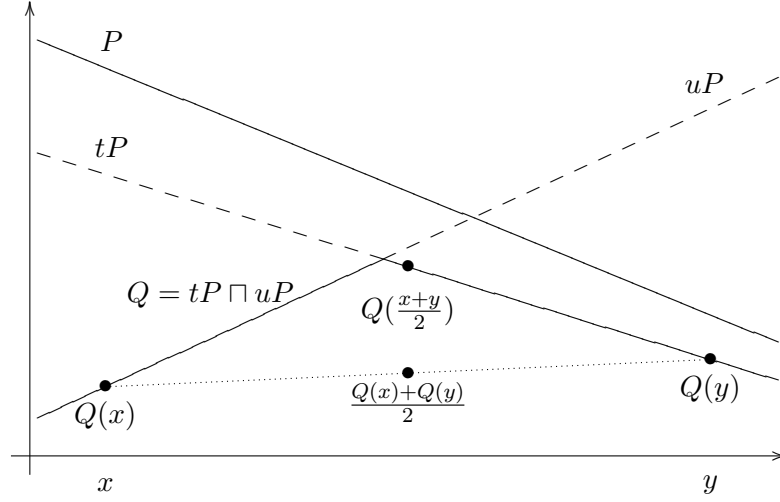


Figure 3.4: A graphical depiction of sub-additivity as convexity.

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

where

$$\text{sub-add } t \iff (\forall P Q s. (\text{sound } P \wedge \text{sound } Q) \longrightarrow t P s + t Q s \leq t (\lambda s'. P s' + Q s') s)$$

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

Here  $P$  is an affine function (expectation)  $\text{real} \Rightarrow \text{real}$ , restricted to some finite interval. In practice the state space (the left-hand type) is typically discrete and multi-dimensional, but on the reals we have a convenient geometrical intuition. The lines  $tP$  and  $uP$  represent the effect of two healthy transformers (again affine). Neither monotonicity nor scaling are represented, but both are feasible: Both lines are bounded above by the greatest value of  $P$ .

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

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

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

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

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

Can be rewritten, using scaling, to:

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

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

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

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

$$\begin{aligned} & \llbracket \bigwedge P Q s. \llbracket \text{sound } P; \text{sound } Q \rrbracket \implies \\ & \quad t P s + t Q s \leq t (\lambda s'. P s' + Q s') s \rrbracket \implies \text{sub-add } t \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *sub-addI2*:

$$\begin{aligned} & \llbracket \bigwedge P Q. \llbracket \text{sound } P; \text{sound } Q \rrbracket \implies \\ & \quad \lambda s. t P s + t Q s \Vdash t (\lambda s. P s + Q s) \rrbracket \implies \\ & \quad \text{sub-add } t \\ & \langle \text{proof} \rangle \end{aligned}$$

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

$$\begin{aligned} & \llbracket \text{sub-add } t; \text{sound } P; \text{sound } Q \rrbracket \implies t P s + t Q s \leq t (\lambda s'. P s' + Q s') s \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *equiv-sub-add*:

$$\begin{aligned} & \text{fixes } t::('s \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real} \\ & \text{assumes } eq: \text{equiv-trans } t \ u \\ & \quad \text{and } sa: \text{sub-add } t \\ & \text{shows } \text{sub-add } u \\ & \langle \text{proof} \rangle \end{aligned}$$

Sublinearity and feasibility imply sub-additivity.

**lemma** *sublinear-subadd*:

$$\begin{aligned} & \text{fixes } t::('s \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real} \\ & \text{assumes } slt: \text{sublinear } t \\ & \quad \text{and } ft: \text{feasible } t \\ & \text{shows } \text{sub-add } t \\ & \langle \text{proof} \rangle \end{aligned}$$

A few properties following from sub-additivity:

**lemma** *standard-negate*:

$$\begin{aligned} & \text{assumes } ht: \text{healthy } t \\ & \quad \text{and } sat: \text{sub-add } t \end{aligned}$$

**shows**  $t \ll P \gg s + t \ll \mathcal{N} P \gg s \leq I$   
 $\langle proof \rangle$

**lemma** *sub-add-sum*:

**fixes**  $t::'s$  trans **and**  $S::'a$  set  
**assumes**  $sat$ : sub-add  $t$   
**and**  $ht$ : healthy  $t$   
**and**  $sP$ :  $\bigwedge x. sound (P x)$   
**shows**  $(\lambda x. \sum y \in S. t (P y) x) \leq t (\lambda x. \sum y \in S. P y x)$   
 $\langle proof \rangle$

**lemma** *sub-add-guard-split*:

**fixes**  $t::'s::finite$  trans **and**  $P::'s$  expect **and**  $s::'s$   
**assumes**  $sat$ : sub-add  $t$   
**and**  $ht$ : healthy  $t$   
**and**  $sP$ : sound  $P$   
**shows**  $(\sum y \in \{s. G s\}. P y * t \ll \lambda z. z = y \gg s) +$   
 $(\sum y \in \{s. \neg G s\}. P y * t \ll \lambda z. z = y \gg s) \leq t P s$   
 $\langle proof \rangle$

### Sub-distributivity

**definition** *sub-distrib* ::

$((s \Rightarrow real) \Rightarrow (s \Rightarrow real)) \Rightarrow bool$

**where**

$sub-distrib t \longleftrightarrow (\forall P s. sound P \longrightarrow t P s \ominus I \leq t (\lambda s'. P s' \ominus I) s)$

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

$\llbracket \bigwedge P s. sound P \Longrightarrow t P s \ominus I \leq t (\lambda s'. P s' \ominus I) s \rrbracket \Longrightarrow sub-distrib t$   
 $\langle proof \rangle$

**lemma** *sub-distribI2*:

$\llbracket \bigwedge P. sound P \Longrightarrow \lambda s. t P s \ominus I \Vdash t (\lambda s. P s \ominus I) \rrbracket \Longrightarrow sub-distrib t$   
 $\langle proof \rangle$

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

$\llbracket sub-distrib t; sound P \rrbracket \Longrightarrow t P s \ominus I \leq t (\lambda s'. P s' \ominus I) s$   
 $\langle proof \rangle$

**lemma** *equiv-sub-distrib*:

**fixes**  $t::(s \Rightarrow real) \Rightarrow 's \Rightarrow real$   
**assumes**  $eq$ : equiv-trans  $t u$   
**and**  $sd$ : sub-distrib  $t$   
**shows** sub-distrib  $u$   
 $\langle proof \rangle$

Sublinearity implies sub-distributivity:

**lemma** *sublinear-sub-distrib*:

**fixes**  $t::(s \Rightarrow real) \Rightarrow 's \Rightarrow real$

**assumes**  $slt$ : *sublinear*  $t$   
**shows** *sub-distrib*  $t$   
 $\langle proof \rangle$

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

**lemma** *sd-sa-sublinear*:  
**fixes**  $t$ ::( $'s \Rightarrow real$ )  $\Rightarrow 's \Rightarrow real$   
**assumes**  $sdt$ : *sub-distrib*  $t$  **and**  $sat$ : *sub-add*  $t$  **and**  $ht$ : *healthy*  $t$   
**shows** *sublinear*  $t$   
 $\langle proof \rangle$

### Sub-conjunctivity

**definition**  
 $sub-conj :: ('s \Rightarrow real) \Rightarrow 's \Rightarrow real \Rightarrow bool$   
**where**  
 $sub-conj\ t \equiv \forall P\ Q. (sound\ P \wedge sound\ Q) \longrightarrow$   
 $t\ P \ \&\&\ t\ Q \Vdash t\ (P \ \&\&\ Q)$

**lemma** *sub-conjI[intro]*:  
 $\llbracket \bigwedge P\ Q. \llbracket sound\ P; sound\ Q \rrbracket \implies$   
 $t\ P \ \&\&\ t\ Q \Vdash t\ (P \ \&\&\ Q) \rrbracket \implies sub-conj\ t$   
 $\langle proof \rangle$

**lemma** *sub-conjD[dest]*:  
 $\llbracket sub-conj\ t; sound\ P; sound\ Q \rrbracket \implies t\ P \ \&\&\ t\ Q \Vdash t\ (P \ \&\&\ Q)$   
 $\langle proof \rangle$

**lemma** *sub-conj-wp-twice*:  
**fixes**  $f$ :: $'s \Rightarrow ('s \Rightarrow real) \Rightarrow 's \Rightarrow real$   
**assumes**  $all$ :  $\forall s. sub-conj\ (f\ s)$   
**shows**  $sub-conj\ (\lambda P\ s. f\ s\ P\ s)$   
 $\langle proof \rangle$

Sublinearity implies sub-conjunctivity:

**lemma** *sublinear-sub-conj*:  
**fixes**  $t$ ::( $'s \Rightarrow real$ )  $\Rightarrow 's \Rightarrow real$   
**assumes**  $slt$ : *sublinear*  $t$   
**shows** *sub-conj*  $t$   
 $\langle proof \rangle$

### Sublinearity under equivalence

Sublinearity is preserved by equivalence.

**lemma** *equiv-sublinear*:  
 $\llbracket equiv-trans\ t\ u; sublinear\ t; healthy\ t \rrbracket \implies sublinear\ u$   
 $\langle proof \rangle$

### 3.2.4 Determinism

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

#### Additivity

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

##### definition

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

##### where

*additive*  $t \equiv \forall P Q. (\text{sound } P \wedge \text{sound } Q) \longrightarrow$   
 $t (\lambda s. P s + Q s) = (\lambda s. t P s + t Q s)$

##### lemma *additiveD*:

$\llbracket \text{additive } t; \text{sound } P; \text{sound } Q \rrbracket \Longrightarrow t (\lambda s. P s + Q s) = (\lambda s. t P s + t Q s)$   
 $\langle \text{proof} \rangle$

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

$\llbracket \bigwedge P Q s. \llbracket \text{sound } P; \text{sound } Q \rrbracket \Longrightarrow t (\lambda s. P s + Q s) s = t P s + t Q s \rrbracket \Longrightarrow$   
 $\text{additive } t$   
 $\langle \text{proof} \rangle$

Additivity is strictly stronger than sub-additivity.

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

*additive*  $t \Longrightarrow \text{sub-add } t$   
 $\langle \text{proof} \rangle$

The additivity property extends to finite summation.

##### lemma *additive-sum*:

**fixes**  $S::'s \text{ set}$

**assumes** *additive*: *additive*  $t$

**and** *healthy*: *healthy*  $t$

**and** *finite*: *finite*  $S$

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

**shows**  $t (\lambda x. \sum_{y \in S}. P y x) = (\lambda x. \sum_{y \in S}. t (P y) x)$   
 $\langle \text{proof} \rangle$

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**  $t::('s:\text{finite} \Rightarrow \text{real}) \Rightarrow 's \Rightarrow \text{real}$

**assumes** *additive*: *additive*  $t$

**and**  $ht$ : *healthy*  $t$



**and**  $sP$ : *sound*  $P$   
**shows**  $t P x = (\sum_{y \in UNIV}. P y * t \ll \lambda z. z = y \gg x)$   
 $\langle proof \rangle$

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**  $t P x = (\sum_{y \in \{s. G s\}}. P y * t \ll \lambda z. z = y \gg x) +$   
 $(\sum_{y \in \{s. \neg G s\}}. P y * t \ll \lambda z. z = y \gg x)$   
 $\langle proof \rangle$

### Maximality

**definition**  
 $maximal :: (('a \Rightarrow real) \Rightarrow 'a \Rightarrow real) \Rightarrow bool$   
**where**  
 $maximal t \equiv \forall c. 0 \leq c \longrightarrow t (\lambda-. c) = (\lambda-. c)$

**lemma** *maximalI[intro]*:  
 $\llbracket \bigwedge c. 0 \leq c \Longrightarrow t (\lambda-. c) = (\lambda-. c) \rrbracket \Longrightarrow maximal t$   
 $\langle proof \rangle$

**lemma** *maximalD[dest]*:  
 $\llbracket maximal t; 0 \leq c \rrbracket \Longrightarrow t (\lambda-. c) = (\lambda-. c)$   
 $\langle proof \rangle$

A transformer that is both additive and maximal is deterministic:

**definition** *determ* ::  $(( 'a \Rightarrow real) \Rightarrow 'a \Rightarrow real) \Rightarrow bool$   
**where**  
 $determ t \equiv additive t \wedge maximal t$

**lemma** *determI[intro]*:  
 $\llbracket additive t; maximal t \rrbracket \Longrightarrow determ t$   
 $\langle proof \rangle$

**lemma** *determ-additiveD[intro]*:  
 $determ t \Longrightarrow additive t$   
 $\langle proof \rangle$

**lemma** *determ-maximalD[intro]*:  
 $determ t \Longrightarrow maximal t$   
 $\langle proof \rangle$

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

**lemma** *determ-negate*:  
**assumes** *determ*: *determ t*  
**shows**  $t \ll P \gg s + t \ll \mathcal{N} P \gg s = 1$   
 $\langle proof \rangle$

### 3.2.5 Modular Reasoning

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

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

**lemma** *entails-combine*:  
**assumes** *wp1*:  $P \Vdash t R$   
**and** *wp2*:  $Q \Vdash t S$   
**and** *sc*: *sub-conj t*  
**and** *sR*: *sound R*  
**and** *sS*: *sound S*  
**shows**  $P \&\& Q \Vdash t (R \&\& S)$   
 $\langle proof \rangle$

These allow mismatched results to be composed

**lemma** *entails-strengthen-post*:  
 $\llbracket P \Vdash t Q; \text{healthy } t; \text{sound } R; Q \Vdash R; \text{sound } Q \rrbracket \implies P \Vdash t R$   
 $\langle proof \rangle$

**lemma** *entails-weaken-pre*:  
 $\llbracket Q \Vdash t R; P \Vdash Q \rrbracket \implies P \Vdash t R$   
 $\langle proof \rangle$

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

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

### 3.2.6 Transforming Standard Expectations

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

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

**lemma** *use-premise*:

**assumes**  $h$ : *healthy*  $t$  **and**  $wP$ :  $\bigwedge s. P\ s \implies I \leq t \ll Q \gg s$   
**shows**  $\ll P \gg \Vdash t \ll Q \gg$   
 $\langle proof \rangle$

The other direction works too.

**lemma** *fold-premise*:  
**assumes**  $ht$ : *healthy*  $t$   
**and**  $wP$ :  $\ll P \gg \Vdash t \ll Q \gg$   
**shows**  $\forall s. P\ s \longrightarrow I \leq t \ll Q \gg s$   
 $\langle proof \rangle$

Predicate conjunction behaves as expected:

**lemma** *conj-post*:  
 $\ll P \Vdash t \ll \lambda s. Q\ s \wedge R\ s \gg; \text{healthy } t \gg \implies P \Vdash t \ll Q \gg$   
 $\langle proof \rangle$

Similar to  $\ll \text{healthy } ?t; \bigwedge s. ?P\ s \implies I \leq ?t \ll ?Q \gg s \gg \implies \ll ?P \gg \Vdash ?t \ll ?Q \gg$ , but more general.

**lemma** *entails-pconj-assumption*:  
**assumes**  $f$ : *feasible*  $t$  **and**  $wP$ :  $\bigwedge s. P\ s \implies Q\ s \leq t\ R\ s$   
**and**  $uQ$ : *unitary*  $Q$  **and**  $uR$ : *unitary*  $R$   
**shows**  $\ll P \gg \&\& Q \Vdash t\ R$   
 $\langle proof \rangle$

**end**

## 3.3 Induction

**theory** *Induction*  
**imports** *Expectations Transformers*  
**begin**

### 3.3.1 The Lattice of Expectations

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

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

This also works semantically, thanks to the definition of healthiness. Given a healthy transformer parameterised by some sound expectation:  $t$ . Imagine that we

wish to find the least fixed point of  $t P$ . In practice,  $t$  is generally doubly healthy, that is  $\forall P. \text{sound } P \longrightarrow \text{healthy } (t P)$  and  $\forall Q. \text{sound } Q \longrightarrow \text{healthy } (\lambda P. t P Q)$ . Thus by feasibility,  $t P Q$  must be bounded by  $\text{bound-of } P$ . Thus, as by definition  $x \leq t P x$  for any fixed point, all must lie in the set of sound expectations bounded above by  $\lambda \cdot \text{bound-of } P$ .

