pGCL for Isabelle

David Cock

March 17, 2025

Contents

Chapter 1

Overview

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

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

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

Chapter 2

Introduction to pGCL

2.1 Language Primitives

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

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

2.1.1 The Basics

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

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

record *coins* =
  a :: *coin*
  b :: *coin*

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

lemma
  $Apply\ (\lambda s.\ s\ \llparenthesis\ a := Heads\ \rrparenthesis) = (a := (\lambda s.\ Heads))$
  $\langle proof \rangle$

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* ($\lambda s.\ s\llparenthesis a := b\ s\rrparenthesis$), 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 $P{::}'s \Rightarrow \mathit{bool}$ is embedded as $«\, P \,»{::}'s \Rightarrow \mathit{real}$, such that $P\ s \longrightarrow «\, P \,»\ s = \mathit{1} \wedge \neg\, P\ s \longrightarrow «\, P \,»\ s = \mathit{0}$.

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

lemma
$\bigwedge x.\ «\lambda s.\ b\ s = x» \Vdash \mathit{wp}\ (a := b)\ «\lambda s.\ a\ s = x»$
$\langle \mathit{proof} \rangle$

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

2.1.3 Probability

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

lemma
$«\lambda s{::}\mathit{real}.\ s \neq \mathit{0}» \Vdash \mathit{wp}\ (\mathit{Apply}\ ((*)\ 2)\ ;;\ \mathit{Apply}\ (\lambda s.\ s\ /\ s))\ «\lambda s.\ s = \mathit{1}»$
$\langle \mathit{proof} \rangle$

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

definition

$\mathit{flip\text{-}a} :: \mathit{real} \Rightarrow \mathit{coins\ prog}$
where
$\mathit{flip\text{-}a}\ p = a := (\lambda\text{-}.\ \mathit{Heads})\ {}_{(\lambda s.\ p)}{\oplus}\ a := (\lambda\text{-}.\ \mathit{Tails})$

lemma
$(\lambda s.\ 1/2) = \mathit{wp}\ (\mathit{flip\text{-}a}\ (1/2))\ «\lambda s.\ a\ s = \mathit{Heads}»$
$\langle \mathit{proof} \rangle$

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
$\lambda s.\ 1/3 \Vdash \mathit{wp}\ (\mathit{flip\text{-}a}\ (2/3) \sqcap \mathit{flip\text{-}a}\ (1/3))\ «\lambda s.\ a\ s = \mathit{Heads}»$
$\langle \mathit{proof} \rangle$

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 $1\ /\ (4{::}'a)$.

definition
$\mathit{flip\text{-}b} :: \mathit{real} \Rightarrow \mathit{coins\ prog}$
where
$\mathit{flip\text{-}b}\ p = b := (\lambda\text{-}.\ \mathit{Heads})\ {}_{(\lambda s.\ p)}{\oplus}\ b := (\lambda\text{-}.\ \mathit{Tails})$

lemma
$(\lambda s.\ 1/4) = \mathit{wp}\ (\mathit{flip\text{-}a}\ (1/2)\ ;;\ \mathit{flip\text{-}b}\ (1/2))$
$\qquad «\lambda s.\ a\ s = \mathit{Heads} \wedge b\ s = \mathit{Heads}»$
$\langle \mathit{proof} \rangle$

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 $\mathit{dice} =$
$\mathit{red}\ :: \mathit{nat}$
$\mathit{blue} :: \mathit{nat}$

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

lemma $\mathit{Puniform\text{-}in}$:
$x \in S \Longrightarrow \mathit{Puniform}\ S\ x = 1\ /\ \mathit{card}\ S$
$\langle \mathit{proof} \rangle$

lemma *Puniform-out*:
$x \notin S \Longrightarrow \mathit{Puniform}\ S\ x = 0$
$\langle \mathit{proof} \rangle$

lemma *supp-Puniform*:
$\mathit{finite}\ S \Longrightarrow \mathit{supp}\ (\mathit{Puniform}\ S) = S$
$\langle \mathit{proof} \rangle$

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

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

The expectations of independent variables add:

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

end

2.2 Loops

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

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

2.2.1 Guaranteed Termination

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

definition *countdown* :: *int prog*
where
$\mathit{countdown} = \mathit{do}\ (\lambda x.\ 0 < x) \longrightarrow \mathit{Apply}\ (\lambda s.\ s - 1)\ \mathit{od}$

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

definition *inv-count* :: $\mathit{int} \Rightarrow \mathit{bool}$
where
$\mathit{inv\text{-}count} = (\lambda x.\ 0 \leq x)$

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

lemma *wp-inv-count*:
$wp\text{-}inv\ (\lambda x.\ 0 < x)\ (Apply\ (\lambda s.\ s - 1))\ «inv\text{-}count»$
$\langle proof\rangle$

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

lemma *term-countdown*:
$«inv\text{-}count» \Vdash wp\ countdown\ (\lambda s.\ 1)$
$\langle proof\rangle$

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

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

definition
wait-for-heads :: *coin prog*
where
$wait\text{-}for\text{-}heads = do\ ((\neq)\ Heads) \longrightarrow flip\ od$

Nonetheless, we can show termination .

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

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

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

2.3.1 The State Space

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

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

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

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

Invariants

We prove explicitly that only valid doors are ever chosen.

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

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

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

2.3.2 The Game

Hide the prize behind door D.

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

Choose door *D*.