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

**lemma** *Inf-exp-lower*:

$\llbracket P \in S; \forall P \in S. \text{nneg } P \rrbracket \Longrightarrow \text{Inf-exp } S \leq P$   
 $\langle \text{proof} \rangle$

**lemma** *Inf-exp-greatest*:

$\llbracket S \neq \{\}; \forall P \in S. Q \leq P \rrbracket \Longrightarrow Q \leq \text{Inf-exp } S$   
 $\langle \text{proof} \rangle$

**definition** *Sup-exp* :: 's expect set  $\Rightarrow$  's expect

**where** *Sup-exp*  $S = (\text{if } S = \{\} \text{ then } \lambda s. 0 \text{ else } (\lambda s. \text{Sup } \{f s \mid f. f \in S\}))$

**lemma** *Sup-exp-upper*:

$\llbracket P \in S; \forall P \in S. \text{bounded-by } b P \rrbracket \Longrightarrow P \leq \text{Sup-exp } S$   
 $\langle \text{proof} \rangle$

**lemma** *Sup-exp-least*:

$\llbracket \forall P \in S. P \leq Q; \text{nneg } Q \rrbracket \Longrightarrow \text{Sup-exp } S \leq Q$   
 $\langle \text{proof} \rangle$

**lemma** *Sup-exp-sound*:

**assumes**  $sS: \bigwedge P. P \in S \Longrightarrow \text{sound } P$   
**and**  $bS: \bigwedge P. P \in S \Longrightarrow \text{bounded-by } b P$   
**shows**  $\text{sound } (\text{Sup-exp } S)$   
 $\langle \text{proof} \rangle$

**definition** *lfp-exp* :: 's trans  $\Rightarrow$  's expect

**where** *lfp-exp*  $t = \text{Inf-exp } \{P. \text{sound } P \wedge t P \leq P\}$

**lemma** *lfp-exp-lowerbound*:

$\llbracket t P \leq P; \text{sound } P \rrbracket \Longrightarrow \text{lfp-exp } t \leq P$   
 $\langle \text{proof} \rangle$

**lemma** *lfp-exp-greatest*:

$\llbracket \bigwedge P. \llbracket t P \leq P; \text{sound } P \rrbracket \Longrightarrow Q \leq P; \text{sound } Q; t R \Vdash R; \text{sound } R \rrbracket \Longrightarrow Q \leq \text{lfp-exp } t$   
 $\langle \text{proof} \rangle$

**lemma** *feasible-lfp-exp-sound*:

$\text{feasible } t \Longrightarrow \text{sound } (\text{lfp-exp } t)$   
 $\langle \text{proof} \rangle$

**lemma** *lfp-exp-sound*:

**assumes**  $fR: t R \Vdash R$  **and**  $sR: \text{sound } R$

**shows**  $\text{sound } (\text{lfp-exp } t)$

$\langle \text{proof} \rangle$

**lemma** *lfp-exp-bound*:

$(\bigwedge P. \text{unitary } P \implies \text{unitary } (t P)) \implies \text{bounded-by } 1 (\text{lfp-exp } t)$

$\langle \text{proof} \rangle$

**lemma** *lfp-exp-unitary*:

$(\bigwedge P. \text{unitary } P \implies \text{unitary } (t P)) \implies \text{unitary } (\text{lfp-exp } t)$

$\langle \text{proof} \rangle$

**lemma** *lfp-exp-lemma2*:

**fixes**  $t:: 's \text{ trans}$

**assumes**  $st: \bigwedge P. \text{sound } P \implies \text{sound } (t P)$

**and**  $mt: \text{mono-trans } t$

**and**  $fR: t R \Vdash R$  **and**  $sR: \text{sound } R$

**shows**  $t (\text{lfp-exp } t) \leq \text{lfp-exp } t$

$\langle \text{proof} \rangle$

**lemma** *lfp-exp-lemma3*:

**assumes**  $st: \bigwedge P. \text{sound } P \implies \text{sound } (t P)$

**and**  $mt: \text{mono-trans } t$

**and**  $fR: t R \Vdash R$  **and**  $sR: \text{sound } R$

**shows**  $\text{lfp-exp } t \leq t (\text{lfp-exp } t)$

$\langle \text{proof} \rangle$

**lemma** *lfp-exp-unfold*:

**assumes**  $nt: \bigwedge P. \text{sound } P \implies \text{sound } (t P)$

**and**  $mt: \text{mono-trans } t$

**and**  $fR: t R \Vdash R$  **and**  $sR: \text{sound } R$

**shows**  $\text{lfp-exp } t = t (\text{lfp-exp } t)$

$\langle \text{proof} \rangle$

**definition**  $\text{gfp-exp} :: 's \text{ trans} \Rightarrow 's \text{ expect}$

**where**  $\text{gfp-exp } t = \text{Sup-exp } \{P. \text{unitary } P \wedge P \leq t P\}$

**lemma** *gfp-exp-upperbound*:

$\llbracket P \leq t P; \text{unitary } P \rrbracket \implies P \leq \text{gfp-exp } t$

$\langle \text{proof} \rangle$

**lemma** *gfp-exp-least*:

$\llbracket \bigwedge P. \llbracket P \leq t P; \text{unitary } P \rrbracket \implies P \leq Q; \text{unitary } Q \rrbracket \implies \text{gfp-exp } t \leq Q$

$\langle \text{proof} \rangle$

**lemma** *gfp-exp-bound*:

$(\bigwedge P. \text{unitary } P \implies \text{unitary } (t P)) \implies \text{bounded-by } 1 (\text{gfp-exp } t)$

$\langle \text{proof} \rangle$

**lemma** *gfp-exp-nneg*[*iff*]:

$nneg (gfp\text{-}exp\ t)$   
 $\langle proof \rangle$

**lemma** *gfp-exp-unitary*:

$(\bigwedge P. unitary\ P \implies unitary\ (t\ P)) \implies unitary\ (gfp\text{-}exp\ t)$   
 $\langle proof \rangle$

**lemma** *gfp-exp-lemma2*:

**assumes** *ft*:  $\bigwedge P. unitary\ P \implies unitary\ (t\ P)$   
**and** *mt*:  $\bigwedge P\ Q. \llbracket unitary\ P; unitary\ Q; P \Vdash Q \rrbracket \implies t\ P \Vdash t\ Q$   
**shows**  $gfp\text{-}exp\ t \leq t (gfp\text{-}exp\ t)$   
 $\langle proof \rangle$

**lemma** *gfp-exp-lemma3*:

**assumes** *ft*:  $\bigwedge P. unitary\ P \implies unitary\ (t\ P)$   
**and** *mt*:  $\bigwedge P\ Q. \llbracket unitary\ P; unitary\ Q; P \Vdash Q \rrbracket \implies t\ P \Vdash t\ Q$   
**shows**  $t (gfp\text{-}exp\ t) \leq gfp\text{-}exp\ t$   
 $\langle proof \rangle$

**lemma** *gfp-exp-unfold*:

$(\bigwedge P. unitary\ P \implies unitary\ (t\ P)) \implies (\bigwedge P\ Q. \llbracket unitary\ P; unitary\ Q; P \Vdash Q \rrbracket \implies t\ P \Vdash t\ Q) \implies$   
 $gfp\text{-}exp\ t = t (gfp\text{-}exp\ t)$   
 $\langle proof \rangle$

### 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  $\lambda\ -. \ 0::'c$ , and the top element is the greatest allowed by feasibility:  $\lambda\ P\ -. \ bound\text{-}of\ P$ .

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

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

**lemma** *Inf-trans-lower*:

$\llbracket t \in S; \forall u \in S. \forall P. sound\ P \longrightarrow sound\ (u\ P) \rrbracket \implies le\text{-}trans\ (Inf\text{-}trans\ S)\ t$   
 $\langle proof \rangle$

**lemma** *Inf-trans-greatest*:

$\llbracket S \neq \{\}; \forall t \in S. \forall P. le\text{-}trans\ u\ t \rrbracket \implies le\text{-}trans\ u\ (Inf\text{-}trans\ S)$   
 $\langle proof \rangle$

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

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

**lemma** *Sup-trans-upper*:

$\llbracket t \in S; \forall u \in S. \forall P. \text{unitary } P \longrightarrow \text{unitary } (u P) \rrbracket \Longrightarrow \text{le-utrans } t \text{ (Sup-trans } S)$   
 $\langle \text{proof} \rangle$

**lemma** *Sup-trans-upper2*:

$\llbracket t \in S; \forall u \in S. \forall P. (\text{nneg } P \wedge \text{bounded-by } b P) \longrightarrow (\text{nneg } (u P) \wedge \text{bounded-by } b (u P));$   
 $\text{nneg } P; \text{bounded-by } b P \rrbracket \Longrightarrow t P \Vdash \text{Sup-trans } S P$   
 $\langle \text{proof} \rangle$

**lemma** *Sup-trans-least*:

$\llbracket \forall t \in S. \text{le-utrans } t u; \bigwedge P. \text{unitary } P \Longrightarrow \text{unitary } (u P) \rrbracket \Longrightarrow \text{le-utrans } (\text{Sup-trans } S) u$   
 $\langle \text{proof} \rangle$

**lemma** *Sup-trans-least2*:

$\llbracket \forall t \in S. \forall P. \text{nneg } P \longrightarrow \text{bounded-by } b P \longrightarrow t P \Vdash u P;$   
 $\forall u \in S. \forall P. (\text{nneg } P \wedge \text{bounded-by } b P) \longrightarrow (\text{nneg } (u P) \wedge \text{bounded-by } b (u P));$   
 $\text{nneg } P; \text{bounded-by } b P; \bigwedge P. \llbracket \text{nneg } P; \text{bounded-by } b P \rrbracket \Longrightarrow \text{nneg } (u P) \rrbracket \Longrightarrow$   
 $\text{Sup-trans } S P \Vdash u P$   
 $\langle \text{proof} \rangle$

**lemma** *feasible-Sup-trans*:

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

**definition**  $\text{lfp-trans} :: ('s \text{ trans} \Rightarrow 's \text{ trans}) \Rightarrow 's \text{ trans}$

**where**  $\text{lfp-trans } T = \text{Inf-trans } \{t. (\forall P. \text{sound } P \longrightarrow \text{sound } (t P)) \wedge \text{le-trans } (T t) t\}$

**lemma** *lfp-trans-lowerbound*:

$\llbracket \text{le-trans } (T t) t; \bigwedge P. \text{sound } P \Longrightarrow \text{sound } (t P) \rrbracket \Longrightarrow \text{le-trans } (\text{lfp-trans } T) t$   
 $\langle \text{proof} \rangle$

**lemma** *lfp-trans-greatest*:

$\llbracket \bigwedge t P. \llbracket \text{le-trans } (T t) t; \bigwedge P. \text{sound } P \Longrightarrow \text{sound } (t P) \rrbracket \Longrightarrow \text{le-trans } u t;$   
 $\bigwedge P. \text{sound } P \Longrightarrow \text{sound } (v P); \text{le-trans } (T v) v \rrbracket \Longrightarrow$   
 $\text{le-trans } u (\text{lfp-trans } T)$   
 $\langle \text{proof} \rangle$

**lemma** *lfp-trans-sound*:

**fixes**  $P Q :: 's \text{ expect}$   
**assumes**  $sP: \text{sound } P$   
**and**  $fv: \text{le-trans } (T v) v$   
**and**  $sv: \bigwedge P. \text{sound } P \Longrightarrow \text{sound } (v P)$   
**shows**  $\text{sound } (\text{lfp-trans } T P)$   
 $\langle \text{proof} \rangle$

**lemma** *lfp-trans-unitary*:

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

**lemma**  $\text{lfp-trans-lemma2}$ :  
**fixes**  $v :: 's \text{ trans}$   
**assumes**  $\text{mono} :: \bigwedge t u. \llbracket \text{le-trans } t u; \bigwedge P. \text{sound } P \implies \text{sound } (t P);$   
 $\bigwedge P. \text{sound } P \implies \text{sound } (u P) \rrbracket \implies \text{le-trans } (T t) (T u)$   
**and**  $nT :: \bigwedge t P. \llbracket \bigwedge Q. \text{sound } Q \implies \text{sound } (t Q); \text{sound } P \rrbracket \implies \text{sound } (T t P)$   
**and**  $fv :: \text{le-trans } (T v) v$   
**and**  $sv :: \bigwedge P. \text{sound } P \implies \text{sound } (v P)$   
**shows**  $\text{le-trans } (T (\text{lfp-trans } T)) (\text{lfp-trans } T)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{lfp-trans-lemma3}$ :  
**fixes**  $v :: 's \text{ trans}$   
**assumes**  $\text{mono} :: \bigwedge t u. \llbracket \text{le-trans } t u; \bigwedge P. \text{sound } P \implies \text{sound } (t P);$   
 $\bigwedge P. \text{sound } P \implies \text{sound } (u P) \rrbracket \implies \text{le-trans } (T t) (T u)$   
**and**  $sT :: \bigwedge t P. \llbracket \bigwedge Q. \text{sound } Q \implies \text{sound } (t Q); \text{sound } P \rrbracket \implies \text{sound } (T t P)$   
**and**  $fv :: \text{le-trans } (T v) v$   
**and**  $sv :: \bigwedge P. \text{sound } P \implies \text{sound } (v P)$   
**shows**  $\text{le-trans } (\text{lfp-trans } T) (T (\text{lfp-trans } T))$   
 $\langle \text{proof} \rangle$

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

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

**lemma**  $\text{gfp-trans-upperbound}$ :  
 $\llbracket \text{le-utrans } t (T t); \bigwedge P. \text{unitary } P \implies \text{unitary } (t P) \rrbracket \implies \text{le-utrans } t (\text{gfp-trans } T)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{gfp-trans-least}$ :  
 $\llbracket \bigwedge t. \llbracket \text{le-utrans } t (T t); \bigwedge P. \text{unitary } P \implies \text{unitary } (t P) \rrbracket \implies \text{le-utrans } t u;$   
 $\bigwedge P. \text{unitary } P \implies \text{unitary } (u P) \rrbracket \implies$   
 $\text{le-utrans } (\text{gfp-trans } T) u$



$\langle proof \rangle$

**lemma** *gfp-trans-unitary*:

**fixes**  $P::$ 's *expect*

**assumes**  $uP$ : unitary  $P$

**shows** unitary (gfp-trans  $T P$ )

$\langle proof \rangle$

**lemma** *gfp-trans-lemma2*:

**assumes** *mono*:  $\bigwedge t u. \llbracket le\text{-utrans } t u; \bigwedge P. \text{unitary } P \implies \text{unitary } (t P);$

$\bigwedge P. \text{unitary } P \implies \text{unitary } (u P) \rrbracket \implies le\text{-utrans } (T t) (T u)$

**and**  $hT$ :  $\bigwedge t P. \llbracket \bigwedge Q. \text{unitary } Q \implies \text{unitary } (t Q); \text{unitary } P \rrbracket \implies \text{unitary } (T t P)$

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

$\langle proof \rangle$

**lemma** *gfp-trans-lemma3*:

**assumes** *mono*:  $\bigwedge t u. \llbracket le\text{-utrans } t u; \bigwedge P. \text{unitary } P \implies \text{unitary } (t P);$

$\bigwedge P. \text{unitary } P \implies \text{unitary } (u P) \rrbracket \implies le\text{-utrans } (T t) (T u)$

**and**  $hT$ :  $\bigwedge t P. \llbracket \bigwedge Q. \text{unitary } Q \implies \text{unitary } (t Q); \text{unitary } P \rrbracket \implies \text{unitary } (T t P)$

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

$\langle proof \rangle$

**lemma** *gfp-trans-unfold*:

**assumes** *mono*:  $\bigwedge t u. \llbracket le\text{-utrans } t u; \bigwedge P. \text{unitary } P \implies \text{unitary } (t P);$

$\bigwedge P. \text{unitary } P \implies \text{unitary } (u P) \rrbracket \implies le\text{-utrans } (T t) (T u)$

**and**  $hT$ :  $\bigwedge t P. \llbracket \bigwedge Q. \text{unitary } Q \implies \text{unitary } (t Q); \text{unitary } P \rrbracket \implies \text{unitary } (T t P)$

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

$\langle proof \rangle$

### 3.3.3 Tail Recursion

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

**lemma** *gfp-pulldown*:

**fixes**  $P::$ 's *expect*

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

**and**  $ft$ :  $\bigwedge t P. \llbracket \bigwedge Q. \text{unitary } Q \implies \text{unitary } (t Q); \text{unitary } P \rrbracket \implies \text{unitary } (T t P)$

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

**and**  $mt$ :  $\bigwedge P Q R. \llbracket \text{unitary } P; \text{unitary } Q; \text{unitary } R; Q \vdash R \rrbracket \implies t P Q \vdash t P R$

**and**  $uP$ : unitary  $P$

**and**  $monoT$ :  $\bigwedge t u. \llbracket le\text{-utrans } t u; \bigwedge P. \text{unitary } P \implies \text{unitary } (t P);$

$\bigwedge P. \text{unitary } P \implies \text{unitary } (u P) \rrbracket \implies le\text{-utrans } (T t) (T u)$

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

$\langle proof \rangle$

**lemma** *lfp-pulldown*:

**fixes**  $P::$ 's *expect* **and**  $t::$ 's *expect*  $\implies$  's *trans*

**and**  $T :: 's \text{ trans} \Rightarrow 's \text{ trans}$   
**assumes** *tailcall*:  $\bigwedge u P. \text{sound } P \Longrightarrow T u P = t P (u P)$   
**and** *st*:  $\bigwedge P Q. \text{sound } P \Longrightarrow \text{sound } Q \Longrightarrow \text{sound } (t P Q)$   
**and** *mt*:  $\bigwedge P. \text{sound } P \Longrightarrow \text{mono-trans } (t P)$   
**and** *monoT*:  $\bigwedge t u. \llbracket \text{le-trans } t u; \bigwedge P. \text{sound } P \Longrightarrow \text{sound } (t P);$   
 $\bigwedge P. \text{sound } P \Longrightarrow \text{sound } (u P) \rrbracket \Longrightarrow \text{le-trans } (T t) (T u)$   
**and** *nT*:  $\bigwedge t P. \llbracket \bigwedge Q. \text{sound } Q \Longrightarrow \text{sound } (t Q); \text{sound } P \rrbracket \Longrightarrow \text{sound } (T t P)$   
**and** *fv*:  $\text{le-trans } (T v) v$   
**and** *sv*:  $\bigwedge P. \text{sound } P \Longrightarrow \text{sound } (v P)$   
**and** *sP*:  $\text{sound } P$   
**shows**  $\text{lfp-trans } T P = \text{lfp-exp } (t P) \text{ (is } ?X P = ?Y P)$   
 $\langle \text{proof} \rangle$

**definition** *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** *Inf-utrans-lower*:  
 $\llbracket t \in S; \forall t \in S. \forall P. \text{unitary } P \longrightarrow \text{unitary } (t P) \rrbracket \Longrightarrow \text{le-utrans } (\text{Inf-utrans } S) t$   
 $\langle \text{proof} \rangle$

**lemma** *Inf-utrans-greatest*:  
 $\llbracket \bigwedge P. \text{unitary } P \Longrightarrow \text{unitary } (t P); \forall u \in S. \text{le-utrans } t u \rrbracket \Longrightarrow \text{le-utrans } t (\text{Inf-utrans } S)$   
 $\langle \text{proof} \rangle$

**end**

## Chapter 4

# The pGCL Language

### 4.1 A Shallow Embedding of pGCL in HOL

**theory** *Embedding* imports *Misc Induction* begin

#### 4.1.1 Core Primitives and Syntax

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

**type-synonym**  $'s \text{ prog} = \text{bool} \Rightarrow ('s \Rightarrow \text{real}) \Rightarrow ('s \Rightarrow \text{real})$

*Abort* either always fails,  $\lambda P s. 0::'c$ , or always succeeds,  $\lambda P s. 1::'c$ .

**definition**  $\text{Abort} :: 's \text{ prog}$   
**where**  $\text{Abort} \equiv \lambda ab P s. \text{if } ab \text{ then } 0 \text{ else } 1$

*Skip* does nothing at all.

**definition**  $\text{Skip} :: 's \text{ prog}$   
**where**  $\text{Skip} \equiv \lambda ab P. P$

*Apply* lifts a state transformer into the space of programs.

**definition**  $\text{Apply} :: ('s \Rightarrow 's) \Rightarrow 's \text{ prog}$   
**where**  $\text{Apply } f \equiv \lambda ab P s. P (f s)$

*Seq* is sequential composition.

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

*PC* is *probabilistic* choice between programs.

**definition**  $\text{PC} :: 's \text{ prog} \Rightarrow ('s \Rightarrow \text{real}) \Rightarrow 's \text{ prog} \Rightarrow 's \text{ prog}$   
( $- \cdot \oplus -$  [58,57,57] 57)

**where**  $PC\ a\ P\ b \equiv \lambda ab\ Q\ s. P\ s * a\ ab\ Q\ s + (1 - P\ s) * b\ ab\ Q\ s$

$DC$  is *demonic* choice between programs.

**definition**  $DC :: 's\ prog \Rightarrow 's\ prog \Rightarrow 's\ prog\ (-\ \sqcap\ -\ [58,57]\ 57)$

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

**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 then least or greatest fixed point.

**definition**  $Mu :: ('s\ prog \Rightarrow 's\ prog) \Rightarrow 's\ prog\ (\mathbf{binder}\ \mu\ 50)$

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

$repeat$  expresses finite repetition

**primrec**

$repeat :: nat \Rightarrow 'a\ prog \Rightarrow 'a\ prog$

**where**

$repeat\ 0\ p = Skip\ |$

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

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

**definition**  $SetDC :: ('a \Rightarrow 's\ prog) \Rightarrow ('s \Rightarrow 'a\ set) \Rightarrow 's\ prog$

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

**syntax**  $-SetDC :: p\ ttrn \Rightarrow ('s \Rightarrow 'a\ set) \Rightarrow 's\ prog \Rightarrow 's\ prog$   
 $(\sqcap\ -\ \in\ -\ / - 100)$

**translations**  $\sqcap_{x \in S}. p == CONST\ SetDC\ (\%x. p)\ S$

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

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

**definition**

$SetPC :: ('a \Rightarrow 's\ prog) \Rightarrow ('s \Rightarrow 'a \Rightarrow real) \Rightarrow 's\ prog$

**where**

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

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

**definition**

$Bind :: ('s \Rightarrow 'a) \Rightarrow ('a \Rightarrow 's\ prog) \Rightarrow 's\ prog$   
**where**  
 $Bind\ g\ f\ ab \equiv \lambda P\ s.\ let\ a = g\ s\ in\ f\ a\ ab\ P\ s$

This gives us something like let syntax

**syntax**  $-Bind :: pptrn \Rightarrow ('s \Rightarrow 'a) \Rightarrow 's\ prog \Rightarrow 's\ prog$   
 $(-is - in - [55,55,55]55)$   
**translations**  $x\ is\ f\ in\ a \Rightarrow CONST\ Bind\ f\ (\%x.\ a)$

**definition**  $flip :: ('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'a \Rightarrow 'c$   
**where**  $[simp]: flip\ f = (\lambda b\ a.\ f\ a\ b)$

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

**syntax**  $-PBind :: pptrn \Rightarrow ('s \Rightarrow real) \Rightarrow 's\ prog \Rightarrow 's\ prog$   
 $(bind - at - in - [55,55,55]55)$   
**translations**  $bind\ x\ at\ p\ in\ a \Rightarrow CONST\ SetPC\ (\%x.\ a)\ (CONST\ flip\ (\%x.\ p))$

**syntax**  $-DBind :: pptrn \Rightarrow ('s \Rightarrow 'a\ set) \Rightarrow 's\ prog \Rightarrow 's\ prog$   
 $(bind - from - in - [55,55,55]55)$   
**translations**  $bind\ x\ from\ S\ in\ a \Rightarrow CONST\ SetDC\ (\%x.\ a)\ S$

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

**syntax**  
 $-assign :: ident \Rightarrow 'a \Rightarrow 's\ prog\ (- := - [1000,900]900)$   
 $\langle ML \rangle$

**syntax**  
 $-SetPC :: ident \Rightarrow ('s \Rightarrow 'a \Rightarrow real) \Rightarrow 's\ prog$   
 $(choose - at - [66,66]66)$   
 $\langle ML \rangle$

**syntax**  
 $-set-dc :: ident \Rightarrow ('s \Rightarrow 'a\ set) \Rightarrow 's\ prog\ (- \in - [66,66]66)$   
 $\langle ML \rangle$

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

**syntax**  
 $-set-dc-UNIV :: ident \Rightarrow 's\ prog\ (any - [66]66)$

**translations**  
 $-set-dc-UNIV\ x \Rightarrow -set-dc\ x\ (\%-. CONST\ UNIV)$

**definition**  
 $wp :: 's\ prog \Rightarrow 's\ trans$   
**where**

$wp\ pr \equiv pr\ True$

**definition**

$wlp :: 's\ prog \Rightarrow 's\ trans$

**where**

$wlp\ pr \equiv pr\ False$

If-Then-Else as a degenerate probabilistic choice.

**abbreviation**(*input*)

$if\text{-}then\text{-}else :: ['s \Rightarrow bool, 's\ prog, 's\ prog] \Rightarrow 's\ prog$   
(*If - Then - Else - 58*)

**where**

$If\ P\ Then\ a\ Else\ b == a \llbracket P \rrbracket \oplus b$

Syntax for loops

**abbreviation**

$do\text{-}while :: ['s \Rightarrow bool, 's\ prog] \Rightarrow 's\ prog$   
(*do -  $\longrightarrow$  // (4 -) // od*)

**where**

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

### 4.1.2 Unfolding rules for non-recursive primitives

**lemma** *eval-wp-Abort:*

$wp\ Abort\ P = (\lambda s. 0)$   
*<proof>*

**lemma** *eval-wlp-Abort:*

$wlp\ Abort\ P = (\lambda s. 1)$   
*<proof>*

**lemma** *eval-wp-Skip:*

$wp\ Skip\ P = P$   
*<proof>*

**lemma** *eval-wlp-Skip:*

$wlp\ Skip\ P = P$   
*<proof>*

**lemma** *eval-wp-Apply:*

$wp\ (Apply\ f)\ P = P\ o\ f$   
*<proof>*

**lemma** *eval-wlp-Apply:*

$wlp\ (Apply\ f)\ P = P\ o\ f$   
*<proof>*

**lemma** *eval-wp-Seq:*

$wp\ (a\ ;;\ b)\ P = (wp\ a\ o\ wp\ b)\ P$

$\langle proof \rangle$

**lemma** *eval-wlp-Seq*:

$$wlp (a ;; b) P = (wlp a \circ wlp b) P$$

$\langle proof \rangle$

**lemma** *eval-wp-PC*:

$$wp (a \oplus b) P = (\lambda s. Q s * wp a P s + (I - Q s) * wp b P s)$$

$\langle proof \rangle$

**lemma** *eval-wlp-PC*:

$$wlp (a \oplus b) P = (\lambda s. Q s * wlp a P s + (I - Q s) * wlp b P s)$$

$\langle proof \rangle$

**lemma** *eval-wp-DC*:

$$wp (a \sqcap b) P = (\lambda s. \min (wp a P s) (wp b P s))$$

$\langle proof \rangle$

**lemma** *eval-wlp-DC*:

$$wlp (a \sqcap b) P = (\lambda s. \min (wlp a P s) (wlp b P s))$$

$\langle proof \rangle$

**lemma** *eval-wp-AC*:

$$wp (a \sqcup b) P = (\lambda s. \max (wp a P s) (wp b P s))$$

$\langle proof \rangle$

**lemma** *eval-wlp-AC*:

$$wlp (a \sqcup b) P = (\lambda s. \max (wlp a P s) (wlp b P s))$$

$\langle proof \rangle$

**lemma** *eval-wp-Embed*:

$$wp (Embed t) = t$$

$\langle proof \rangle$

**lemma** *eval-wlp-Embed*:

$$wlp (Embed t) = t$$

$\langle proof \rangle$

**lemma** *eval-wp-SetDC*:

$$wp (SetDC p S) R s = Inf ((\lambda a. wp (p a) R s) ` S s)$$

$\langle proof \rangle$

**lemma** *eval-wlp-SetDC*:

$$wlp (SetDC p S) R s = Inf ((\lambda a. wlp (p a) R s) ` S s)$$

$\langle proof \rangle$

**lemma** *eval-wp-SetPC*:

$$wp (SetPC f p) P = (\lambda s. \sum_{a \in \text{supp } (p s)}. p s a * wp (f a) P s)$$

$\langle proof \rangle$

**lemma** *eval-wlp-SetPC*:

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

*<proof>*

**lemma** *eval-wp-Mu*:

$$wp (\mu t. T t) = lfp\text{-trans } (\lambda t. wp (T (Embed t)))$$

*<proof>*

**lemma** *eval-wlp-Mu*:

$$wlp (\mu t. T t) = gfp\text{-trans } (\lambda t. wlp (T (Embed t)))$$

*<proof>*

**lemma** *eval-wp-Bind*:

$$wp (Bind g f) = (\lambda P s. wp (f (g s)) P s)$$

*<proof>*

**lemma** *eval-wlp-Bind*:

$$wlp (Bind g f) = (\lambda P s. wlp (f (g s)) P s)$$

*<proof>*

Use `simp add:wp_eval` to fully unfold a program fragment

**lemmas** *wp-eval = eval-wp-Abort eval-wlp-Abort eval-wp-Skip eval-wlp-Skip*  
*eval-wp-Apply eval-wlp-Apply eval-wp-Seq eval-wlp-Seq*  
*eval-wp-PC eval-wlp-PC eval-wp-DC eval-wlp-DC*  
*eval-wp-AC eval-wlp-AC*  
*eval-wp-Embed eval-wlp-Embed eval-wp-SetDC eval-wlp-SetDC*  
*eval-wp-SetPC eval-wlp-SetPC eval-wp-Mu eval-wlp-Mu*  
*eval-wp-Bind eval-wlp-Bind*

**lemma** *Skip-Seq*:

$$Skip ;; A = A$$

*<proof>*

**lemma** *Seq-Skip*:

$$A ;; Skip = A$$

*<proof>*

Use these as `simp` rules to clear out Skips

**lemmas** *skip-simps = Skip-Seq Seq-Skip*

**end**

## 4.2 Healthiness

**theory** *Healthiness imports Embedding begin*



### 4.2.1 The Healthiness of the Embedding

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

**lemma** *healthy-wp-Abort*:

*healthy* (*wp Abort*)

*<proof>*

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

*nearly-healthy* (*wlp Abort*)

*<proof>*

**lemma** *healthy-wp-Skip*:

*healthy* (*wp Skip*)

*<proof>*

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

*nearly-healthy* (*wlp Skip*)

*<proof>*

**lemma** *healthy-wp-Seq*:

**fixes** *t::'s prog* **and** *u*

**assumes** *ht: healthy* (*wp t*) **and** *hu: healthy* (*wp u*)

**shows** *healthy* (*wp (t ;; u)*)

*<proof>*

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

**fixes** *t::'s prog* **and** *u*

**assumes** *ht: nearly-healthy* (*wlp t*) **and** *hu: nearly-healthy* (*wlp u*)

**shows** *nearly-healthy* (*wlp (t ;; u)*)

*<proof>*

**lemma** *healthy-wp-PC*:

**fixes** *f::'s prog*

**assumes** *hf: healthy* (*wp f*) **and** *hg: healthy* (*wp g*)

**and** *uP: unitary P*

**shows** *healthy* (*wp (f P⊕ g)*)

*<proof>*

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

**fixes** *f::'s prog*

**assumes** *hf: nearly-healthy* (*wlp f*)

**and** *hg: nearly-healthy* (*wlp g*)

**and** *uP: unitary P*

**shows** *nearly-healthy* (*wlp (f P⊕ g)*)

*<proof>*

**lemma** *healthy-wp-DC*:

**fixes** *f::'s prog*

**assumes**  $hf: \text{healthy } (wp\ f)$  **and**  $hg: \text{healthy } (wp\ g)$   
**shows**  $\text{healthy } (wp\ (f \sqcap g))$   
 $\langle \text{proof} \rangle$

**lemma** *nearly-healthy-wlp-DC*:  
**fixes**  $f:: 's\ prog$   
**assumes**  $hf: \text{nearly-healthy } (wlp\ f)$   
**and**  $hg: \text{nearly-healthy } (wlp\ g)$   
**shows**  $\text{nearly-healthy } (wlp\ (f \sqcap g))$   
 $\langle \text{proof} \rangle$

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

**lemma** *nearly-healthy-wlp-AC*:  
**fixes**  $f:: 's\ prog$   
**assumes**  $hf: \text{nearly-healthy } (wlp\ f)$   
**and**  $hg: \text{nearly-healthy } (wlp\ g)$   
**shows**  $\text{nearly-healthy } (wlp\ (f \sqcup g))$   
 $\langle \text{proof} \rangle$

**lemma** *healthy-wp-Embed*:  
 $\text{healthy } t \implies \text{healthy } (wp\ (\text{Embed } t))$   
 $\langle \text{proof} \rangle$

**lemma** *nearly-healthy-wlp-Embed*:  
 $\text{nearly-healthy } t \implies \text{nearly-healthy } (wlp\ (\text{Embed } t))$   
 $\langle \text{proof} \rangle$

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

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

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

**lemma** *nearly-healthy-wlp-SetDC*:

**fixes**  $prog::'b \Rightarrow 'a\ prog$  **and**  $S::'a \Rightarrow 'b\ set$

**assumes** *healthy*:  $\bigwedge x\ s.\ x \in S\ s \implies\ nearly\text{-}healthy\ (wlp\ (prog\ x))$

**and** *nonempty*:  $\bigwedge s.\ \exists x.\ x \in S\ s$

**shows**  $nearly\text{-}healthy\ (wlp\ (SetDC\ prog\ S))$  (**is**  $nearly\text{-}healthy\ ?T$ )  
 $\langle proof \rangle$

**lemma** *healthy-wp-SetPC*:

**fixes**  $p::'s \Rightarrow 'a \Rightarrow real$

**and**  $f::'a \Rightarrow 's\ prog$

**assumes** *healthy*:  $\bigwedge a\ s.\ a \in\ supp\ (p\ s) \implies\ healthy\ (wp\ (f\ a))$

**and** *sound*:  $\bigwedge s.\ sound\ (p\ s)$

**and** *sub-dist*:  $\bigwedge s.\ (\sum a \in\ supp\ (p\ s).\ p\ s\ a) \leq 1$

**shows**  $healthy\ (wp\ (SetPC\ f\ p))$  (**is**  $healthy\ ?X$ )  
 $\langle proof \rangle$

**lemma** *nearly-healthy-wlp-SetPC*:

**fixes**  $p::'s \Rightarrow 'a \Rightarrow real$

**and**  $f::'a \Rightarrow 's\ prog$

**assumes** *healthy*:  $\bigwedge a\ s.\ a \in\ supp\ (p\ s) \implies\ nearly\text{-}healthy\ (wlp\ (f\ a))$

**and** *sound*:  $\bigwedge s.\ sound\ (p\ s)$

**and** *sub-dist*:  $\bigwedge s.\ (\sum a \in\ supp\ (p\ s).\ p\ s\ a) \leq 1$

**shows**  $nearly\text{-}healthy\ (wlp\ (SetPC\ f\ p))$  (**is**  $nearly\text{-}healthy\ ?X$ )  
 $\langle proof \rangle$

**lemma** *healthy-wp-Apply*:

$healthy\ (wp\ (Apply\ f))$

$\langle proof \rangle$

**lemma** *nearly-healthy-wlp-Apply*:

$nearly\text{-}healthy\ (wlp\ (Apply\ f))$

$\langle proof \rangle$

**lemma** *healthy-wp-Bind*:

**fixes**  $f::'s \Rightarrow 'a$

**assumes** *hsub*:  $\bigwedge s.\ healthy\ (wp\ (p\ (f\ s)))$

**shows**  $healthy\ (wp\ (Bind\ f\ p))$

$\langle proof \rangle$

**lemma** *nearly-healthy-wlp-Bind*:

**fixes**  $f::'s \Rightarrow 'a$

**assumes** *hsub*:  $\bigwedge s.\ nearly\text{-}healthy\ (wlp\ (p\ (f\ s)))$

**shows**  $nearly\text{-}healthy\ (wlp\ (Bind\ f\ p))$

$\langle proof \rangle$

### 4.2.2 Healthiness for Loops

**lemma** *wp-loop-step-mono*:

**fixes**  $t\ u::'s\ trans$

**assumes**  $hb$ : healthy (wp body)  
**and**  $le$ : le-trans  $t u$   
**and**  $ht$ :  $\bigwedge P. \text{sound } P \implies \text{sound } (t P)$   
**and**  $hu$ :  $\bigwedge P. \text{sound } P \implies \text{sound } (u P)$   
**shows** le-trans (wp (body ;; Embed  $t \ll G \gg \oplus \text{Skip}$ ))  
                   (wp (body ;; Embed  $u \ll G \gg \oplus \text{Skip}$ ))  
 $\langle \text{proof} \rangle$

**lemma** wlp-loop-step-mono:  
**fixes**  $t u$ ::'s trans  
**assumes**  $mb$ : nearly-healthy (wlp body)  
**and**  $le$ : le-utrans  $t u$   
**and**  $ht$ :  $\bigwedge P. \text{unitary } P \implies \text{unitary } (t P)$   
**and**  $hu$ :  $\bigwedge P. \text{unitary } P \implies \text{unitary } (u P)$   
**shows** le-utrans (wlp (body ;; Embed  $t \ll G \gg \oplus \text{Skip}$ ))  
                   (wlp (body ;; Embed  $u \ll G \gg \oplus \text{Skip}$ ))  
 $\langle \text{proof} \rangle$

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

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

**lemma** lfp-loop-greatest:  
**fixes**  $P$ ::'s expect  
**assumes**  $lb$ :  $\bigwedge R. \lambda s. \ll G \gg s * \text{wp body } R s + \ll \mathcal{N} G \gg s * P s \Vdash R \implies \text{sound } R \implies Q \Vdash R$   
**and**  $hb$ : healthy (wp body)  
**and**  $sP$ : sound  $P$   
**and**  $sQ$ : sound  $Q$   
**shows**  $Q \Vdash \text{lfp-exp } (\lambda Q s. \ll G \gg s * \text{wp body } Q s + \ll \mathcal{N} G \gg s * P s)$   
 $\langle \text{proof} \rangle$

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

**lemma** wlp-loop-step-unitary:  
**fixes**  $t u$ ::'s trans  
**assumes**  $hb$ : nearly-healthy (wlp body)  
**and**  $ht$ :  $\bigwedge P. \text{unitary } P \implies \text{unitary } (t P)$   
**and**  $uP$ : unitary  $P$   
**shows** unitary (wlp (body ;; Embed  $t \ll G \gg \oplus \text{Skip}$ )  $P$ )  
 $\langle \text{proof} \rangle$

**lemma** *wp-loop-step-sound*:  
**fixes**  $t u :: 's \text{ trans}$   
**assumes**  $hb: \text{healthy } (wp \text{ body})$   
**and**  $ht: \bigwedge P. \text{sound } P \implies \text{sound } (t P)$   
**and**  $sP: \text{sound } P$   
**shows**  $\text{sound } (wp \text{ (body ;; Embed } t \ll G \gg \oplus \text{Skip) } P)$   
 $\langle \text{proof} \rangle$

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

**lemma** *wlp-Loop1*:  
**fixes**  $\text{body} :: 's \text{ prog}$   
**assumes**  $\text{unitary}: \text{unitary } P$   
**and**  $\text{healthy}: \text{nearly-healthy } (wlp \text{ body})$   
**shows**  $wlp \text{ (do } G \longrightarrow \text{body } od) P =$   
 $\text{gfp-exp } (\lambda Q s. \ll G \gg s * wlp \text{ body } Q s + \ll \mathcal{N} G \gg s * P s)$   
**(is**  $?X = \text{gfp-exp } (?Y P)$ **)**  
 $\langle \text{proof} \rangle$

**lemma** *wp-loop-sound*:  
**assumes**  $sP: \text{sound } P$   
**and**  $hb: \text{healthy } (wp \text{ body})$   
**shows**  $\text{sound } (wp \text{ do } G \longrightarrow \text{body } od P)$   
 $\langle \text{proof} \rangle$

Likewise, we can rewrite strict loops.