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

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

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

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

definition $\mathit{hide\text{-}prize} :: \mathit{game\ prog}$
where $\mathit{hide\text{-}prize} \equiv \mathit{hide\text{-}behind}\ \mathit{1} \bigsqcap \mathit{hide\text{-}behind}\ \mathit{2} \bigsqcap \mathit{hide\text{-}behind}\ \mathit{3}$

Guess uniformly at random.

definition $\mathit{make\text{-}guess} :: \mathit{game\ prog}$
where $\mathit{make\text{-}guess} \equiv \mathit{guess\text{-}behind}\ \mathit{1}\ {}_{(\lambda s.\ 1/3)}\!\oplus$
$\quad \mathit{guess\text{-}behind}\ \mathit{2}\ {}_{(\lambda s.\ 1/2)}\!\oplus\ \mathit{guess\text{-}behind}\ \mathit{3}$

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

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

Switch your guess to the other unopened door.

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

The complete game, either with or without switching guesses.

definition $\mathit{monty} :: \mathit{bool} \Rightarrow \mathit{game\ prog}$
where

$$\begin{aligned}
&\mathit{monty\ switch} \equiv \mathit{hide\text{-}prize}\ ;;\\
&\qquad\qquad \mathit{make\text{-}guess}\ ;;\\
&\qquad\qquad \mathit{reveal}\ ;;\\
&\qquad\qquad (\mathit{if\ switch\ then\ switch\text{-}guess\ else\ Skip})
\end{aligned}$$

2.3.3 A Brute Force Solution

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

lemma $\mathit{eval\text{-}win}[\mathit{simp}]$:
$p = g \Longrightarrow$ «*player-wins*» $(s\llparenthesis\ \mathit{prize} := p,\ \mathit{guess} := g,\ \mathit{clue} := c\ \rrparenthesis) = \mathit{1}$
$\langle \mathit{proof} \rangle$

lemma $\mathit{eval\text{-}loss}[\mathit{simp}]$:
$p \neq g \Longrightarrow$ «*player-wins*» $(s\llparenthesis\ \mathit{prize} := p,\ \mathit{guess} := g,\ \mathit{clue} := c\ \rrparenthesis) = \mathit{0}$
$\langle \mathit{proof} \rangle$

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

lemma *wp-monty-noswitch*:
 $(\lambda s.\ 1/3) = wp\ (monty\ False)$ «*player-wins*»
 ⟨*proof*⟩

lemma *swap-upd*:
 $s(\!|\ prize := p,\ clue := c,\ guess := g\ |\!) =$
 $s(\!|\ prize := p,\ guess := g,\ clue := c\ |\!)$
 ⟨*proof*⟩

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*:
 $(\lambda s.\ 2/3) = wp\ (monty\ True)$ «*player-wins*»
 ⟨*proof*⟩

2.3.4 A Modular Approach

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

Healthiness

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

lemma *wd-hide-prize*:
 well-def hide-prize
 ⟨*proof*⟩

lemma *wd-make-guess*:
 well-def make-guess
 ⟨*proof*⟩

lemma *wd-reveal*:
 well-def reveal
⟨*proof*⟩

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

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

Annotations

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

hide-prize chooses a valid door.

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

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

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

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

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

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

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

```
lemma distinct-game:
  ⟦ guess g ≠ prize g; clue g ≠ prize g; clue g ≠ guess g;
    inv-prize g; inv-guess g; inv-clue g ⟧ ⟹
  {1, 2, 3} = {guess g, prize g, clue g}
  ⟨proof⟩
```

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

```
lemma wp-switch-guess:
  «λg. guess g ≠ prize g ∧ clue g ≠ prize g ∧ clue g ≠ guess g ∧
       inv-prize g ∧ inv-guess g ∧ inv-clue g» ⊩
   wp switch-guess «player-wins»
⟨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* (*monty True*) «*player-wins*»
⟨*proof*⟩

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* (*monty True*) «*player-wins*»
  ⟨*proof*⟩

end

Chapter 3

Semantic Structures

3.1 Expectations

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

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

a	b	$a \to b$	x	y	$x \leq y$
F	F	T	0	0	T
F	T	T	0	1	T
T	F	F	1	0	F
T	T	T	1	1	T

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 $0.7 \times 2.0 + 0.3 \times 3.0 = 2.3$.

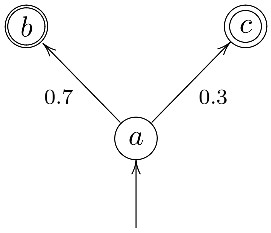

Figure 3.1: A probabilistic automaton

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* :: $real \Rightarrow ('a \Rightarrow real) \Rightarrow bool$
where *bounded-by b P* $\equiv \forall\, x.\, P\, x \leq b$

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

lemma *bounded-byI*[*intro*]:
$[\![\bigwedge x.\, P\, x \leq b]\!] \Longrightarrow$ *bounded-by b P*
$\langle proof \rangle$

lemma *bounded-byI2*[*intro*]:
$P \leq (\lambda s.\, b) \Longrightarrow$ *bounded-by b P*
$\langle proof \rangle$

lemma *bounded-byD*[*dest*]:
bounded-by b P $\Longrightarrow P\, x \leq b$
$\langle proof \rangle$

lemma *bounded-byD2*[*dest*]:
bounded-by b P $\Longrightarrow P \leq (\lambda s.\, b)$
$\langle proof \rangle$

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

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

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

definition *bound-of* :: $('a \Rightarrow real) \Rightarrow real$
where *bound-of P* $\equiv$ *Sup* $(P$ ‘ *UNIV*$)$

lemma *bounded-bdd-above*[*intro*]:
assumes *bP*: *bounded P*
shows *bdd-above* (*range P*)
$\langle proof \rangle$

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*
$\langle$*proof*$\rangle$

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

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

lemma *bounded-by-mono*:
$[\![$ *bounded-by a P*; *a* $\leq$ *b* $]\!]$ $\Longrightarrow$ *bounded-by b P*
$\langle$*proof*$\rangle$

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

This is occasionally easier to apply:

lemma *bounded-by-bound-of-alt*:
$[\![$ *bounded P*; *bound-of P* = *a* $]\!]$ $\Longrightarrow$ *bounded-by a P*
$\langle$*proof*$\rangle$

lemma *bounded-const*[*simp*]:
bounded (λ*x*. *c*)
$\langle$*proof*$\rangle$

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

lemma *bounded-by-mono-alt*[*intro*]:
$[\![$ *bounded-by b Q*; *P* $\leq$ *Q* $]\!]$ $\Longrightarrow$ *bounded-by b P*
$\langle$*proof*$\rangle$

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

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

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

$[\![\, P \le Q;\ \mathit{bounded}\ P;\ \mathit{bounded}\ Q \,]\!] \Longrightarrow \mathit{bound\text{-}of}\ P \le \mathit{bound\text{-}of}\ Q$
⟨*proof*⟩

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

lemma *le-bound-of*[*intro*]:
$\bigwedge x.\ \mathit{bounded}\ f \Longrightarrow f\ x \le \mathit{bound\text{-}of}\ f$
⟨*proof*⟩

3.1.2 Non-Negative Functions.

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

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

lemma *nnegI*[*intro*]:
$[\![\, \bigwedge x.\ 0 \le P\ x \,]\!] \Longrightarrow \mathit{nneg}\ P$
⟨*proof*⟩

lemma *nnegI2*[*intro*]:
$(\lambda s.\ 0) \le P \Longrightarrow \mathit{nneg}\ P$
⟨*proof*⟩

lemma *nnegD*[*dest*]:
$\mathit{nneg}\ P \Longrightarrow 0 \le P\ x$
⟨*proof*⟩

lemma *nnegD2*[*dest*]:
$\mathit{nneg}\ P \Longrightarrow (\lambda s.\ 0) \le P$
⟨*proof*⟩

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

lemma *nneg-const*[*iff*]:
$\mathit{nneg}\ (\lambda x.\ c) \longleftrightarrow 0 \le c$
⟨*proof*⟩

lemma *nneg-o*[*intro,simp*]:
$\mathit{nneg}\ P \Longrightarrow \mathit{nneg}\ (P\ o\ f)$
⟨*proof*⟩

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

$[\![\, \mathit{bounded}\ P;\ \mathit{nneg}\ P\,]\!] \Longrightarrow 0 \leq \mathit{bound\text{-}of}\ P$
$\langle \mathit{proof} \rangle$

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

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

3.1.3 Sound Expectations

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

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

lemma *soundI*:
$[\![\, \mathit{bounded}\ P;\ \mathit{nneg}\ P\,]\!] \Longrightarrow \mathit{sound}\ P$