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

**lemma** *nearly-healthy-wlp-loop*:  
**fixes**  $\text{body} :: 's \text{ prog}$   
**assumes**  $hb: \text{nearly-healthy } (wlp \text{ body})$   
**shows**  $\text{nearly-healthy } (wlp \text{ (do } G \longrightarrow \text{body } od))$   
 $\langle \text{proof} \rangle$

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

**lemma** *healthy-wp-loop*:  
**fixes**  $\text{body} :: 's \text{ prog}$   
**assumes**  $hb: \text{healthy } (wp \text{ body})$   
**shows**  $\text{healthy } (wp \text{ (do } G \longrightarrow \text{body } od))$

$\langle proof \rangle$

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* =  $(\forall M. (\forall i. (M \Vdash M (Suc \ i)) \wedge \text{sound } (M \ i)) \longrightarrow$   
 $(\exists b. \forall i. \text{bounded-by } b \ (M \ i)) \longrightarrow$   
 $t \ (Sup\text{-exp } (range \ M)) = Sup\text{-exp } (range \ (t \ o \ M)))$

**lemma** *bd-ctsD*:

$\llbracket \text{bd-cts } t; \bigwedge i. M \ i \Vdash M \ (Suc \ i); \bigwedge i. \text{sound } (M \ i); \bigwedge i. \text{bounded-by } b \ (M \ i) \rrbracket \Longrightarrow$   
 $t \ (Sup\text{-exp } (range \ M)) = Sup\text{-exp } (range \ (t \ o \ M))$   
 $\langle proof \rangle$

**lemma** *bd-ctsI*:

$(\bigwedge b \ M. (\bigwedge i. M \ i \Vdash M \ (Suc \ i)) \Longrightarrow (\bigwedge i. \text{sound } (M \ i)) \Longrightarrow (\bigwedge i. \text{bounded-by } b \ (M \ i)) \Longrightarrow$   
 $t \ (Sup\text{-exp } (range \ M)) = Sup\text{-exp } (range \ (t \ o \ M))) \Longrightarrow \text{bd-cts } t$   
 $\langle proof \rangle$

A generalised property for transformers of transformers.

**definition** *bd-cts-tr* :: '*s trans*  $\Rightarrow$  '*s trans*  $\Rightarrow$  *bool*

**where** *bd-cts-tr* *T* =  $(\forall M. (\forall i. \text{le-trans } (M \ i) \ (M \ (Suc \ i)) \wedge \text{feasible } (M \ i)) \longrightarrow$   
 $\text{equiv-trans } (T \ (Sup\text{-trans } (M \ ' \ UNIV))) \ (Sup\text{-trans } ((T \ o \ M) \ ' \ UNIV)))$

**lemma** *bd-cts-trD*:

$\llbracket \text{bd-cts-tr } T; \bigwedge i. \text{le-trans } (M \ i) \ (M \ (Suc \ i)); \bigwedge i. \text{feasible } (M \ i) \rrbracket \Longrightarrow$   
 $\text{equiv-trans } (T \ (Sup\text{-trans } (M \ ' \ UNIV))) \ (Sup\text{-trans } ((T \ o \ M) \ ' \ UNIV))$   
 $\langle proof \rangle$

**lemma** *bd-cts-trI*:

$(\bigwedge M. (\bigwedge i. \text{le-trans } (M \ i) \ (M \ (\text{Suc } i))) \implies (\bigwedge i. \text{feasible } (M \ i)) \implies$   
 $\text{equiv-trans } (T \ (\text{Sup-trans } (M \ ' \ \text{UNIV}))) \ (\text{Sup-trans } ((T \ o \ M) \ ' \ \text{UNIV}))) \implies \text{bd-cts-tr}$   
 $T$   
 $\langle \text{proof} \rangle$

### 4.3.1 Continuity of Primitives

**lemma** *cts-wp-Abort*:

$\text{bd-cts } (\text{wp } (\text{Abort}::'s \ \text{prog}))$   
 $\langle \text{proof} \rangle$

**lemma** *cts-wp-Skip*:

$\text{bd-cts } (\text{wp } \text{Skip})$   
 $\langle \text{proof} \rangle$

**lemma** *cts-wp-Apply*:

$\text{bd-cts } (\text{wp } (\text{Apply } f))$   
 $\langle \text{proof} \rangle$

**lemma** *cts-wp-Bind*:

**fixes**  $a::'a \Rightarrow 's \ \text{prog}$   
**assumes**  $ca: \bigwedge s. \text{bd-cts } (\text{wp } (a \ (f \ s)))$   
**shows**  $\text{bd-cts } (\text{wp } (\text{Bind } f \ a))$   
 $\langle \text{proof} \rangle$

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

**lemma** *cts-wp-DC*:

**fixes**  $a \ b::'s \ \text{prog}$   
**assumes**  $ca: \text{bd-cts } (\text{wp } a)$   
**and**  $cb: \text{bd-cts } (\text{wp } b)$   
**and**  $ha: \text{healthy } (\text{wp } a)$   
**and**  $hb: \text{healthy } (\text{wp } b)$   
**shows**  $\text{bd-cts } (\text{wp } (a \ \sqcap \ b))$   
 $\langle \text{proof} \rangle$

**lemma** *cts-wp-Seq*:

**fixes**  $a \ b::'s \ \text{prog}$   
**assumes**  $ca: \text{bd-cts } (\text{wp } a)$   
**and**  $cb: \text{bd-cts } (\text{wp } b)$   
**and**  $hb: \text{healthy } (\text{wp } b)$   
**shows**  $\text{bd-cts } (\text{wp } (a \ ; \ b))$   
 $\langle \text{proof} \rangle$

**lemma** *cts-wp-PC*:

**fixes**  $a \ b::'s \ \text{prog}$

**assumes**  $ca: bd\text{-cts } (wp\ a)$   
**and**  $cb: bd\text{-cts } (wp\ b)$   
**and**  $ha: healthy\ (wp\ a)$   
**and**  $hb: healthy\ (wp\ b)$   
**and**  $up: unitary\ p$   
**shows**  $bd\text{-cts } (wp\ (PC\ a\ p\ b))$   
 $\langle proof \rangle$

Both set-based choice operators are only continuous for finite sets (probabilistic choice *can* be extended infinitely, but we have not done so). The proofs for both are inductive, and rely on the above results on binary operators.

**lemma** *SetPC-Bind*:  
 $SetPC\ a\ p = Bind\ p\ (\lambda p. SetPC\ a\ (\lambda\cdot. p))$   
 $\langle proof \rangle$

**lemma** *SetPC-remove*:  
**assumes**  $nz: p\ x \neq 0$  **and**  $nI: p\ x \neq 1$   
**and**  $fsupp: finite\ (supp\ p)$   
**shows**  $SetPC\ a\ (\lambda\cdot. p) = PC\ (a\ x)\ (\lambda\cdot. p\ x)\ (SetPC\ a\ (\lambda\cdot. dist\text{-remove}\ p\ x))$   
 $\langle proof \rangle$

**lemma** *cts-bot*:  
 $bd\text{-cts } (\lambda(P::'s\ expect)\ (s::'s). 0::real)$   
 $\langle proof \rangle$

**lemma** *wp-SetPC-nil*:  
 $wp\ (SetPC\ a\ (\lambda s\ a.\ 0)) = (\lambda P\ s.\ 0)$   
 $\langle proof \rangle$

**lemma** *SetPC-sgl*:  
 $supp\ p = \{x\} \implies SetPC\ a\ (\lambda\cdot. p) = (\lambda ab\ P\ s.\ p\ x * a\ x\ ab\ P\ s)$   
 $\langle proof \rangle$

**lemma** *bd-cts-scale*:  
**fixes**  $a::'s\ trans$   
**assumes**  $ca: bd\text{-cts } a$   
**and**  $ha: healthy\ a$   
**and**  $nnc: 0 \leq c$   
**shows**  $bd\text{-cts } (\lambda P\ s.\ c * a\ P\ s)$   
 $\langle proof \rangle$

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



$\langle proof \rangle$

**lemma** *cts-wp-SetPC*:

**fixes**  $a:: 'a \Rightarrow 's\ prog$

**assumes**  $ca: \bigwedge x\ s. x \in (supp\ (p\ s)) \Longrightarrow bd\text{-cts}\ (wp\ (a\ x))$

**and**  $ha: \bigwedge x\ s. x \in (supp\ (p\ s)) \Longrightarrow healthy\ (wp\ (a\ x))$

**and**  $up: \bigwedge s. unitary\ (p\ s)$

**and**  $sump: \bigwedge s. sum\ (p\ s)\ (supp\ (p\ s)) \leq 1$

**and**  $fsupp: \bigwedge s. finite\ (supp\ (p\ s))$

**shows**  $bd\text{-cts}\ (wp\ (SetPC\ a\ p))$

$\langle proof \rangle$

**lemma** *wp-SetDC-Bind*:

$SetDC\ a\ S = Bind\ S\ (\lambda S. SetDC\ a\ (\lambda-. S))$

$\langle proof \rangle$

**lemma** *SetDC-finite-insert*:

**assumes**  $fS: finite\ S$

**and**  $neS: S \neq \{\}$

**shows**  $SetDC\ a\ (\lambda-. insert\ x\ S) = a\ x \sqcap SetDC\ a\ (\lambda-. S)$

$\langle proof \rangle$

**lemma** *SetDC-singleton*:

$SetDC\ a\ (\lambda-. \{x\}) = a\ x$

$\langle proof \rangle$

**lemma** *cts-wp-SetDC-const*:

**fixes**  $a:: 'a \Rightarrow 's\ prog$

**assumes**  $ca: \bigwedge x. x \in S \Longrightarrow bd\text{-cts}\ (wp\ (a\ x))$

**and**  $ha: \bigwedge x. x \in S \Longrightarrow healthy\ (wp\ (a\ x))$

**and**  $fS: finite\ S$

**and**  $neS: S \neq \{\}$

**shows**  $bd\text{-cts}\ (wp\ (SetDC\ a\ (\lambda-. S)))$

$\langle proof \rangle$

**lemma** *cts-wp-SetDC*:

**fixes**  $a:: 'a \Rightarrow 's\ prog$

**assumes**  $ca: \bigwedge x\ s. x \in S\ s \Longrightarrow bd\text{-cts}\ (wp\ (a\ x))$

**and**  $ha: \bigwedge x\ s. x \in S\ s \Longrightarrow healthy\ (wp\ (a\ x))$

**and**  $fS: \bigwedge s. finite\ (S\ s)$

**and**  $neS: \bigwedge s. S\ s \neq \{\}$

**shows**  $bd\text{-cts}\ (wp\ (SetDC\ a\ S))$

$\langle proof \rangle$

**lemma** *cts-wp-repeat*:

$bd\text{-cts}\ (wp\ a) \Longrightarrow healthy\ (wp\ a) \Longrightarrow bd\text{-cts}\ (wp\ (repeat\ n\ a))$

$\langle proof \rangle$

**lemma** *cts-wp-Embed*:

$bd\text{-}cts\ t \implies bd\text{-}cts\ (wp\ (Embed\ t))$   
 $\langle proof \rangle$

### 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**  $cb: bd\text{-}cts\ (wp\ body)$

**shows**  $bd\text{-}cts\text{-}tr\ (\lambda x. wp\ (body\ ;;\ Embed\ x\ \ll G \gg \oplus\ Skip))\ (is\ bd\text{-}cts\text{-}tr\ ?F)$   
 $\langle proof \rangle$

**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**  $sP: sound\ P$

**and**  $hb: healthy\ (wp\ body)$

**shows**  $mono\text{-}trans\ (\lambda Q\ s. \ll G \gg\ s * wp\ body\ Q\ s + \ll \mathcal{N}\ G \gg\ s * P\ s)$   
 $\langle proof \rangle$

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**  $sP: sound\ P$

**shows**  $lfp\text{-}exp\ (\lambda Q\ s. \ll G \gg\ s * wp\ body\ Q\ s + \ll \mathcal{N}\ G \gg\ s * P\ s) =$

$(\lambda s. \ll G \gg\ s * wp\ body\ (lfp\text{-}exp\ (\lambda Q\ s. \ll G \gg\ s * wp\ body\ Q\ s + \ll \mathcal{N}\ G \gg\ s * P\ s))\ s +$   
 $\ll \mathcal{N}\ G \gg\ s * P\ s)$

$\langle proof \rangle$

**lemma** *wp-loop-step-unitary*:

**fixes**  $body::'s\ prog$

**assumes**  $hb: healthy\ (wp\ body)$

**and**  $uP: unitary\ P$  **and**  $uQ: unitary\ Q$

**shows** *unitary*  $(\lambda s. \ll G \gg s * wp \text{ body } Q s + \ll \mathcal{N} G \gg s * P s)$   
 $\langle proof \rangle$

**lemma** *lfp-loop-unitary*:

**fixes** *body*:: 's prog

**assumes** *hb*: *healthy* (*wp body*)

**and** *uP*: *unitary P*

**shows** *unitary* (*lfp-exp*  $(\lambda Q s. \ll G \gg s * wp \text{ body } Q s + \ll \mathcal{N} G \gg s * P s)$ )

$\langle proof \rangle$

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*:  $\bigwedge S. \ll \forall x \in S. P \text{ (fst } x) \text{ (snd } x); \forall x \in S. \text{feasible (fst } x);$   
 $\forall x \in S. \forall Q. \text{unitary } Q \longrightarrow \text{unitary (snd } x \text{ } Q) \gg \Longrightarrow$

$P \text{ (Sup-trans (fst ' } S)) \text{ (Inf-utrans (snd ' } S))$

— The property holds at limit points.

**and** *IH*:  $\bigwedge t u. \ll P t u; \text{feasible } t; \bigwedge Q. \text{unitary } Q \Longrightarrow \text{unitary (} u \text{ } Q) \gg \Longrightarrow$

$P \text{ (wp (body ;; Embed } t \ll G \gg \oplus \text{Skip}))}$

$\text{(wlp (body ;; Embed } u \ll G \gg \oplus \text{Skip}))}$

— The inductive step. The property is preserved by a single loop iteration.

**and** *P-equiv*:  $\bigwedge t' u' u. \ll P t u; \text{equiv-trans } t \text{ } t'; \text{equiv-utrans } u \text{ } u' \gg \Longrightarrow P t' u'$

— The property must be preserved by equivalence

**shows** *P* (*wp* (*do G*  $\longrightarrow$  *body od*)) (*wlp* (*do G*  $\longrightarrow$  *body od*))

— The property can refer to both interpretations simultaneously. The unifier will happily apply the rule to just one or the other, however.

$\langle proof \rangle$

#### 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  $\Rightarrow$  ('s  $\Rightarrow$  bool)  $\Rightarrow$  nat  $\Rightarrow$  's trans

**where** *iterates body G i* =  $((\lambda x. \text{wp (body ;; Embed } x \ll G \gg \oplus \text{Skip})) \wedge^{\wedge} i) (\lambda P s. 0)$

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

*iterates body G 0* =  $(\lambda P s. 0)$

$\langle proof \rangle$

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

The iterates themselves are all continuous.

**lemma cts-iterates:**  
**fixes** *body::'s prog*  
**assumes** *hb: healthy (wp body)*

**and** *cb*: *bd-cts* (*wp body*)  
**shows** *bd-cts* (*iterates body G i*)  
 ⟨*proof*⟩

Therefore so is the loop itself.

**lemma** *cts-wp-loop*:  
**fixes** *body*::'s *prog*  
**assumes** *hb*: *healthy* (*wp body*)  
**and** *cb*: *bd-cts* (*wp body*)  
**shows** *bd-cts* (*wp do G*  $\longrightarrow$  *body od*)  
 ⟨*proof*⟩

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

**end**

## 4.5 Sublinearity

**theory** *Sublinearity* **imports** *Embedding Healthiness LoopInduction* **begin**

### 4.5.1 Nonrecursive Primitives

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

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

**lemma** *sublinear-wp-Abort*:  
*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**  $x::'s \text{ prog}$   
**assumes**  $slx: \text{sublinear } (wp \ x)$  **and**  $sly: \text{sublinear } (wp \ y)$   
**and**  $uP: \text{unitary } P$   
**shows**  $\text{sublinear } (wp \ (x \ P \oplus \ y))$   
 $\langle \text{proof} \rangle$

**lemma** *sublinear-wp-DC*:  
**fixes**  $x::'s \text{ prog}$   
**assumes**  $slx: \text{sublinear } (wp \ x)$  **and**  $sly: \text{sublinear } (wp \ y)$   
**shows**  $\text{sublinear } (wp \ (x \ \sqcap \ y))$   
 $\langle \text{proof} \rangle$

As for continuity, we insist on a finite support.