$\langle \mathit{proof} \rangle$

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

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

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

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

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*: *sound P* **and** *sQ*: *sound Q*
shows $\mathit{sound}\ (\lambda s.\ P\ s + Q\ s)$
$\langle \mathit{proof} \rangle$

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*:
  $0 \leq c \Longrightarrow$ *sound* $(\lambda s.\ c)$
  $\langle proof \rangle$

lemma *sound-o*[*intro*,*simp*]:
  sound P $\Longrightarrow$ *sound* $(P\ o\ f)$
  $\langle proof \rangle$

lemma *sc-bounded-by*[*intro*,*simp*]:
  $[\![$ *sound P*; $0 \leq c$ $]\!] \Longrightarrow$ *bounded-by* $(c *$ *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 $c *$ *bound-of* $P =$ *bound-of* $(\lambda x.\ c * P\ x)$
$\langle proof \rangle$

lemma *sc-sound*:
  $[\![$ *sound P*; $0 \leq c$ $]\!] \Longrightarrow$ *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 $P{::}'s \Rightarrow real$ **and** Q
assumes *bP*: *bounded-by a P*
  and *bQ*: *bounded-by b Q*
shows *bounded-by* $(a + b)$ $(\lambda s.\ P\ s + Q\ s)$
$\langle proof \rangle$

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

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* $(\lambda s.\ P\ s * Q\ s)$
$\langle proof \rangle$

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

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 expect* ⇒ *bool*
where *unitary P* $\longleftrightarrow$ *sound P* $\wedge$ *bounded-by 1 P*

lemma *unitaryI*[*intro*]:
$[\![$ *sound P*; *bounded-by 1 P* $]\!] \Longrightarrow$ *unitary P*
$\langle proof \rangle$

lemma *unitaryI2*:
$[\![$ *nneg P*; *bounded-by 1 P* $]\!] \Longrightarrow$ *unitary P*
$\langle proof \rangle$

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

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

3.1.5 Standard Expectations

definition
embed-bool :: $('s \Rightarrow bool) \Rightarrow {}'s \Rightarrow real$ (‹« - »› *1000*)
where

$«P» \equiv (\lambda s.\ \mathit{if}\ P\ s\ \mathit{then}\ 1\ \mathit{else}\ 0)$

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

lemma *embed-bool-nneg*[*simp*,*intro*]:
$\mathit{nneg}\ «P»$
⟨*proof*⟩

lemma *embed-bool-bounded-by-1*[*simp*,*intro*]:
$\mathit{bounded\text{-}by}\ 1\ «P»$
⟨*proof*⟩

lemma *embed-bool-bounded*[*simp*,*intro*]:
$\mathit{bounded}\ «P»$
⟨*proof*⟩

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

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

lemma *eval-embed-true*[*simp*]:
$P\ s \Longrightarrow «P»\ s = 1$
⟨*proof*⟩

lemma *eval-embed-false*[*simp*]:
$\neg P\ s \Longrightarrow «P»\ s = 0$
⟨*proof*⟩

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

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

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

lemma *expect-1-I*:
$P\ x \Longrightarrow 1 \leq «P»\ x$
⟨*proof*⟩

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

sound «*P*»
⟨*proof*⟩

lemma *embed-o*[*simp*]:
$«P» \; o \; f = «P \; o \; f»$
⟨*proof*⟩

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

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

lemma *negateI*:
$\neg\ P\ s \Longrightarrow \mathcal{N}\ P\ s$
⟨*proof*⟩

lemma *embed-split*:
$f\,s = «P»\ s * f\,s + «\mathcal{N}\ P»\ s * f\,s$
⟨*proof*⟩

lemma *negate-embed*:
$«\mathcal{N}\ P»\ s = 1 - «P»\ s$
⟨*proof*⟩

lemma *eval-nembed-true*[*simp*]:
$P\ s \Longrightarrow «\mathcal{N}\ P»\ s = 0$
⟨*proof*⟩

lemma *eval-nembed-false*[*simp*]:
$\neg P\ s \Longrightarrow «\mathcal{N}\ P»\ s = 1$
⟨*proof*⟩

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

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

lemma *embed-bool-cancel*:
$«G»\ s * «\mathcal{N}\ G»\ s = 0$
⟨*proof*⟩

3.1.6 Entailment

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

abbreviation *entails* :: $('s \Rightarrow real) \Rightarrow ('s \Rightarrow real) \Rightarrow bool$ (‹- ⊩ -› *50*)

where $P \Vdash Q \equiv P \le Q$

lemma *entailsI*[*intro*]:
$[\![\bigwedge s.\ P\ s \le Q\ s]\!] \Longrightarrow P \Vdash Q$
⟨*proof*⟩

lemma *entailsD*[*dest*]:
$P \Vdash Q \Longrightarrow P\ s \le Q\ s$
⟨*proof*⟩

lemma *eq-entails*[*intro*]:
$P = Q \Longrightarrow P \Vdash Q$
⟨*proof*⟩

lemma *entails-trans*[*trans*]:
$[\![\ P \Vdash Q;\ Q \Vdash R\ ]\!] \Longrightarrow P \Vdash R$
⟨*proof*⟩

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

lemma *implies-entails*:
$[\![\ \bigwedge s.\ P\ s \Longrightarrow Q\ s\ ]\!] \Longrightarrow \text{«}P\text{»} \Vdash \text{«}Q\text{»}$
⟨*proof*⟩

lemma *entails-implies*:
$\bigwedge s.\ [\![\ \text{«}P\text{»} \Vdash \text{«}Q\text{»};\ P\ s\ ]\!] \Longrightarrow Q\ s$
⟨*proof*⟩

3.1.7 Expectation Conjunction

definition
$pconj :: real \Rightarrow real \Rightarrow\ real$ (**infixl** ‹.&› *71*)
where
$p\ .\&\ q \equiv p + q \ominus 1$

definition
$exp\text{-}conj :: ('s \Rightarrow real) \Rightarrow ('s \Rightarrow real) \Rightarrow ('s \Rightarrow real)$ (**infixl** ‹&&› *71*)
where $a\ \&\&\ b \equiv \lambda s.\ (a\ s\ .\&\ b\ s)$

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

lemma *pconj-lzero*[*intro*,*simp*]:
$b \le 1 \Longrightarrow 0\ .\&\ b = 0$
⟨*proof*⟩

lemma *pconj-rzero*[*intro*,*simp*]:
$b \le 1 \Longrightarrow b\ .\&\ 0 = 0$
⟨*proof*⟩

lemma *pconj-lone*[*intro*,*simp*]:
$0 \leq b \Longrightarrow 1 \mathbin{.\&} b = b$
$\langle proof \rangle$

lemma *pconj-rone*[*intro*,*simp*]:
$0 \leq b \Longrightarrow b \mathbin{.\&} 1 = b$
$\langle proof \rangle$

lemma *pconj-bconj*:
$«a»\ s \mathbin{.\&} «b»\ s = «\lambda s.\ a\ s \wedge b\ s»\ s$
$\langle proof \rangle$

lemma *pconj-comm*[*ac-simps*]:
$a \mathbin{.\&} b = b \mathbin{.\&} a$
$\langle proof \rangle$

lemma *pconj-assoc*:
$[\![\, 0 \leq a;\ a \leq 1;\ 0 \leq b;\ b \leq 1;\ 0 \leq c;\ c \leq 1 \,]\!] \Longrightarrow$
$a \mathbin{.\&} (b \mathbin{.\&} c) = (a \mathbin{.\&} b) \mathbin{.\&} c$
$\langle proof \rangle$

lemma *pconj-mono*:
$[\![\, a \leq b;\ c \leq d \,]\!] \Longrightarrow a \mathbin{.\&} c \leq b \mathbin{.\&} d$
$\langle proof \rangle$

lemma *pconj-nneg*[*intro*,*simp*]:
$0 \leq a \mathbin{.\&} b$
$\langle proof \rangle$

lemma *min-pconj*:
$(min\ a\ b) \mathbin{.\&} (min\ c\ d) \leq min\ (a \mathbin{.\&} c)\ (b \mathbin{.\&} d)$
$\langle proof \rangle$

lemma *pconj-less-one*[*simp*]:
$a + b < 1 \Longrightarrow a \mathbin{.\&} b = 0$
$\langle proof \rangle$

lemma *pconj-ge-one*[*simp*]:
$1 \leq a + b \Longrightarrow a \mathbin{.\&} b = a + b - 1$
$\langle proof \rangle$

lemma *pconj-idem*[*simp*]:
$«P»\ s \mathbin{.\&} «P»\ s = «P»\ s$
$\langle proof \rangle$

3.1.8 Rules Involving Conjunction.

lemma *exp-conj-mono-left*:

$P \Vdash Q \Longrightarrow P \;\&\&\; R \Vdash Q \;\&\&\; R$
$\langle proof \rangle$

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

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*]:
$(P \;\&\&\; Q)\; o\, f = (P\; o\, f) \;\&\&\; (Q\; o\, f)$
$\langle proof \rangle$

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*]:
sound P $\Longrightarrow$ «λ-. *True*» && $P = P$
$\langle proof \rangle$

lemma *exp-conj-top-right*[*simp*]:
sound P $\Longrightarrow P$ && «λ-. *True*» $= P$
$\langle proof \rangle$

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

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

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

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

$\textit{bounded-by } 1\ P \Longrightarrow P\ \&\&\ (\lambda s.\ 0) = (\lambda s.\ 0)$
⟨*proof*⟩

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

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

3.1.9 Rules Involving Entailment and Conjunction Together

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

lemma *entails-frame*:
assumes *ePR*: $P \Vdash R$
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*: $\bigwedge s.\ \exists x.\ P\ (x\ s)\ s = 1$
and *framed*: $\bigwedge x.\ P\ x\ \&\&\ R \Vdash Q\ x\ \&\&\ S$
and *sR*: *sound R* **and** *sS*: *sound S*
and *bQ*: $\bigwedge x.\ \textit{bounded-by } 1\ (Q\ x)$
shows $R \Vdash S$
⟨*proof*⟩

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

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

lemma *unitary-embed*[*iff*]:
unitary «*P*»
⟨*proof*⟩

lemma *unitary-const*[*iff*]:
$[\![\ 0 \le c;\ c \le 1\ ]\!] \Longrightarrow \textit{unitary}\ (\lambda s.\ c)$
⟨*proof*⟩

lemma *unitary-mult*:
  assumes *uA*: *unitary A* **and** *uB*: *unitary B*
  shows *unitary* $(\lambda s.\ A\ s * B\ s)$
⟨*proof*⟩

lemma *exp-conj-unitary*:
  ⟦ *unitary P*; *unitary Q* ⟧ $\Longrightarrow$ *unitary* (*P* && *Q*)
  ⟨*proof*⟩

lemma *unitary-comp*[*simp*]:
  unitary P $\Longrightarrow$ *unitary* (*P o f*)
  ⟨*proof*⟩

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

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

end

3.2 Expectation Transformers

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

Transformers are functions from expectations to expectations i.e. (*'s* ⇒ *real*) ⇒ *'s* ⇒ *real*.

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