**lemma** *sublinear-wp-SetPC*:  
**fixes**  $p::'a \Rightarrow 's \text{ prog}$   
**assumes**  $slp: \bigwedge s \ a. a \in \text{supp } (P \ s) \Longrightarrow \text{sublinear } (wp \ (p \ a))$   
**and**  $sum: \bigwedge s. (\sum a \in \text{supp } (P \ s). P \ s \ a) \leq 1$   
**and**  $nnP: \bigwedge s \ a. 0 \leq P \ s \ a$   
**and**  $fin: \bigwedge s. \text{finite } (\text{supp } (P \ s))$   
**shows**  $\text{sublinear } (wp \ (\text{SetPC } p \ P))$   
 $\langle \text{proof} \rangle$

**lemma** *sublinear-wp-SetDC*:  
**fixes**  $p::'a \Rightarrow 's \text{ prog}$   
**assumes**  $slp: \bigwedge s \ a. a \in S \ s \Longrightarrow \text{sublinear } (wp \ (p \ a))$   
**and**  $hp: \bigwedge s \ a. a \in S \ s \Longrightarrow \text{healthy } (wp \ (p \ a))$   
**and**  $ne: \bigwedge s. S \ s \neq \{\}$   
**shows**  $\text{sublinear } (wp \ (\text{SetDC } p \ S))$   
 $\langle \text{proof} \rangle$

**lemma** *sublinear-wp-Embed*:  
 $\text{sublinear } t \Longrightarrow \text{sublinear } (wp \ (\text{Embed } t))$   
 $\langle \text{proof} \rangle$

**lemma** *sublinear-wp-repeat*:  
 $\llbracket \text{sublinear } (wp \ p); \text{healthy } (wp \ p) \rrbracket \Longrightarrow \text{sublinear } (wp \ (\text{repeat } n \ p))$   
 $\langle \text{proof} \rangle$

**lemma** *sublinear-wp-Bind*:  
 $\llbracket \bigwedge s. \text{sublinear } (wp \ (a \ (f \ s))) \rrbracket \Longrightarrow \text{sublinear } (wp \ (\text{Bind } f \ a))$   
 $\langle \text{proof} \rangle$

## 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  $\longrightarrow$  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  $\longrightarrow$  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  $\longrightarrow$  body od))
<proof>

```

```

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

```

**end**

## 4.6 Determinism

**theory** *Determinism* **imports** *WellDefined* **begin**

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

### 4.6.1 Additivity

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

*wlp Abort* is not additive.

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

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

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

**lemma** *additive-wp-PC*:  
 $\llbracket \text{additive (wp } a); \text{additive (wp } b) \rrbracket \implies \text{additive (wp (} a \oplus b))$   
 ⟨*proof*⟩

*DC* is not additive.

**lemma** *additive-wp-SetPC*:  
 $\llbracket \bigwedge x s. x \in \text{supp } (p s) \implies \text{additive (wp (} a x)); \bigwedge s. \text{finite (supp } (p s)) \rrbracket \implies$   
*additive (wp (SetPC a p))*  
 ⟨*proof*⟩

**lemma** *additive-wp-Bind*:  
 $\llbracket \bigwedge x. \text{additive (wp (} a (f x))) \rrbracket \implies \text{additive (wp (Bind } f a))$   
 ⟨*proof*⟩

**lemma** *additive-wp-Embed*:  
 $\llbracket \text{additive } t \rrbracket \implies \text{additive (wp (Embed } t))$   
 ⟨*proof*⟩



**lemma** *additive-wp-repeat*:

$additive (wp\ a) \implies well\text{-}def\ a \implies additive (wp\ (repeat\ n\ a))$   
 ⟨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*:

$maximal (wp\ Skip)$   
 ⟨proof⟩

**lemma** *max-wp-Apply*:

$maximal (wp\ (Apply\ f))$   
 ⟨proof⟩

**lemma** *max-wp-Seq*:

$\llbracket maximal (wp\ a); maximal (wp\ b) \rrbracket \implies maximal (wp\ (a\ ;\ b))$   
 ⟨proof⟩

**lemma** *max-wp-PC*:

$\llbracket maximal (wp\ a); maximal (wp\ b) \rrbracket \implies maximal (wp\ (a\ p\oplus\ b))$   
 ⟨proof⟩

**lemma** *max-wp-DC*:

$\llbracket maximal (wp\ a); maximal (wp\ b) \rrbracket \implies maximal (wp\ (a\ \sqcap\ b))$   
 ⟨proof⟩

**lemma** *max-wp-SetPC*:

$\llbracket \bigwedge s\ a.\ a \in supp\ (P\ s) \implies maximal (wp\ (p\ a)); \bigwedge s.\ (\sum a \in supp\ (P\ s).\ P\ s\ a) = 1 \rrbracket \implies$   
 $maximal (wp\ (SetPC\ p\ P))$   
 ⟨proof⟩

**lemma** *max-wp-SetDC*:

**fixes**  $p::'a \Rightarrow 's\ prog$   
**assumes**  $mp: \bigwedge s\ a.\ a \in S\ s \implies maximal (wp\ (p\ a))$   
**and**  $ne: \bigwedge s.\ S\ s \neq \{\}$   
**shows**  $maximal (wp\ (SetDC\ p\ S))$   
 ⟨proof⟩

**lemma** *max-wp-Embed*:

$maximal\ t \implies maximal (wp\ (Embed\ t))$   
 ⟨proof⟩

**lemma** *max-wp-repeat*:  
 $\text{maximal } (wp\ a) \implies \text{maximal } (wp\ (\text{repeat } n\ a))$   
 $\langle \text{proof} \rangle$

**lemma** *max-wp-Bind*:  
**assumes**  $ma: \bigwedge s. \text{maximal } (wp\ (a\ (f\ s)))$   
**shows**  $\text{maximal } (wp\ (\text{Bind } f\ a))$   
 $\langle \text{proof} \rangle$

**lemmas** *max-intros* =  
 $\text{max-wp-Skip}$   $\text{max-wp-Apply}$   
 $\text{max-wp-Seq}$   $\text{max-wp-PC}$   
 $\text{max-wp-DC}$   $\text{max-wp-SetPC}$   
 $\text{max-wp-SetDC}$   $\text{max-wp-Embed}$   
 $\text{max-wp-Bind}$   $\text{max-wp-repeat}$

A healthy transformer that terminates is maximal.

**lemma** *healthy-term-max*:  
**assumes**  $ht: \text{healthy } t$   
**and**  $trm: \lambda s. I \Vdash t\ (\lambda s. I)$   
**shows**  $\text{maximal } t$   
 $\langle \text{proof} \rangle$

### 4.6.3 Determinism

**lemma** *det-wp-Skip*:  
 $\text{determ } (wp\ \text{Skip})$   
 $\langle \text{proof} \rangle$

**lemma** *det-wp-Apply*:  
 $\text{determ } (wp\ (\text{Apply } f))$   
 $\langle \text{proof} \rangle$

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

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

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

**lemma** *det-wp-Bind*:

$$(\bigwedge x. \text{determ} (\text{wp} (a (f x)))) \implies \text{determ} (\text{wp} (\text{Bind } f a))$$

*<proof>*

**lemma** *det-wp-Embed*:

$$\text{determ } t \implies \text{determ} (\text{wp} (\text{Embed } t))$$

*<proof>*

**lemma** *det-wp-repeat*:

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

*<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 conjunction 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 interpretations (*wp* and *wlp*).

**definition**

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

**where**

$$\text{wp-under-wlp } \text{prog} \equiv \forall P. \text{unitary } P \longrightarrow \text{wp } \text{prog } P \Vdash \text{wlp } \text{prog } P$$

**lemma** *wp-under-wlpI[intro]*:

$$\llbracket \bigwedge P. \text{unitary } P \implies \text{wp } \text{prog } P \Vdash \text{wlp } \text{prog } P \rrbracket \implies \text{wp-under-wlp } \text{prog}$$

*<proof>*

**lemma** *wp-under-wlpD*[*dest*]:

$\llbracket \text{wp-under-wlp prog; unitary } P \rrbracket \implies \text{wp prog } P \Vdash \text{wlp prog } P$   
 $\langle \text{proof} \rangle$

**lemma** *wp-under-le-trans*:

$\text{wp-under-wlp } a \implies \text{le-utrans } (\text{wp } a) (\text{wlp } a)$   
 $\langle \text{proof} \rangle$

**lemma** *wp-under-wlp-Abort*:

$\text{wp-under-wlp Abort}$   
 $\langle \text{proof} \rangle$

**lemma** *wp-under-wlp-Skip*:

$\text{wp-under-wlp Skip}$   
 $\langle \text{proof} \rangle$

**lemma** *wp-under-wlp-Apply*:

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

**lemma** *wp-under-wlp-Seq*:

**assumes** *h-wlp-a: nearly-healthy* ( $\text{wlp } a$ )  
**and** *h-wp-b: healthy* ( $\text{wp } b$ )  
**and** *h-wlp-b: nearly-healthy* ( $\text{wlp } b$ )  
**and** *wp-u-a: wp-under-wlp*  $a$   
**and** *wp-u-b: wp-under-wlp*  $b$   
**shows**  $\text{wp-under-wlp } (a ;; b)$   
 $\langle \text{proof} \rangle$

**lemma** *wp-under-wlp-PC*:

**assumes** *h-wp-a: healthy* ( $\text{wp } a$ )  
**and** *h-wlp-a: nearly-healthy* ( $\text{wlp } a$ )  
**and** *h-wp-b: healthy* ( $\text{wp } b$ )  
**and** *h-wlp-b: nearly-healthy* ( $\text{wlp } b$ )  
**and** *wp-u-a: wp-under-wlp*  $a$   
**and** *wp-u-b: wp-under-wlp*  $b$   
**and** *uP: unitary*  $P$   
**shows**  $\text{wp-under-wlp } (a \oplus b)$   
 $\langle \text{proof} \rangle$

**lemma** *wp-under-wlp-DC*:

**assumes** *wp-u-a: wp-under-wlp*  $a$   
**and** *wp-u-b: wp-under-wlp*  $b$   
**shows**  $\text{wp-under-wlp } (a \sqcap b)$   
 $\langle \text{proof} \rangle$

**lemma** *wp-under-wlp-SetPC*:

**assumes**  $\text{wp-u-f: } \bigwedge s a. a \in \text{supp } (P s) \implies \text{wp-under-wlp } (f a)$

**and**  $nP$ :  $\bigwedge s a. a \in \text{supp} (P s) \implies 0 \leq P s a$   
**shows**  $\text{wp-under-wlp} (\text{SetPC } f P)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{wp-under-wlp-SetDC}$ :  
**assumes**  $\text{wp-u-f}$ :  $\bigwedge s a. a \in S s \implies \text{wp-under-wlp} (f a)$   
**and**  $\text{hf}$ :  $\bigwedge s a. a \in S s \implies \text{healthy} (\text{wp} (f a))$   
**and**  $nS$ :  $\bigwedge s. S s \neq \{\}$   
**shows**  $\text{wp-under-wlp} (\text{SetDC } f S)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{wp-under-wlp-Embed}$ :  
 $\text{wp-under-wlp} (\text{Embed } t)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{wp-under-wlp-loop}$ :  
**fixes**  $\text{body} :: 's \text{ prog}$   
**assumes**  $\text{hwp}$ :  $\text{healthy} (\text{wp } \text{body})$   
**and**  $\text{hwlp}$ :  $\text{nearly-healthy} (\text{wlp } \text{body})$   
**and**  $\text{wp-under}$ :  $\text{wp-under-wlp } \text{body}$   
**shows**  $\text{wp-under-wlp} (\text{do } G \longrightarrow \text{body } \text{od})$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{wp-under-wlp-repeat}$ :  
 $\llbracket \text{healthy} (\text{wp } a); \text{nearly-healthy} (\text{wlp } a); \text{wp-under-wlp } a \rrbracket \implies$   
 $\text{wp-under-wlp} (\text{repeat } n a)$   
 $\langle \text{proof} \rangle$

**lemma**  $\text{wp-under-wlp-Bind}$ :  
 $\llbracket \bigwedge s. \text{wp-under-wlp} (a (f s)) \rrbracket \implies \text{wp-under-wlp} (\text{Bind } f a)$   
 $\langle \text{proof} \rangle$

**lemmas**  $\text{wp-under-wlp-intros} =$   
 $\text{wp-under-wlp-Abort } \text{wp-under-wlp-Skip}$   
 $\text{wp-under-wlp-Apply } \text{wp-under-wlp-Seq}$   
 $\text{wp-under-wlp-PC } \text{wp-under-wlp-DC}$   
 $\text{wp-under-wlp-SetPC } \text{wp-under-wlp-SetDC}$   
 $\text{wp-under-wlp-Embed } \text{wp-under-wlp-loop}$   
 $\text{wp-under-wlp-repeat } \text{wp-under-wlp-Bind}$

### 4.7.2 Sub-Distributivity of Conjunction

#### definition

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

#### where

$\text{sub-distrib-pconj } \text{prog} \equiv$   
 $\forall P Q. \text{unitary } P \longrightarrow \text{unitary } Q \longrightarrow$   
 $\text{wlp } \text{prog } P \ \&\& \ \text{wlp } \text{prog } Q \Vdash \text{wlp } \text{prog } (P \ \&\& \ Q)$

**lemma** *sub-distrib-pconjI*[*intro*]:  
 $\llbracket \wedge P Q. \llbracket \text{unitary } P; \text{unitary } Q \rrbracket \implies \text{wlp prog } P \ \&\& \ \text{wp prog } Q \Vdash \text{wp prog } (P \ \&\& \ Q) \rrbracket$   
 $\implies$   
*sub-distrib-pconj prog*  
 $\langle \text{proof} \rangle$

**lemma** *sub-distrib-pconjD*[*dest*]:  
 $\wedge P Q. \llbracket \text{sub-distrib-pconj prog}; \text{unitary } P; \text{unitary } Q \rrbracket \implies$   
 $\text{wlp prog } P \ \&\& \ \text{wp prog } Q \Vdash \text{wp prog } (P \ \&\& \ Q)$   
 $\langle \text{proof} \rangle$

**lemma** *sdp-Abort*:  
*sub-distrib-pconj Abort*  
 $\langle \text{proof} \rangle$

**lemma** *sdp-Skip*:  
*sub-distrib-pconj Skip*  
 $\langle \text{proof} \rangle$

**lemma** *sdp-Seq*:  
**fixes** *a* **and** *b*  
**assumes** *sdp-a*: *sub-distrib-pconj a*  
**and** *sdp-b*: *sub-distrib-pconj b*  
**and** *h-wp-a*: *healthy (wp a)*  
**and** *h-wp-b*: *healthy (wp b)*  
**and** *h-wlp-b*: *nearly-healthy (wlp b)*  
**shows** *sub-distrib-pconj (a ;; b)*  
 $\langle \text{proof} \rangle$

**lemma** *sdp-Apply*:  
*sub-distrib-pconj (Apply f)*  
 $\langle \text{proof} \rangle$

**lemma** *sdp-DC*:  
**fixes** *a::'s prog* **and** *b*  
**assumes** *sdp-a*: *sub-distrib-pconj a*  
**and** *sdp-b*: *sub-distrib-pconj b*  
**and** *h-wp-a*: *healthy (wp a)*  
**and** *h-wp-b*: *healthy (wp b)*  
**and** *h-wlp-b*: *nearly-healthy (wlp b)*  
**shows** *sub-distrib-pconj (a  $\sqcap$  b)*  
 $\langle \text{proof} \rangle$

**lemma** *sdp-PC*:  
**fixes** *a::'s prog* **and** *b*  
**assumes** *sdp-a*: *sub-distrib-pconj a*  
**and** *sdp-b*: *sub-distrib-pconj b*  
**and** *h-wp-a*: *healthy (wp a)*  
**and** *h-wp-b*: *healthy (wp b)*

**and**  $h\text{-wlp}\text{-}b$ : *nearly-healthy* ( $wlp\ b$ )  
**and**  $uP$ : *unitary*  $P$   
**shows** *sub-distrib-pconj* ( $a\ p\oplus\ b$ )  
 $\langle proof \rangle$

**lemma** *sdp-Embed*:  
 $\llbracket \bigwedge P\ Q. \llbracket \text{unitary } P; \text{unitary } Q \rrbracket \implies t\ P \ \&\&\ t\ Q \vdash t\ (P \ \&\&\ Q) \rrbracket \implies$   
*sub-distrib-pconj* (*Embed*  $t$ )  
 $\langle proof \rangle$

**lemma** *sdp-repeat*:  
**fixes**  $a::'s\ prog$   
**assumes**  $sdp$ : *sub-distrib-pconj*  $a$   
**and**  $hwp$ : *healthy* ( $wp\ a$ ) **and**  $hwlp$ : *nearly-healthy* ( $wlp\ a$ )  
**shows** *sub-distrib-pconj* (*repeat*  $n\ a$ ) (**is**  $?X\ n$ )  
 $\langle proof \rangle$

**lemma** *sdp-SetPC*:  
**fixes**  $p::'a \Rightarrow 's\ prog$   
**assumes**  $sdp$ :  $\bigwedge s\ a. a \in \text{supp } (P\ s) \implies \text{sub-distrib-pconj } (p\ a)$   
**and**  $fin$ :  $\bigwedge s. \text{finite } (\text{supp } (P\ s))$   
**and**  $nnp$ :  $\bigwedge s\ a. 0 \leq P\ s\ a$   
**and**  $sub$ :  $\bigwedge s. \text{sum } (P\ s) (\text{supp } (P\ s)) \leq 1$   
**shows** *sub-distrib-pconj* (*SetPC*  $p\ P$ )  
 $\langle proof \rangle$

**lemma** *sdp-SetDC*:  
**fixes**  $p::'a \Rightarrow 's\ prog$   
**assumes**  $sdp$ :  $\bigwedge s\ a. a \in S\ s \implies \text{sub-distrib-pconj } (p\ a)$   
**and**  $hwp$ :  $\bigwedge s\ a. a \in S\ s \implies \text{healthy } (wp\ (p\ a))$   
**and**  $hwlp$ :  $\bigwedge s\ a. a \in S\ s \implies \text{nearly-healthy } (wlp\ (p\ a))$   
**and**  $ne$ :  $\bigwedge s. S\ s \neq \{\}$   
**shows** *sub-distrib-pconj* (*SetDC*  $p\ S$ )  
 $\langle proof \rangle$

**lemma** *sdp-Bind*:  
 $\llbracket \bigwedge s. \text{sub-distrib-pconj } (p\ (f\ s)) \rrbracket \implies \text{sub-distrib-pconj } (\text{Bind } f\ p)$   
 $\langle proof \rangle$

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

**lemma** *sdp-loop*:  
**fixes**  $body::'s\ prog$   
**assumes**  $sdp\text{-}body$ : *sub-distrib-pconj*  $body$   
**and**  $hwlp$ : *nearly-healthy* ( $wlp\ body$ )  
**and**  $hwp$ : *healthy* ( $wp\ body$ )  
**shows** *sub-distrib-pconj* (*do*  $G \longrightarrow body\ od$ )  
 $\langle proof \rangle$

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

### 4.7.3 The Well-Defined Predicate.

**definition**

*well-def* :: 's prog ⇒ bool

**where**

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

**lemma** *well-defl[intro]*:

[[ *healthy* (wp prog); *nearly-healthy* (wlp prog);  
*wp-under-wlp* prog; *sub-distrib-pconj* prog; *sublinear* (wp prog);  
*bd-cts* (wp prog) ]] ⇒  
*well-def* prog  
 ⟨proof⟩

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

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

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

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

**lemma** *well-def-wp-under[dest]*:

*well-def* prog ⇒ *wp-under-wlp* prog  
 ⟨proof⟩

**lemma** *well-def-sdp[dest]*:

*well-def* prog ⇒ *sub-distrib-pconj* prog  
 ⟨proof⟩

**lemma** *well-def-wp-sublinear[dest]*:

*well-def* prog ⇒ *sublinear* (wp prog)  
 ⟨proof⟩

**lemma** *well-def-wp-cts[dest]*:

*well-def* prog ⇒ *bd-cts* (wp prog)  
 ⟨proof⟩

**lemmas** *wd-dests* =

*well-def-wp-healthy well-def-wlp-nearly-healthy*  
*well-def-wp-under well-def-sdp*



*well-def-wp-sublinear well-def-wp-cts*

**lemma** *wd-Abort:*

*well-def Abort*  
 $\langle \text{proof} \rangle$

**lemma** *wd-Skip:*

*well-def Skip*  
 $\langle \text{proof} \rangle$

**lemma** *wd-Apply:*

*well-def (Apply f)*  
 $\langle \text{proof} \rangle$

**lemma** *wd-Seq:*

$\llbracket \text{well-def } a; \text{well-def } b \rrbracket \implies \text{well-def } (a ;; b)$   
 $\langle \text{proof} \rangle$

**lemma** *wd-PC:*

$\llbracket \text{well-def } a; \text{well-def } b; \text{unitary } P \rrbracket \implies \text{well-def } (a \oplus b)$   
 $\langle \text{proof} \rangle$

**lemma** *wd-DC:*

$\llbracket \text{well-def } a; \text{well-def } b \rrbracket \implies \text{well-def } (a \sqcap b)$   
 $\langle \text{proof} \rangle$

**lemma** *wd-SetDC:*

$\llbracket \bigwedge x s. x \in S s \implies \text{well-def } (a x); \bigwedge s. S s \neq \{\};$   
 $\bigwedge s. \text{finite } (S s) \rrbracket \implies \text{well-def } (\text{SetDC } a S)$   
 $\langle \text{proof} \rangle$

**lemma** *wd-SetPC:*

$\llbracket \bigwedge x s. x \in (\text{supp } (p s)) \implies \text{well-def } (a x); \bigwedge s. \text{unitary } (p s); \bigwedge s. \text{finite } (\text{supp } (p s));$   
 $\bigwedge s. \text{sum } (p s) (\text{supp } (p s)) \leq 1 \rrbracket \implies \text{well-def } (\text{SetPC } a p)$   
 $\langle \text{proof} \rangle$

**lemma** *wd-Embed:*

**fixes** *t::'s trans*  
**assumes** *ht: healthy t and st: sublinear t and ct: bd-cts t*  
**shows** *well-def (Embed t)*  
 $\langle \text{proof} \rangle$

**lemma** *wd-repeat:*

*well-def a*  $\implies$  *well-def (repeat n a)*  
 $\langle \text{proof} \rangle$

**lemma** *wd-Bind:*

$\llbracket \bigwedge s. \text{well-def } (a (f s)) \rrbracket \implies \text{well-def } (\text{Bind } f a)$

$\langle proof \rangle$

**lemma** *wd-loop*:

$well-def\ body \implies well-def\ (do\ G \longrightarrow body\ od)$

$\langle proof \rangle$

**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**

$wp-inv :: ('s \Rightarrow bool) \Rightarrow 's\ prog \Rightarrow ('s \Rightarrow real) \Rightarrow bool$

**where**

$wp-inv\ G\ body\ I \longleftrightarrow (\forall s. \langle G \rangle s * I\ s \leq wp\ body\ I\ s)$

**lemma** *wp-invI*:

$\bigwedge I. (\bigwedge s. \langle G \rangle s * I\ s \leq wp\ body\ I\ s) \implies wp-inv\ G\ body\ I$

$\langle proof \rangle$

**definition**

$wlp-inv :: ('s \Rightarrow bool) \Rightarrow 's\ prog \Rightarrow ('s \Rightarrow real) \Rightarrow bool$

**where**

$wlp-inv\ G\ body\ I \longleftrightarrow (\forall s. \langle G \rangle s * I\ s \leq wlp\ body\ I\ s)$

**lemma** *wlp-invI*:

$\bigwedge I. (\bigwedge s. \langle G \rangle s * I\ s \leq wlp\ body\ I\ s) \implies wlp-inv\ G\ body\ I$

$\langle proof \rangle$

**lemma** *wlp-invD*:

$wlp-inv\ G\ body\ I \implies \langle G \rangle s * I\ s \leq wlp\ body\ I\ s$

$\langle proof \rangle$

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**  $\langle\!\langle G \rangle\!\rangle \ \&\& \ \langle\!\langle I \rangle\!\rangle \ \Vdash \ wp \ body \ \langle\!\langle I \rangle\!\rangle$   
 $\langle proof \rangle$

### 4.8.2 Partial Correctness

Partial correctness for loops [McIver and Morgan, 2004, Lemma 7.2.2, §7, p. 185].

**lemma** *wlp-Loop*:  
**assumes** *wd*: *well-def body*  
**and** *ul*: *unitary I*  
**and** *inv*: *wlp-inv G body I*  
**shows**  $I \leq wlp \ do \ G \ \longrightarrow \ body \ od \ (\lambda s. \ \langle\!\langle \mathcal{N} \ G \rangle\!\rangle \ s * I \ s)$   
 $(\text{is } I \leq wlp \ do \ G \ \longrightarrow \ body \ od \ ?P)$   
 $\langle proof \rangle$

### 4.8.3 Total Correctness

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

**lemma** *wp-Loop*:  
**assumes** *wd*: *well-def body*  
**and** *inv*: *wlp-inv G body I*  
**and** *unit*: *unitary I*  
**shows**  $I \ \&\& \ wp \ (do \ G \ \longrightarrow \ body \ od) \ (\lambda s. \ I) \ \Vdash \ wp \ (do \ G \ \longrightarrow \ body \ od) \ (\lambda s. \ \langle\!\langle \mathcal{N} \ G \rangle\!\rangle \ s * I \ s)$   
 $(\text{is } I \ \&\& \ ?T \ \Vdash \ wp \ ?loop \ ?X)$   
 $\langle proof \rangle$

### 4.8.4 Unfolding

**lemma** *wp-loop-unfold*:  
**fixes** *body* :: *'s prog*  
**assumes** *sP*: *sound P*  
**and** *h*: *healthy (wp body)*  
**shows**  $wp \ (do \ G \ \longrightarrow \ body \ od) \ P =$   
 $(\lambda s. \ \langle\!\langle \mathcal{N} \ G \rangle\!\rangle \ s * P \ s + \ \langle\!\langle G \rangle\!\rangle \ s * wp \ body \ (wp \ (do \ G \ \longrightarrow \ body \ od) \ P) \ s)$   
 $\langle proof \rangle$

**lemma** *wp-loop-nguard*:  
 $\llbracket \ healthy \ (wp \ body); \ sound \ P; \ \neg \ G \ s \rrbracket \Longrightarrow \ wp \ do \ G \ \longrightarrow \ body \ od \ P \ s = P \ s$   
 $\langle proof \rangle$

**lemma** *wp-loop-guard*:  
 $\llbracket \ healthy \ (wp \ body); \ sound \ P; \ G \ s \rrbracket \Longrightarrow$   
 $wp \ do \ G \ \longrightarrow \ body \ od \ P \ s = wp \ (body \ ;; \ do \ G \ \longrightarrow \ body \ od) \ P \ s$   
 $\langle proof \rangle$

end

## 4.9 The Algebra of pGCL

**theory Algebra imports WellDefined begin**

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

### 4.9.1 Program Refinement

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

**definition**

$refines :: 's prog \Rightarrow 's prog \Rightarrow bool$  (**infix**  $\sqsubseteq$  70)

**where**

$prog \sqsubseteq prog' \equiv \forall P. sound\ P \longrightarrow wp\ prog\ P \Vdash wp\ prog'\ P$

**lemma**  $refinesI$ [intro]:

$\llbracket \bigwedge P. sound\ P \Longrightarrow wp\ prog\ P \Vdash wp\ prog'\ P \rrbracket \Longrightarrow prog \sqsubseteq prog'$   
 $\langle proof \rangle$

**lemma**  $refinesD$ [dest]:

$\llbracket prog \sqsubseteq prog'; sound\ P \rrbracket \Longrightarrow wp\ prog\ P \Vdash wp\ prog'\ P$   
 $\langle proof \rangle$

The equivalence relation below will turn out to be that induced by refinement. It is also the application of *equiv-trans* to the weakest precondition.

**definition**

$pequiv :: 's prog \Rightarrow 's prog \Rightarrow bool$  (**infix**  $\simeq$  70)

**where**

$prog \simeq prog' \equiv \forall P. sound\ P \longrightarrow wp\ prog\ P = wp\ prog'\ P$

**lemma**  $pequivI$ [intro]:

$\llbracket \bigwedge P. sound\ P \Longrightarrow wp\ prog\ P = wp\ prog'\ P \rrbracket \Longrightarrow prog \simeq prog'$   
 $\langle proof \rangle$

**lemma**  $pequivD$ [dest,simp]:

$\llbracket prog \simeq prog'; sound\ P \rrbracket \Longrightarrow wp\ prog\ P = wp\ prog'\ P$   
 $\langle proof \rangle$

**lemma**  $pequiv-equiv-trans$ :

$a \simeq b \longleftrightarrow equiv-trans\ (wp\ a)\ (wp\ b)$

$\langle proof \rangle$

### 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 seperately

**lemma** *DC-comm*[*ac-simps*]:

$$a \sqcap b = b \sqcap a$$

$\langle proof \rangle$

**lemma** *DC-assoc*[*ac-simps*]:

$$a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c$$

$\langle proof \rangle$

**lemma** *DC-idem*:

$$a \sqcap a = a$$

$\langle proof \rangle$

**lemma** *AC-comm*[*ac-simps*]:

$$a \sqcup b = b \sqcup a$$

$\langle proof \rangle$

**lemma** *AC-assoc*[*ac-simps*]:

$$a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c$$

$\langle proof \rangle$

**lemma** *AC-idem*:

$$a \sqcup a = a$$

$\langle proof \rangle$

**lemma** *PC-quasi-comm*:

$$a \text{ } p \oplus b = b \text{ } (\lambda s. 1 - p \text{ } s) \oplus a$$

$\langle proof \rangle$

**lemma** *PC-idem*:

$$a \text{ } p \oplus a = a$$

$\langle proof \rangle$

**lemma** *Seq-assoc*[*ac-simps*]:

$$A ;; (B ;; C) = A ;; B ;; C$$

$\langle proof \rangle$

**lemma** *Abort-refines*[*intro*]:

$$\text{well-def } a \implies \text{Abort} \sqsubseteq a$$

$\langle proof \rangle$

**Laws relating demonic choice and refinement****lemma** *left-refines-DC*:
$$(a \sqcap b) \sqsubseteq a$$

*<proof>*

**lemma** *right-refines-DC*:
$$(a \sqcap b) \sqsubseteq b$$

*<proof>*

**lemma** *DC-refines*:

**fixes**  $a::s$  prog and  $b$  and  $c$   
**assumes**  $rab: a \sqsubseteq b$  and  $rac: a \sqsubseteq c$   
**shows**  $a \sqsubseteq (b \sqcap c)$   
*<proof>*

**lemma** *DC-mono*:

**fixes**  $a::s$  prog  
**assumes**  $rab: a \sqsubseteq b$  and  $rcd: c \sqsubseteq d$   
**shows**  $(a \sqcap c) \sqsubseteq (b \sqcap d)$   
*<proof>*

**Laws relating angelic choice and refinement****lemma** *left-refines-AC*:
$$a \sqsubseteq (a \sqcup b)$$

*<proof>*

**lemma** *right-refines-AC*:
$$b \sqsubseteq (a \sqcup b)$$

*<proof>*

**lemma** *AC-refines*:

**fixes**  $a::s$  prog and  $b$  and  $c$   
**assumes**  $rac: a \sqsubseteq c$  and  $rbc: b \sqsubseteq c$   
**shows**  $(a \sqcup b) \sqsubseteq c$   
*<proof>*

**lemma** *AC-mono*:

**fixes**  $a::s$  prog  
**assumes**  $rab: a \sqsubseteq b$  and  $rcd: c \sqsubseteq d$   
**shows**  $(a \sqcup c) \sqsubseteq (b \sqcup d)$   
*<proof>*

**Laws depending on the arithmetic of  $a_p \oplus b$  and  $a \sqcap b$  together****lemma** *DC-refines-PC*:

**assumes** *unit*: unitary  $p$   
**shows**  $(a \sqcap b) \sqsubseteq (a_p \oplus b)$   
*<proof>*

**Laws depending on the arithmetic of  $a \oplus b$  and  $a \sqcup b$  together****lemma** *PC-refines-AC*:**assumes** *unit*: unitary  $p$ **shows**  $(a \oplus b) \sqsubseteq (a \sqcup b)$  $\langle$ *proof* $\rangle$ **Laws depending on the arithmetic of  $a \sqcup b$  and  $a \sqcap b$  together****lemma** *DC-refines-AC*: $(a \sqcap b) \sqsubseteq (a \sqcup b)$  $\langle$ *proof* $\rangle$ **Laws Involving Refinement and Equivalence****lemma** *pr-trans*[*trans*]:**fixes**  $A:: 'a \text{ prog}$ **assumes** *prAB*:  $A \sqsubseteq B$ **and** *prBC*:  $B \sqsubseteq C$ **shows**  $A \sqsubseteq C$  $\langle$ *proof* $\rangle$ **lemma** *pequiv-refl*[*intro!*,*simp*]: $a \simeq a$  $\langle$ *proof* $\rangle$ **lemma** *pequiv-comm*[*ac-simps*]: $a \simeq b \iff b \simeq a$  $\langle$ *proof* $\rangle$ **lemma** *pequiv-pr*[*dest*]: $a \simeq b \implies a \sqsubseteq b$  $\langle$ *proof* $\rangle$ **lemma** *pequiv-trans*[*intro*,*trans*]: $\llbracket a \simeq b; b \simeq c \rrbracket \implies a \simeq c$  $\langle$ *proof* $\rangle$ **lemma** *pequiv-pr-trans*[*intro*,*trans*]: $\llbracket a \simeq b; b \sqsubseteq c \rrbracket \implies a \sqsubseteq c$  $\langle$ *proof* $\rangle$ **lemma** *pr-pequiv-trans*[*intro*,*trans*]: $\llbracket a \sqsubseteq b; b \simeq c \rrbracket \implies a \sqsubseteq c$  $\langle$ *proof* $\rangle$ 

Refinement induces equivalence by antisymmetry:

**lemma** *pequiv-antisym*: $\llbracket a \sqsubseteq b; b \sqsubseteq a \rrbracket \implies a \simeq b$  $\langle$ *proof* $\rangle$

**lemma** *pequiv-DC*:

$$\llbracket a \simeq c; b \simeq d \rrbracket \implies (a \sqcap b) \simeq (c \sqcap d)$$

*<proof>*

**lemma** *pequiv-AC*:

$$\llbracket a \simeq c; b \simeq d \rrbracket \implies (a \sqcup b) \simeq (c \sqcup d)$$

*<proof>*

### 4.9.3 Deterministic Programs are Maximal

Any sub-additive refinement of a deterministic program is in fact an equivalence. Deterministic programs are thus maximal (under the refinement order) among sub-additive programs.

**lemma** *refines-determ*:

**fixes** *a::'s prog*  
**assumes** *da: determ (wp a)*  
**and** *wa: well-def a*  
**and** *wb: well-def b*  
**and** *dr: a  $\sqsubseteq$  b*  
**shows** *a  $\simeq$  b*

Proof by contradiction.

*<proof>*

### 4.9.4 The Algebraic Structure of Refinement

Well-defined programs form a half-bounded semilattice under refinement, where *Abort* is bottom, and  $a \sqcap b$  is *inf*. There is no unique top element, but all fully-deterministic programs are maximal.

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

**quotient-type** *'s program =*

*'s prog / partial :  $\lambda a b. a \simeq b \wedge \text{well-def } a \wedge \text{well-def } b$*   
*<proof>*

**instantiation** *program :: (type) semilattice-inf begin*

**lift-definition**

*less-eq-program :: 'a program  $\Rightarrow$  'a program  $\Rightarrow$  bool is refines*  
*<proof>*

**lift-definition**

*less-program :: 'a program  $\Rightarrow$  'a program  $\Rightarrow$  bool*  
**is**  $\lambda a b. a \sqsubseteq b \wedge \neg b \sqsubseteq a$   
*<proof>*

**lift-definition**



*inf-program* :: 'a program  $\Rightarrow$  'a program  $\Rightarrow$  'a program **is** DC  
 <proof>

**instance**  
 <proof>  
**end**

**instantiation** *program* :: (type) bot **begin**

**lift-definition**

*bot-program* :: 'a program **is** Abort  
 <proof>

**instance** <proof>  
**end**

**lemma** *eq-det*:  $\bigwedge a b :: 's \text{ prog. } \llbracket a \simeq b; \text{determ } (wp \ a) \rrbracket \Longrightarrow \text{determ } (wp \ b)$   
 <proof>

**lift-definition**

*pdeterm* :: 's program  $\Rightarrow$  bool  
**is**  $\lambda a. \text{determ } (wp \ a)$   
 <proof>

**lemma** *determ-maximal*:

$\llbracket \text{pdeterm } a; a \leq x \rrbracket \Longrightarrow a = x$   
 <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  $\Rightarrow$  'a)  $\Rightarrow$  ('b  $\Rightarrow$  bool)  $\Rightarrow$  'a prog  $\Rightarrow$  'b prog  $\Rightarrow$  bool

**where**

*drefines*  $\varphi \ G \ A \ B \equiv \forall P \ Q. (\text{unitary } P \wedge \text{unitary } Q \wedge (P \Vdash wp \ A \ Q)) \longrightarrow$   
 $(\llbracket G \rrbracket \ \&\& \ (P \circ \varphi) \Vdash wp \ B \ (Q \circ \varphi))$

**lemma** *drefinesD[dest]*:

$\llbracket \text{drefines } \varphi \ G \ A \ B; \text{unitary } P; \text{unitary } Q; P \Vdash wp \ A \ Q \rrbracket \Longrightarrow$   
 $\llbracket G \rrbracket \ \&\& \ (P \circ \varphi) \Vdash wp \ B \ (Q \circ \varphi)$   
 <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**  $wpA: P \Vdash wp A Q$   
**and**  $G: G s$   
**shows**  $(P o \varphi) s \leq wp B (Q o \varphi) s$   
 $\langle proof \rangle$

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**  $wp a Q (\varphi s) \leq wp b (Q o \varphi) s$   
 $\langle proof \rangle$

**lemma** *drefinesI[intro]*:  
 $\llbracket \bigwedge P Q. \llbracket unitary P; unitary Q; P \Vdash wp A Q \rrbracket \implies$   
 $\langle G \rangle \&\& (P o \varphi) \Vdash wp B (Q o \varphi) \rrbracket \implies$   
 $drefines \varphi G A B$   
 $\langle proof \rangle$

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

**lemma** *drefinesI2*:  
**fixes**  $A::'a prog$   
**and**  $B::'b prog$   
**and**  $\varphi::'b \Rightarrow 'a$   
**and**  $G::'b \Rightarrow bool$   
**assumes**  $wB: well-def B$   
**and**  $withAs:$   
 $\bigwedge P Q s. \llbracket unitary P; unitary Q;$   
 $G s; P \Vdash wp A Q \rrbracket \implies (P o \varphi) s \leq wp B (Q o \varphi) s$   
**shows**  $drefines \varphi G A B$   
 $\langle proof \rangle$

**lemma** *dr-strengthen-guard*:  
**fixes**  $a::'s prog$  **and**  $b::'t prog$   
**assumes**  $fg: \bigwedge s. F s \implies G s$   
**and**  $drab: drefines \varphi G a b$   
**shows**  $drefines \varphi F a b$   
 $\langle proof \rangle$

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

**definition**

$pcorres :: ('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'a prog \Rightarrow 'b prog \Rightarrow bool$   
**where**  
 $pcorres \varphi G A B \longleftrightarrow$   
 $(\forall Q. unitary Q \longrightarrow \langle G \rangle \&\& (wp A Q o \varphi) = \langle G \rangle \&\& wp B (Q o \varphi))$

**lemma** *pcorresI*:

$\llbracket \bigwedge Q. \text{unitary } Q \implies \langle G \rangle \&\& (\text{wp } A \ Q \ o \ \varphi) = \langle G \rangle \&\& \text{wp } B \ (Q \ o \ \varphi) \rrbracket \implies$   
 $\text{pcorres } \varphi \ G \ A \ B$   
 ⟨proof⟩

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

**lemma** *pcorresI2[intro]*:

**fixes**  $A::'a \ \text{prog}$  **and**  $B::'b \ \text{prog}$   
**assumes**  $\text{with } G: \bigwedge Q \ s. \llbracket \text{unitary } Q; G \ s \rrbracket \implies \text{wp } A \ Q \ (\varphi \ s) = \text{wp } B \ (Q \ o \ \varphi) \ s$   
**and**  $wA: \text{well-def } A$   
**and**  $wB: \text{well-def } B$   
**shows**  $\text{pcorres } \varphi \ G \ A \ B$   
 ⟨proof⟩

**lemma** *pcorresD*:

$\llbracket \text{pcorres } \varphi \ G \ A \ B; \text{unitary } Q \rrbracket \implies \langle G \rangle \&\& (\text{wp } A \ Q \ o \ \varphi) = \langle G \rangle \&\& \text{wp } B \ (Q \ o \ \varphi)$   
 ⟨proof⟩

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

**lemma** *pcorresD2*:

**assumes**  $pc: \text{pcorres } \varphi \ G \ A \ B$   
**and**  $uQ: \text{unitary } Q$   
**and**  $wA: \text{well-def } A$  **and**  $wB: \text{well-def } B$   
**and**  $G: G \ s$   
**shows**  $\text{wp } A \ Q \ (\varphi \ s) = \text{wp } B \ (Q \ o \ \varphi) \ s$   
 ⟨proof⟩

## 4.9.6 The Algebra of Data Refinement

Program refinement implies a trivial data refinement:

**lemma** *refines-drefines*:

**fixes**  $a::'s \ \text{prog}$   
**assumes**  $rab: a \sqsubseteq b$  **and**  $wb: \text{well-def } b$   
**shows**  $\text{drefines } (\lambda s. s) \ G \ a \ b$   
 ⟨proof⟩

Data refinement is transitive:

**lemma** *dr-trans[trans]*:

**fixes**  $A::'a \ \text{prog}$  **and**  $B::'b \ \text{prog}$  **and**  $C::'c \ \text{prog}$   
**assumes**  $drAB: \text{drefines } \varphi \ G \ A \ B$   
**and**  $drBC: \text{drefines } \varphi' \ G' \ B \ C$   
**and**  $Gimp: \bigwedge s. G' \ s \implies G \ (\varphi' \ s)$   
**shows**  $\text{drefines } (\varphi \ o \ \varphi') \ G' \ A \ C$   
 ⟨proof⟩

Data refinement composes with program refinement:

**lemma** *pr-dr-trans[trans]*:

**assumes**  $prAB: A \sqsubseteq B$   
**and**  $drBC: drefines\ \varphi\ G\ B\ C$   
**shows**  $drefines\ \varphi\ G\ A\ C$   
 $\langle proof \rangle$

**lemma**  $dr-pr-trans[trans]$ :  
**assumes**  $drAB: drefines\ \varphi\ G\ A\ B$   
**assumes**  $prBC: B \sqsubseteq C$   
**shows**  $drefines\ \varphi\ G\ A\ C$   
 $\langle proof \rangle$

If the projection  $\varphi$  commutes with the transformer, then data refinement is reflexive:

**lemma**  $dr-refl$ :  
**assumes**  $wa: well-def\ a$   
**and**  $comm: \bigwedge Q. unitary\ Q \implies wp\ a\ Q\ o\ \varphi \Vdash wp\ a\ (Q\ o\ \varphi)$   
**shows**  $drefines\ \varphi\ G\ a\ a$   
 $\langle proof \rangle$

Correspondence implies data refinement

**lemma**  $pcorres-drefine$ :  
**assumes**  $corres: pcorres\ \varphi\ G\ A\ C$   
**and**  $wC: well-def\ C$   
**shows**  $drefines\ \varphi\ G\ A\ C$   
 $\langle proof \rangle$

Any *data* refinement of a deterministic program is correspondence. This is the analogous result to that relating program refinement and equivalence.

**lemma**  $drefines-determ$ :  
**fixes**  $a::'a\ prog$  **and**  $b::'b\ prog$   
**assumes**  $da: determ\ (wp\ a)$   
**and**  $wa: well-def\ a$   
**and**  $wb: well-def\ b$   
**and**  $dr: drefines\ \varphi\ G\ a\ b$   
**shows**  $pcorres\ \varphi\ G\ a\ b$

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

$\langle proof \rangle$

#### 4.9.7 Structural Rules for Correspondence

**lemma**  $pcorres-Skip$ :  
 $pcorres\ \varphi\ G\ Skip\ Skip$   
 $\langle proof \rangle$

Correspondence composes over sequential composition.

**lemma** *pcorres-Seq*:

**fixes**  $A::'b \text{ prog}$  **and**  $B::'c \text{ prog}$   
**and**  $C::'b \text{ prog}$  **and**  $D::'c \text{ prog}$   
**and**  $\varphi::'c \Rightarrow 'b$   
**assumes**  $pcAB: pcorres \varphi G A B$   
**and**  $pcCD: pcorres \varphi H C D$   
**and**  $wA: well-def A$  **and**  $wB: well-def B$   
**and**  $wC: well-def C$  **and**  $wD: well-def D$   
**and**  $p3p2: \bigwedge Q. \text{unitary } Q \Longrightarrow \langle I \rangle \&\& wp B Q = wp B (\langle H \rangle \&\& Q)$   
**and**  $p1p3: \bigwedge s. G s \Longrightarrow I s$   
**shows**  $pcorres \varphi G (A;;C) (B;;D)$   
 $\langle proof \rangle$

### 4.9.8 Structural Rules for Data Refinement

**lemma** *dr-Skip*:

**fixes**  $\varphi::'c \Rightarrow 'b$   
**shows**  $drefines \varphi G \text{Skip Skip}$   
 $\langle proof \rangle$

**lemma** *dr-Abort*:

**fixes**  $\varphi::'c \Rightarrow 'b$   
**shows**  $drefines \varphi G \text{Abort Abort}$   
 $\langle proof \rangle$

**lemma** *dr-Apply*:

**fixes**  $\varphi::'c \Rightarrow 'b$   
**assumes**  $commutes: f o \varphi = \varphi o g$   
**shows**  $drefines \varphi G (\text{Apply } f) (\text{Apply } g)$   
 $\langle proof \rangle$

**lemma** *dr-Seq*:

**assumes**  $drAB: drefines \varphi P A B$   
**and**  $drBC: drefines \varphi Q C D$   
**and**  $wpB: \langle P \rangle \vdash wp B \langle Q \rangle$   
**and**  $wB: well-def B$   
**and**  $wC: well-def C$   
**and**  $wD: well-def D$   
**shows**  $drefines \varphi P (A;;C) (B;;D)$   
 $\langle proof \rangle$

**lemma** *dr-repeat*:

**fixes**  $\varphi::'a \Rightarrow 'b$   
**assumes**  $dr-ab: drefines \varphi G a b$   
**and**  $Gpr: \langle G \rangle \vdash wp b \langle G \rangle$   
**and**  $wa: well-def a$   
**and**  $wb: well-def b$   
**shows**  $drefines \varphi G (\text{repeat } n a) (\text{repeat } n b) (\text{is } ?X n)$   
 $\langle proof \rangle$

end

## 4.10 Structured Reasoning

**theory** *StructuredReasoning* **imports** *Algebra* **begin**

By linking the algebraic, the syntactic, and the semantic views of computation, we derive a set of rules for decomposing expectation entailment proofs, firstly over the syntactic structure of a program, and secondly over the refinement relation. These rules also form the basis for automated reasoning.

### 4.10.1 Syntactic Decomposition

**lemma** *wp-Abort*:

$(\lambda s. 0) \Vdash wp \text{ Abort } Q$   
 $\langle \text{proof} \rangle$

**lemma** *wlp-Abort*:

$(\lambda s. 1) \Vdash wlp \text{ Abort } Q$   
 $\langle \text{proof} \rangle$

**lemma** *wp-Skip*:

$P \Vdash wp \text{ Skip } P$   
 $\langle \text{proof} \rangle$

**lemma** *wlp-Skip*:

$P \Vdash wlp \text{ Skip } P$   
 $\langle \text{proof} \rangle$

**lemma** *wp-Apply*:

$Q \circ f \Vdash wp (\text{Apply } f) Q$   
 $\langle \text{proof} \rangle$

**lemma** *wlp-Apply*:

$Q \circ f \Vdash wlp (\text{Apply } f) Q$   
 $\langle \text{proof} \rangle$

**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**  $P \Vdash wp (a ;; b) R$   
 $\langle \text{proof} \rangle$

**lemma** *wlp-Seq*:

**assumes** *ent-a*:  $P \Vdash \text{wlp } a \ Q$

**and** *ent-b*:  $Q \Vdash \text{wlp } b \ R$

**and** *wa*: *well-def a*

**and** *wb*: *well-def b*

**and** *u-Q*: *unitary Q*

**and** *u-R*: *unitary R*

**shows**  $P \Vdash \text{wlp } (a ;; b) \ R$

*<proof>*

**lemma** *wp-PC*:

$(\lambda s. P \ s * \text{wp } a \ Q \ s + (1 - P \ s) * \text{wp } b \ Q \ s) \Vdash \text{wp } (a \oplus b) \ Q$

*<proof>*

**lemma** *wlp-PC*:

$(\lambda s. P \ s * \text{wlp } a \ Q \ s + (1 - P \ s) * \text{wlp } b \ Q \ s) \Vdash \text{wlp } (a \oplus b) \ Q$

*<proof>*

A simpler rule for when the probability does not depend on the state.

**lemma** *PC-fixed*:

**assumes** *wpa*:  $P \Vdash a \ ab \ R$

**and** *wpb*:  $Q \Vdash b \ ab \ R$

**and** *np*:  $0 \leq p$  **and** *bp*:  $p \leq 1$

**shows**  $(\lambda s. p * P \ s + (1 - p) * Q \ s) \Vdash (a \ (\lambda s. p) \oplus b) \ ab \ R$

*<proof>*

**lemma** *wp-PC-fixed*:

$\llbracket P \Vdash \text{wp } a \ R; Q \Vdash \text{wp } b \ R; 0 \leq p; p \leq 1 \rrbracket \implies$

$(\lambda s. p * P \ s + (1 - p) * Q \ s) \Vdash \text{wp } (a \ (\lambda s. p) \oplus b) \ R$

*<proof>*

**lemma** *wlp-PC-fixed*:

$\llbracket P \Vdash \text{wlp } a \ R; Q \Vdash \text{wlp } b \ R; 0 \leq p; p \leq 1 \rrbracket \implies$

$(\lambda s. p * P \ s + (1 - p) * Q \ s) \Vdash \text{wlp } (a \ (\lambda s. p) \oplus b) \ R$

*<proof>*

**lemma** *wp-DC*:

$(\lambda s. \min (\text{wp } a \ Q \ s) (\text{wp } b \ Q \ s)) \Vdash \text{wp } (a \sqcap b) \ Q$

*<proof>*

**lemma** *wlp-DC*:

$(\lambda s. \min (\text{wlp } a \ Q \ s) (\text{wlp } b \ Q \ s)) \Vdash \text{wlp } (a \sqcap b) \ Q$

*<proof>*

Combining annotations for both branches:

**lemma** *DC-split*:

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

**assumes** *wpa*:  $P \Vdash a \ ab \ R$

**and** *wpb*:  $Q \Vdash b \ ab \ R$

**shows**  $(\lambda s. \min (P s) (Q s)) \Vdash (a \sqcap b) \text{ ab } R$   
 $\langle \text{proof} \rangle$

**lemma** *wp-DC-split*:

$\llbracket P \Vdash \text{wp prog } R; Q \Vdash \text{wp prog}' R \rrbracket \implies$   
 $(\lambda s. \min (P s) (Q s)) \Vdash \text{wp (prog } \sqcap \text{ prog}') R$   
 $\langle \text{proof} \rangle$

**lemma** *wlp-DC-split*:

$\llbracket P \Vdash \text{wlp prog } R; Q \Vdash \text{wlp prog}' R \rrbracket \implies$   
 $(\lambda s. \min (P s) (Q s)) \Vdash \text{wlp (prog } \sqcap \text{ prog}') R$   
 $\langle \text{proof} \rangle$

**lemma** *wp-DC-split-same*:

$\llbracket P \Vdash \text{wp prog } Q; P \Vdash \text{wp prog}' Q \rrbracket \implies P \Vdash \text{wp (prog } \sqcap \text{ prog}') Q$   
 $\langle \text{proof} \rangle$

**lemma** *wlp-DC-split-same*:

$\llbracket P \Vdash \text{wlp prog } Q; P \Vdash \text{wlp prog}' Q \rrbracket \implies P \Vdash \text{wlp (prog } \sqcap \text{ prog}') Q$   
 $\langle \text{proof} \rangle$

**lemma** *SetPC-split*:

**fixes**  $f:: 'x \Rightarrow 'y \text{ prog}$   
**and**  $p:: 'y \Rightarrow 'x \Rightarrow \text{real}$   
**assumes**  $\text{rec}: \bigwedge x s. x \in \text{supp } (p s) \implies P x \Vdash f x \text{ ab } Q$   
**and**  $\text{nnp}: \bigwedge s. \text{nneg } (p s)$   
**shows**  $(\lambda s. \sum x \in \text{supp } (p s). p s x * P x s) \Vdash \text{SetPC } f p \text{ ab } Q$   
 $\langle \text{proof} \rangle$

**lemma** *wp-SetPC-split*:

$\llbracket \bigwedge x s. x \in \text{supp } (p s) \implies P x \Vdash \text{wp } (f x) Q; \bigwedge s. \text{nneg } (p s) \rrbracket \implies$   
 $(\lambda s. \sum x \in \text{supp } (p s). p s x * P x s) \Vdash \text{wp (SetPC } f p) Q$   
 $\langle \text{proof} \rangle$

**lemma** *wlp-SetPC-split*:

$\llbracket \bigwedge x s. x \in \text{supp } (p s) \implies P x \Vdash \text{wlp } (f x) Q; \bigwedge s. \text{nneg } (p s) \rrbracket \implies$   
 $(\lambda s. \sum x \in \text{supp } (p s). p s x * P x s) \Vdash \text{wlp (SetPC } f p) Q$   
 $\langle \text{proof} \rangle$

**lemma** *wp-SetDC-split*:

$\llbracket \bigwedge s x. x \in S s \implies P \Vdash \text{wp } (f x) Q; \bigwedge s. S s \neq \{\} \rrbracket \implies$   
 $P \Vdash \text{wp (SetDC } f S) Q$   
 $\langle \text{proof} \rangle$

**lemma** *wlp-SetDC-split*:

$\llbracket \bigwedge s x. x \in S s \implies P \Vdash \text{wlp } (f x) Q; \bigwedge s. S s \neq \{\} \rrbracket \implies$   
 $P \Vdash \text{wlp (SetDC } f S) Q$   
 $\langle \text{proof} \rangle$



**lemma** *wp-SetDC*:

**assumes**  $wp: \bigwedge s x. x \in S s \implies P x \Vdash wp (f x) Q$   
**and**  $ne: \bigwedge s. S s \neq \{\}$   
**and**  $sP: \bigwedge x. sound (P x)$   
**shows**  $(\lambda s. Inf ((\lambda x. P x s) ' S s)) \Vdash wp (SetDC f S) Q$   
 $\langle proof \rangle$

**lemma** *wlp-SetDC*:

**assumes**  $wp: \bigwedge s x. x \in S s \implies P x \Vdash wlp (f x) Q$   
**and**  $ne: \bigwedge s. S s \neq \{\}$   
**and**  $sP: \bigwedge x. sound (P x)$   
**shows**  $(\lambda s. Inf ((\lambda x. P x s) ' S s)) \Vdash wlp (SetDC f S) Q$   
 $\langle proof \rangle$

**lemma** *wp-Embed*:

$P \Vdash t Q \implies P \Vdash wp (Embed t) Q$   
 $\langle proof \rangle$

**lemma** *wlp-Embed*:

$P \Vdash t Q \implies P \Vdash wlp (Embed t) Q$   
 $\langle proof \rangle$

**lemma** *wp-Bind*:

$\llbracket \bigwedge s. P s \leq wp (a (f s)) Q s \rrbracket \implies P \Vdash wp (Bind f a) Q$   
 $\langle proof \rangle$

**lemma** *wlp-Bind*:

$\llbracket \bigwedge s. P s \leq wlp (a (f s)) Q s \rrbracket \implies P \Vdash wlp (Bind f a) Q$   
 $\langle proof \rangle$

**lemma** *wp-repeat*:

$\llbracket P \Vdash wp a Q; Q \Vdash wp (repeat n a) R; \text{ well-def } a; \text{ sound } Q; \text{ sound } R \rrbracket \implies P \Vdash wp (repeat (Suc n) a) R$   
 $\langle proof \rangle$

**lemma** *wlp-repeat*:

$\llbracket P \Vdash wlp a Q; Q \Vdash wlp (repeat n a) R; \text{ well-def } a; \text{ unitary } Q; \text{ unitary } R \rrbracket \implies P \Vdash wlp (repeat (Suc n) a) R$   
 $\langle proof \rangle$

Note that the loop rules presented in section [Section 4.8](#) are of the same form, and would belong here, had they not already been stated.

The following rules are specialisations of those for general transformers, and are easier for the unifier to match.

**lemmas** *wp-strengthen-post=*

*entails-strengthen-post* [**where**  $t = wp a$  **for**  $a$ ]

**lemma** *wlp-strengthen-post*:

$$P \Vdash wlp\ a\ Q \implies \text{nearly-healthy } (wlp\ a) \implies \text{unitary } R \implies Q \Vdash R \implies \text{unitary } Q \implies \\ P \Vdash wlp\ a\ R \\ \langle \text{proof} \rangle$$

**lemmas** *wp-weaken-pre=*  
*entails-weaken-pre*[**where**  $t=wp\ a\ \text{for } a$ ]

**lemmas** *wlp-weaken-pre=*  
*entails-weaken-pre*[**where**  $t=wlp\ a\ \text{for } a$ ]

**lemmas** *wp-scale=*  
*entails-scale*[**where**  $t=wp\ a\ \text{for } a$ , *OF - well-def-wp-healthy*]

### 4.10.2 Algebraic Decomposition

Refinement is a powerful tool for decomposition, belied by the simplicity of the rule. This is an *axiomatic* formulation of refinement (all annotations of the  $a$  are annotations of  $b$ ), rather than an operational version (all traces of  $b$  are traces of  $a$ ).

**lemma** *wp-refines*:  
 $\llbracket a \sqsubseteq b; P \Vdash wp\ a\ Q; \text{sound } Q \rrbracket \implies P \Vdash wp\ b\ Q \\ \langle \text{proof} \rangle$

**lemmas** *wp-drefines = drefinesD*

### 4.10.3 Hoare triples

The Hoare triple, or validity predicate, is logically equivalent to the weakest-precondition entailment form. The benefit is that it allows us to define transitivity rules for computational (also/finally) reasoning.

**definition**  
 $wp\text{-valid} :: ('a \Rightarrow \text{real}) \Rightarrow 'a\ \text{prog} \Rightarrow ('a \Rightarrow \text{real}) \Rightarrow \text{bool} (\{\!\{-}\!\} - \{\!\{-}\!\}p)$   
**where**  
 $wp\text{-valid } P\ \text{prog } Q \equiv P \Vdash wp\ \text{prog } Q$

**lemma** *wp-validI*:  
 $P \Vdash wp\ \text{prog } Q \implies \{\!\{P\}\!\} \text{prog } \{\!\{Q\}\!\}p \\ \langle \text{proof} \rangle$

**lemma** *wp-validD*:  
 $\{\!\{P\}\!\} \text{prog } \{\!\{Q\}\!\}p \implies P \Vdash wp\ \text{prog } Q \\ \langle \text{proof} \rangle$

**lemma** *valid-Seq*:  
 $\llbracket \{\!\{P\}\!\} a\ \{\!\{Q\}\!\}p; \{\!\{Q\}\!\} b\ \{\!\{R\}\!\}p; \text{well-def } a; \text{well-def } b; \text{sound } Q; \text{sound } R \rrbracket \implies \\ \{\!\{P\}\!\} a\ ;\ ;\ b\ \{\!\{R\}\!\}p \\ \langle \text{proof} \rangle$

We make it available to the computational reasoner:

**declare** *valid-Seq*[*trans*]

**end**

## 4.11 Loop Termination

**theory** *Termination* **imports** *Embedding StructuredReasoning Loops* **begin**

Termination for loops can be shown by classical means (using a variant, or a measure function), or by probabilistic means: We only need that the loop terminates *with probability one*.

### 4.11.1 Trivial Termination

A maximal transformer (program) doesn't affect termination. This is essentially saying that such a program doesn't abort (or diverge).

**lemma** *maximal-Seq-term*:

**fixes**  $r :: 's \text{ prog}$  **and**  $s :: 's \text{ prog}$

**assumes**  $mr$ : *maximal* ( $wp\ r$ )

**and**  $ws$ : *well-def*  $s$

**and**  $ts$ :  $(\lambda s. I) \Vdash wp\ s\ (\lambda s. I)$

**shows**  $(\lambda s. I) \Vdash wp\ (r ;; s)\ (\lambda s. I)$

*<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**  $\langle \mathcal{N}\ G \rangle \Vdash wp\ do\ G \longrightarrow body\ od\ (\lambda s. I)$

*<proof>*

### 4.11.2 Classical Termination

The first non-trivial termination result is quite standard: If we can provide a natural-number-valued measure, that decreases on every iteration, and implies termination on reaching zero, the loop terminates.

**lemma** *loop-term-nat-measure-noinv*:

**fixes**  $m :: 's \Rightarrow nat$  **and**  $body :: 's \text{ prog}$

**assumes**  $wb$ : *well-def*  $body$

**and**  $guard$ :  $\bigwedge s. m\ s = 0 \longrightarrow \neg G\ s$

**and**  $variant$ :  $\bigwedge n. \langle \lambda s. m\ s = Suc\ n \rangle \Vdash wp\ body\ \langle \lambda s. m\ s = n \rangle$

**shows**  $\lambda s. I \Vdash wp\ do\ G \longrightarrow body\ od\ (\lambda s. I)$

*<proof>*

This version allows progress to depend on an invariant. Termination is then determined by the invariant's value in the initial state.

**lemma** *loop-term-nat-measure*:

**fixes**  $m :: 's \Rightarrow \text{nat}$  **and**  $\text{body} :: 's \text{ prog}$   
**assumes**  $\text{wb}$ : *well-def body*  
**and**  $\text{guard}$ :  $\bigwedge s. m\ s = 0 \longrightarrow \neg G\ s$   
**and**  $\text{variant}$ :  $\bigwedge n. \langle \lambda s. m\ s = \text{Suc}\ n \rangle \&\& \langle I \rangle \Vdash \text{wp}\ \text{body}\ \langle \lambda s. m\ s = n \rangle$   
**and**  $\text{inv}$ :  $\text{wp-inv}\ G\ \text{body}\ \langle I \rangle$   
**shows**  $\langle I \rangle \Vdash \text{wp}\ \text{do}\ G \longrightarrow \text{body}\ \text{od}\ (\lambda s. I)$   
 $\langle \text{proof} \rangle$

### 4.11.3 Probabilistic Termination

Any loop that has a non-zero chance of terminating after each step terminates with probability 1.

**lemma** *termination-0-1*:  
**fixes**  $\text{body} :: 's \text{ prog}$   
**assumes**  $\text{wb}$ : *well-def body*  
— The loop terminates in one step with nonzero probability  
**and**  $\text{onestep}$ :  $(\lambda s. p) \Vdash \text{wp}\ \text{body}\ \langle \mathcal{N}\ G \rangle$   
**and**  $\text{nzp}$ :  $0 < p$   
— The body is maximal i.e. it terminates absolutely.  
**and**  $\text{mb}$ : *maximal (wp body)*  
**shows**  $\lambda s. I \Vdash \text{wp}\ \text{do}\ G \longrightarrow \text{body}\ \text{od}\ (\lambda s. I)$   
 $\langle \text{proof} \rangle$

**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**  $\text{wd}$   
*theorems to automatically establish well-definedness*  
**named-theorems**  $\text{pwp-core}$   
*core probabilistic wp rules, for evaluating primitive terms*  
**named-theorems**  $\text{pwp}$   
*user-supplied probabilistic wp rules*  
**named-theorems**  $\text{pwl}$   
*user-supplied probabilistic wlp rules*

$\langle \text{ML} \rangle$

**declare**  $\text{wd-intros}[\text{wd}]$

**lemmas**  $\text{core-wp-rules} =$   
 $\text{wp-Skip}$      $\text{wlp-Skip}$   
 $\text{wp-Abort}$      $\text{wlp-Abort}$

```
wp-Apply    wlp-Apply  
wp-Seq     wlp-Seq  
wp-DC-split wlp-DC-split  
wp-PC-fixed wlp-PC-fixed  
wp-SetDC   wlp-SetDC  
wp-SetPC-split wlp-SetPC-split
```

```
declare core-wp-rules[pwp-core]
```

```
end
```



# Additional Material

## 4.13 Miscellaneous Mathematics

**theory** *Misc*

**imports**

*HOL-Analysis.Multivariate-Analysis*

**begin lemma** *sum-UNIV*:

**fixes** *S::'a::finite set*

**assumes** *complete*:  $\bigwedge x. x \notin S \implies f x = 0$

**shows**  $\text{sum } f S = \text{sum } f UNIV$

*<proof>*

**lemma** *cInf-mono*:

**fixes** *A::'a::conditionally-complete-lattice set*

**assumes** *lower*:  $\bigwedge b. b \in B \implies \exists a \in A. a \leq b$

**and** *bounded*:  $\bigwedge a. a \in A \implies c \leq a$

**and** *ne*:  $B \neq \{\}$

**shows**  $\text{Inf } A \leq \text{Inf } B$

*<proof>*

**lemma** *max-distrib*:

**fixes** *c::real*

**assumes** *nn*:  $0 \leq c$

**shows**  $c * \text{max } a b = \text{max } (c * a) (c * b)$

*<proof>*

**lemma** *mult-div-mono-left*:

**fixes** *c::real*

**assumes** *nnc*:  $0 \leq c$  **and** *nzc*:  $c \neq 0$

**and** *inv*:  $a \leq \text{inverse } c * b$

**shows**  $c * a \leq b$

*<proof>*

**lemma** *mult-div-mono-right*:

**fixes** *c::real*

**assumes** *nnc*:  $0 \leq c$  **and** *nzc*:  $c \neq 0$

**and** *inv*:  $\text{inverse } c * a \leq b$

**shows**  $a \leq c * b$

*<proof>*

**lemma** *min-distrib*:

**fixes**  $c::real$

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

**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**  $ne: S \neq \{\}$

**shows**  $\exists x \in S. \forall y \in S. x \leq y$

*<proof>*

**lemma** *cSup-add*:

**fixes**  $c::real$

**assumes**  $ne: S \neq \{\}$

**and**  $bS: \bigwedge x. x \in S \implies x \leq b$

**shows**  $Sup\ S + c = Sup\ \{x + c \mid x. x \in S\}$

*<proof>*

**lemma** *cSup-mult*:

**fixes**  $c::real$

**assumes**  $ne: S \neq \{\}$

**and**  $bS: \bigwedge x. x \in S \implies x \leq b$

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

**shows**  $c * Sup\ S = Sup\ \{c * x \mid x. x \in S\}$

*<proof>*

**lemma** *closure-contains-Sup*:

**fixes**  $S :: real\ set$

**assumes**  $neS: S \neq \{\}$  **and**  $bS: \forall x \in S. x \leq B$

**shows**  $Sup\ S \in closure\ S$

*<proof>*

**lemma** *tendsto-min*:

**fixes**  $x\ y::real$

**assumes**  $ta: a \longrightarrow x$

**and**  $tb: b \longrightarrow y$

**shows**  $(\lambda i. \min (a\ i) (b\ i)) \longrightarrow \min\ x\ y$

*<proof>*

**definition** *supp* ::  $('s \Rightarrow real) \Rightarrow 's\ set$

**where**  $supp\ f = \{x. f\ x \neq 0\}$

**definition** *dist-remove* ::  $('s \Rightarrow real) \Rightarrow 's \Rightarrow 's \Rightarrow real$

**where**  $dist\ remove\ p\ x = (\lambda y. if\ y=x\ then\ 0\ else\ p\ y / (1 - p\ x))$

**lemma** *supp-dist-remove*:



$$p x \neq 0 \implies p x \neq 1 \implies \text{supp } (\text{dist-remove } p x) = \text{supp } p - \{x\}$$

*<proof>*

**lemma** *supp-empty*:

$$\text{supp } f = \{\} \implies f x = 0$$

*<proof>*

**lemma** *nsupp-zero*:

$$x \notin \text{supp } f \implies f x = 0$$

*<proof>*

**lemma** *sum-supp*:

**fixes**  $f :: 'a :: \text{finite} \Rightarrow \text{real}$

**shows**  $\text{sum } f (\text{supp } f) = \text{sum } f \text{ UNIV}$

*<proof>*

### 4.13.1 Truncated Subtraction

**definition**

*tminus*  $:: \text{real} \Rightarrow \text{real} \Rightarrow \text{real}$  (**infixl**  $\ominus$  60)

**where**

$$x \ominus y = \max (x - y) 0$$

**lemma** *minus-le-tminus*[*intro!,simp*]:

$$a - b \leq a \ominus b$$

*<proof>*

**lemma** *tminus-cancel-1*:

$$0 \leq a \implies a + 1 \ominus 1 = a$$

*<proof>*

**lemma** *tminus-zero-imp-le*:

$$x \ominus y \leq 0 \implies x \leq y$$

*<proof>*

**lemma** *tminus-zero*[*simp*]:

$$0 \leq x \implies x \ominus 0 = x$$

*<proof>*

**lemma** *tminus-left-mono*:

$$a \leq b \implies a \ominus c \leq b \ominus c$$

*<proof>*

**lemma** *tminus-less*:

$$\llbracket 0 \leq a; 0 \leq b \rrbracket \implies a \ominus b \leq a$$

*<proof>*

**lemma** *tminus-left-distrib*:

**assumes**  $nna: 0 \leq a$

**shows**  $a * (b \ominus c) = a * b \ominus a * c$   
 ⟨proof⟩

**lemma** *tminus-le[simp]*:  
 $b \leq a \implies a \ominus b = a - b$   
 ⟨proof⟩

**lemma** *tminus-le-alt[simp]*:  
 $a \leq b \implies a \ominus b = 0$   
 ⟨proof⟩

**lemma** *tminus-nle[simp]*:  
 $\neg b \leq a \implies a \ominus b = 0$   
 ⟨proof⟩

**lemma** *tminus-add-mono*:  
 $(a+b) \ominus (c+d) \leq (a \ominus c) + (b \ominus d)$   
 ⟨proof⟩

**lemma** *tminus-sum-mono*:  
**assumes**  $fS$ : *finite S*  
**shows**  $\text{sum } f S \ominus \text{sum } g S \leq \text{sum } (\lambda x. f x \ominus g x) S$   
 (**is** ?X S)  
 ⟨proof⟩

**lemma** *tminus-nneg[simp,intro]*:  
 $0 \leq a \ominus b$   
 ⟨proof⟩

**lemma** *tminus-right-antimono*:  
**assumes**  $clb$ :  $c \leq b$   
**shows**  $a \ominus b \leq a \ominus c$   
 ⟨proof⟩

**lemma** *min-tminus-distrib*:  
 $\min a b \ominus c = \min (a \ominus c) (b \ominus c)$   
 ⟨proof⟩

**end**

# Bibliography

- David Cock. Verifying probabilistic correctness in Isabelle with pGCL. In *Proceedings of the 7th Systems Software Verification*, pages 1–10, Sydney, Australia, November 2012. doi: 10.4204/EPTCS.102.15.
- David Cock. Practical probability: Applying pGCL to lattice scheduling. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, pages 1–16, Rennes, France, July 2013. doi: 10.1007/978-3-642-39634-2\_23.
- David Cock. From probabilistic operational semantics to information theory - side channels with pGCL in Isabelle. In *Proceedings of the 5th International Conference on Interactive Theorem Proving*, pages 1–15, Vienna, Austria, July 2014a. Springer.
- David Cock. *Leakage in Trustworthy Systems*. PhD thesis, University of New South Wales, 2014b.
- Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975. ISSN 0001-0782. doi: 10.1145/360933.360975.
- Joe Hurd, Annabelle McIver, and Carroll Morgan. Probabilistic guarded commands mechanized in HOL. *Theoretical Computer Science*, 346(1):96 – 112, 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2005.08.005. URL <http://www.sciencedirect.com/science/article/pii/S0304397505004767>.
- Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer, 2004.
- Steve Selvin. A problem in probability (letter to the editor). *American Statistician*, 29(1):67, Feb 1975.