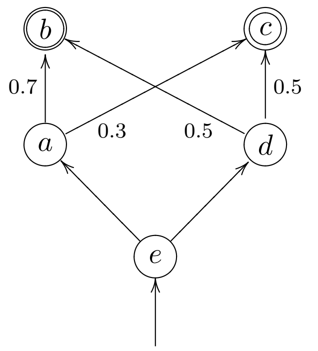

Figure 3.2: A nondeterministic-probabilistic automaton.

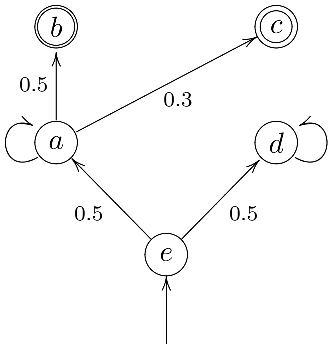

Figure 3.3: A diverging automaton.

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 probabilty of terminating from the initial state (e) is no higher than 0.5. If it instead takes the edge to state a, we again see a self loop, and thus in theory an infinite trace. In this case, however, every time the automaton reaches state a, with probability $0.5 + 0.3 = 0.8$, it transitions to a terminating state. An infinite trace of transitions $a \rightarrow a \rightarrow \ldots$ thus has probability 0, and the automaton terminates with probability 1. We formalise such probabilistic termination 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 ($\bot = (\lambda s. 0)$, $\top = (\lambda s. b)$), and is closed under the action of feasible transformers, including $\sqcap$ and $\sqcup$, which are themselves feasible. We are thus able to define both least and greatest fixed points on this set, and thus give semantics to recursive programs built from feasible components.

3.2.1 Comparing Transformers

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

definition
  le-trans :: *'s trans* ⇒ *'s trans* ⇒ *bool*
where
  le-trans t u ≡ ∀ *P*. *sound P* ⟶ *t P* ≤ *u P*

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

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

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

lemma *le-transI*[*intro*]:
$[\![\ \bigwedge P.\ \textit{sound}\ P \Longrightarrow t\ P \le u\ P\ ]\!] \Longrightarrow \textit{le-trans}\ t\ u$
⟨*proof*⟩

lemma *le-utransI*[*intro*]:
$[\![\ \bigwedge P.\ \textit{unitary}\ P \Longrightarrow t\ P \le u\ P\ ]\!] \Longrightarrow \textit{le-utrans}\ t\ u$
⟨*proof*⟩

lemma *le-transD*[*dest*]:
$[\![\ \textit{le-trans}\ t\ u;\ \textit{sound}\ P\ ]\!] \Longrightarrow t\ P \le u\ P$
⟨*proof*⟩

lemma *le-utransD*[*dest*]:
$[\![\ \textit{le-utrans}\ t\ u;\ \textit{unitary}\ P\ ]\!] \Longrightarrow t\ P \le u\ P$
⟨*proof*⟩

lemma *le-trans-trans*[*trans*]:
$[\![\ \textit{le-trans}\ x\ y;\ \textit{le-trans}\ y\ z\ ]\!] \Longrightarrow \textit{le-trans}\ x\ z$
⟨*proof*⟩

lemma *le-utrans-trans*[*trans*]:
$[\![\ \textit{le-utrans}\ x\ y;\ \textit{le-utrans}\ y\ z\ ]\!] \Longrightarrow \textit{le-utrans}\ x\ z$
⟨*proof*⟩

lemma *le-trans-refl*[*iff*]:
$\textit{le-trans}\ x\ x$
⟨*proof*⟩

lemma *le-utrans-refl*[*iff*]:
$\textit{le-utrans}\ x\ x$
⟨*proof*⟩

lemma *le-trans-le-utrans*[*dest*]:
$\textit{le-trans}\ t\ u \Longrightarrow \textit{le-utrans}\ t\ u$
⟨*proof*⟩

definition
$\textit{l-trans} :: \textit{'s trans} \Rightarrow \textit{'s trans} \Rightarrow \textit{bool}$
where
$\textit{l-trans}\ t\ u \longleftrightarrow \textit{le-trans}\ t\ u \wedge \neg\ \textit{le-trans}\ u\ t$

Transformer equivalence is induced by comparison:

definition
$\textit{equiv-trans} :: \textit{'s trans} \Rightarrow \textit{'s trans} \Rightarrow \textit{bool}$
where
$\textit{equiv-trans}\ t\ u \longleftrightarrow \textit{le-trans}\ t\ u \wedge \textit{le-trans}\ u\ t$

definition
$\mathit{equiv\text{-}utrans} :: \mathit{'s\ trans} \Rightarrow \mathit{'s\ trans} \Rightarrow \mathit{bool}$
where
$\mathit{equiv\text{-}utrans}\ t\ u \longleftrightarrow \mathit{le\text{-}utrans}\ t\ u \wedge \mathit{le\text{-}utrans}\ u\ t$

lemma *equiv-transI*[*intro*]:
$\llbracket \bigwedge P.\ \mathit{sound}\ P \Longrightarrow t\ P = u\ P \rrbracket \Longrightarrow \mathit{equiv\text{-}trans}\ t\ u$
$\langle \mathit{proof} \rangle$

lemma *equiv-utransI*[*intro*]:
$\llbracket \bigwedge P.\ \mathit{sound}\ P \Longrightarrow t\ P = u\ P \rrbracket \Longrightarrow \mathit{equiv\text{-}utrans}\ t\ u$
$\langle \mathit{proof} \rangle$

lemma *equiv-transD*[*dest*]:
$\llbracket \mathit{equiv\text{-}trans}\ t\ u;\ \mathit{sound}\ P \rrbracket \Longrightarrow t\ P = u\ P$
$\langle \mathit{proof} \rangle$

lemma *equiv-utransD*[*dest*]:
$\llbracket \mathit{equiv\text{-}utrans}\ t\ u;\ \mathit{unitary}\ P \rrbracket \Longrightarrow t\ P = u\ P$
$\langle \mathit{proof} \rangle$

lemma *equiv-trans-refl*[*iff*]:
$\mathit{equiv\text{-}trans}\ t\ t$
$\langle \mathit{proof} \rangle$

lemma *equiv-utrans-refl*[*iff*]:
$\mathit{equiv\text{-}utrans}\ t\ t$
$\langle \mathit{proof} \rangle$

lemma *le-trans-antisym*:
$\llbracket \mathit{le\text{-}trans}\ x\ y;\ \mathit{le\text{-}trans}\ y\ x \rrbracket \Longrightarrow \mathit{equiv\text{-}trans}\ x\ y$
$\langle \mathit{proof} \rangle$

lemma *le-utrans-antisym*:
$\llbracket \mathit{le\text{-}utrans}\ x\ y;\ \mathit{le\text{-}utrans}\ y\ x \rrbracket \Longrightarrow \mathit{equiv\text{-}utrans}\ x\ y$
$\langle \mathit{proof} \rangle$

lemma *equiv-trans-comm*[*ac-simps*]:
$\mathit{equiv\text{-}trans}\ t\ u \longleftrightarrow \mathit{equiv\text{-}trans}\ u\ t$
$\langle \mathit{proof} \rangle$

lemma *equiv-utrans-comm*[*ac-simps*]:
$\mathit{equiv\text{-}utrans}\ t\ u \longleftrightarrow \mathit{equiv\text{-}utrans}\ u\ t$
$\langle \mathit{proof} \rangle$

lemma *equiv-imp-le*[*intro*]:
$\mathit{equiv\text{-}trans}\ t\ u \Longrightarrow \mathit{le\text{-}trans}\ t\ u$
$\langle \mathit{proof} \rangle$

lemma *equivu-imp-le*[*intro*]:
equiv-utrans t u ⟹ *le-utrans t u*
⟨*proof*⟩

lemma *equiv-imp-le-alt*:
equiv-trans t u ⟹ *le-trans u t*
⟨*proof*⟩

lemma *equiv-uimp-le-alt*:
equiv-utrans t u ⟹ *le-utrans u t*
⟨*proof*⟩

lemma *le-trans-equiv-rsp*[*simp*]:
equiv-trans t u ⟹ *le-trans t v* ⟷ *le-trans u v*
⟨*proof*⟩

lemma *le-utrans-equiv-rsp*[*simp*]:
equiv-utrans t u ⟹ *le-utrans t v* ⟷ *le-utrans u v*
⟨*proof*⟩

lemma *equiv-trans-le-trans*[*trans*]:
⟦ *equiv-trans t u*; *le-trans u v* ⟧ ⟹ *le-trans t v*
⟨*proof*⟩

lemma *equiv-utrans-le-utrans*[*trans*]:
⟦ *equiv-utrans t u*; *le-utrans u v* ⟧ ⟹ *le-utrans t v*
⟨*proof*⟩

lemma *le-trans-equiv-rsp-right*[*simp*]:
equiv-trans t u ⟹ *le-trans v t* ⟷ *le-trans v u*
⟨*proof*⟩

lemma *le-utrans-equiv-rsp-right*[*simp*]:
equiv-utrans t u ⟹ *le-utrans v t* ⟷ *le-utrans v u*
⟨*proof*⟩

lemma *le-trans-equiv-trans*[*trans*]:
⟦ *le-trans t u*; *equiv-trans u v* ⟧ ⟹ *le-trans t v*
⟨*proof*⟩

lemma *le-utrans-equiv-utrans*[*trans*]:
⟦ *le-utrans t u*; *equiv-utrans u v* ⟧ ⟹ *le-utrans t v*
⟨*proof*⟩

lemma *equiv-trans-trans*[*trans*]:
assumes *xy*: *equiv-trans x y*
and *yz*: *equiv-trans y z*
shows *equiv-trans x z*

$\langle proof \rangle$

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 $\Longrightarrow$ *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* $\longleftrightarrow (\forall P\ b.\ \textit{bounded-by}\ b\ P \wedge nneg\ P \longrightarrow$
$\textit{bounded-by}\ b\ (t\ P) \wedge nneg\ (t\ P))$

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*]:
$[\![\bigwedge b\ P.\ [\![\textit{bounded-by}\ b\ P; nneg\ P]\!] \Longrightarrow \textit{bounded-by}\ b\ (t\ P);$
$\bigwedge b\ P.\ [\![\textit{bounded-by}\ b\ P; nneg\ P]\!] \Longrightarrow nneg\ (t\ P)]\!] \Longrightarrow feasible\ t$
$\langle proof \rangle$

lemma *feasible-boundedD*[*dest*]:
$[\![feasible\ t; \textit{bounded-by}\ b\ P; nneg\ P]\!] \Longrightarrow \textit{bounded-by}\ b\ (t\ P)$
$\langle proof \rangle$

lemma *feasible-nnegD*[*dest*]:
$[\![feasible\ t; \textit{bounded-by}\ b\ P; nneg\ P]\!] \Longrightarrow nneg\ (t\ P)$
$\langle proof \rangle$

lemma *feasible-sound*[*dest*]:
$[\![feasible\ t; sound\ P]\!] \Longrightarrow sound\ (t\ P)$
$\langle proof \rangle$

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*:

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

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

lemma *feasible-fixes-top*:
$\mathit{feasible}\ t \Longrightarrow t\ (\lambda s.\ \mathit{1}) \le (\lambda s.\ (\mathit{1}{::}\mathit{real}))$
$\langle \mathit{proof} \rangle$

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

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

Monotonicity

definition
$\mathit{mono\text{-}trans} :: (('s \Rightarrow \mathit{real}) \Rightarrow ('s \Rightarrow \mathit{real})) \Rightarrow \mathit{bool}$
where
$\mathit{mono\text{-}trans}\ t \equiv \forall P\ Q.\ (\mathit{sound}\ P \wedge \mathit{sound}\ Q \wedge P \le Q) \longrightarrow t\ P \le t\ Q$

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: $[\![\mathit{mono\text{-}trans}\ t;\ P \Vdash u\ Q;\ Q \Vdash t\ R]\!] \Longrightarrow P \Vdash u\ (t\ R)$.

lemma *mono-transI*[*intro*]:
$[\![\ \bigwedge P\ Q.\ [\![\ \mathit{sound}\ P;\ \mathit{sound}\ Q;\ P \le Q\ ]\!] \Longrightarrow\ t\ P \le t\ Q\ ]\!] \Longrightarrow \mathit{mono\text{-}trans}\ t$
$\langle \mathit{proof} \rangle$

lemma *mono-transD*[*dest*]:
$[\![\ \mathit{mono\text{-}trans}\ t;\ \mathit{sound}\ P;\ \mathit{sound}\ Q;\ P \le Q\ ]\!] \Longrightarrow t\ P \le t\ Q$
$\langle \mathit{proof} \rangle$

Scaling

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

definition
$\mathit{scaling} :: (('s \Rightarrow \mathit{real}) \Rightarrow ('s \Rightarrow \mathit{real})) \Rightarrow \mathit{bool}$
where
$\mathit{scaling}\ t \equiv \forall P\ c\ x.\ \mathit{sound}\ P \wedge 0 \leq c \longrightarrow c * t\ P\ x = t\ (\lambda x.\ c * P\ x)\ x$

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

lemma *scalingI*[*intro*]:
$[\![\bigwedge P\ c\ x.\ [\![\mathit{sound}\ P;\ 0 \leq c]\!] \Longrightarrow c * t\ P\ x = t\ (\lambda x.\ c * P\ x)\ x]\!] \Longrightarrow \mathit{scaling}\ t$
⟨*proof*⟩

lemma *scalingD*[*dest*]:
$[\![\mathit{scaling}\ t;\ \mathit{sound}\ P;\ 0 \leq c]\!] \Longrightarrow c * t\ P\ x = t\ (\lambda x.\ c * P\ x)\ x$
⟨*proof*⟩

lemma *right-scalingD*:
assumes *st*: $\mathit{scaling}\ t$
and *sP*: $\mathit{sound}\ P$
and *nnc*: $0 \leq c$
shows $t\ P\ s * c = t\ (\lambda s.\ P\ s * c)\ s$
⟨*proof*⟩

Healthiness

Healthy transformers are feasible and monotonic, and respect scaling

definition
$\mathit{healthy} :: (('s \Rightarrow \mathit{real}) \Rightarrow ('s \Rightarrow \mathit{real})) \Rightarrow \mathit{bool}$
where
$\mathit{healthy}\ t \longleftrightarrow \mathit{feasible}\ t \wedge \mathit{mono\text{-}trans}\ t \wedge \mathit{scaling}\ t$

lemma *healthyI*[*intro*]:
$[\![\mathit{feasible}\ t;\ \mathit{mono\text{-}trans}\ t;\ \mathit{scaling}\ t]\!] \Longrightarrow \mathit{healthy}\ t$
⟨*proof*⟩

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

lemma *healthy-monoD*[*dest*]:
$\mathit{healthy}\ t \Longrightarrow \mathit{mono\text{-}trans}\ t$
⟨*proof*⟩

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

lemma *healthy-feasibleD*[*dest*]:
healthy t ⟹ *feasible t*
⟨*proof*⟩

lemma *healthy-scalingD*[*dest*]:
healthy t ⟹ *scaling t*
⟨*proof*⟩

lemma *healthy-bounded-byD*[*intro*]:
⟦ *healthy t*; *bounded-by b P*; *nneg P* ⟧ ⟹ *bounded-by b* (*t P*)
⟨*proof*⟩

lemma *healthy-bounded-byD2*:
⟦ *healthy t*; *bounded-by b P*; *sound P* ⟧ ⟹ *bounded-by b* (*t P*)
⟨*proof*⟩

lemma *healthy-boundedD*[*dest*,*simp*]:
⟦ *healthy t*; *sound P* ⟧ ⟹ *bounded* (*t P*)
⟨*proof*⟩

lemma *healthy-nnegD*[*dest*,*simp*]:
⟦ *healthy t*; *sound P* ⟧ ⟹ *nneg* (*t P*)
⟨*proof*⟩

lemma *healthy-nnegD2*[*dest*,*simp*]:
⟦ *healthy t*; *bounded-by b P*; *nneg P* ⟧ ⟹ *nneg* (*t P*)
⟨*proof*⟩

lemma *healthy-sound*[*intro*]:
⟦ *healthy t*; *sound P* ⟧ ⟹ *sound* (*t P*)
⟨*proof*⟩

lemma *healthy-unitary*[*intro*]:
⟦ *healthy t*; *unitary P* ⟧ ⟹ *unitary* (*t P*)
⟨*proof*⟩

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

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

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

lemma *le-trans-bot*[*intro*,*simp*]:
healthy t ⟹ *le-trans* (λ*P s*. *0*) *t*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

lemma *nearly-healthy-monoD*[*dest*]:
$[\![$ *nearly-healthy t*; $P \Vdash Q$; *unitary P*; *unitary Q* $]\!] \Longrightarrow t\,P \Vdash t\,Q$
⟨*proof*⟩

lemma *nearly-healthy-unitaryD*[*dest*]:
$[\![$ *nearly-healthy t*; *unitary P* $]\!] \Longrightarrow$ *unitary* $(t\,P)$
⟨*proof*⟩

lemma *healthy-nearly-healthy*[*dest*]:
assumes *ht*: *healthy t*
shows *nearly-healthy t*
⟨*proof*⟩

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

lemma *sublinearI*[*intro*]:
$[\![\ \bigwedge a\ b\ c\ P\ Q\ s.\ [\![$ *sound P*; *sound Q*; $0 \leq a$; $0 \leq b$; $0 \leq c$ $]\!] \Longrightarrow$
$a * t\,P\,s + b * t\,Q\,s \ominus c \leq$
$t\,(\lambda s'.\ a * P\,s' + b * Q\,s' \ominus c)\ s\ ]\!] \Longrightarrow$ *sublinear t*
⟨*proof*⟩

lemma *sublinearD*[*dest*]:
$[\![$ *sublinear t*; *sound P*; *sound Q*; $0 \leq a$; $0 \leq b$; $0 \leq c$ $]\!] \Longrightarrow$
$a * t\,P\,s + b * t\,Q\,s \ominus c \leq$
$t\,(\lambda s'.\ a * P\,s' + b * Q\,s' \ominus c)\ s$
⟨*proof*⟩

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* ::

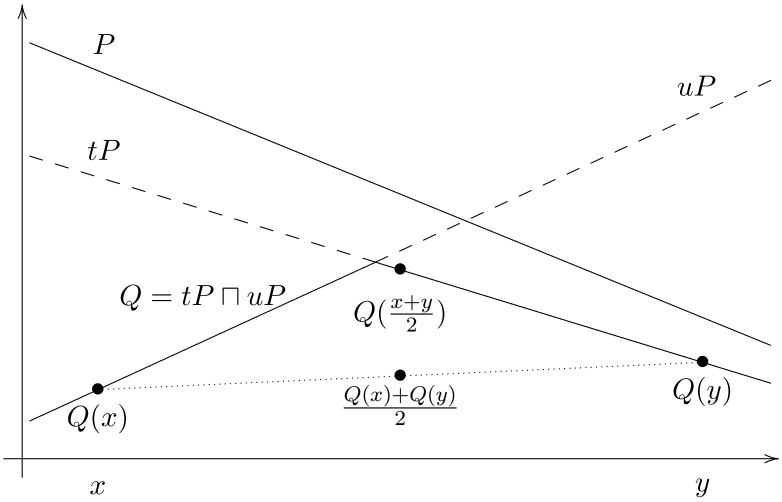

Figure 3.4: A graphical depiction of sub-additivity as convexity.

$(('s \Rightarrow real) \Rightarrow ('s \Rightarrow real)) \Rightarrow bool$
where
$sub\text{-}add\ t \longleftrightarrow (\forall\, P\ Q\ s.\ (sound\ P \wedge sound\ Q) \longrightarrow$
$t\ P\ s + t\ Q\ s \leq t\ (\lambda s'.\ P\ s' + Q\ s')\ s)$

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

Here P is an affine function (expectation) *real* $\Rightarrow$ *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:

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

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

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

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*]:
$\llbracket \bigwedge P\ Q\ s.\ \llbracket\ sound\ P;\ sound\ Q\ \rrbracket \Longrightarrow$
$\quad t\ P\ s + t\ Q\ s \leq t\ (\lambda s'.\ P\ s' + Q\ s')\ s\ \rrbracket \Longrightarrow sub\text{-}add\ t$
⟨*proof*⟩

lemma *sub-addI2*:
$\llbracket \bigwedge P\ Q.\ \llbracket\ sound\ P;\ sound\ Q\ \rrbracket \Longrightarrow$
$\quad \lambda s.\ t\ P\ s + t\ Q\ s \Vdash t\ (\lambda s.\ P\ s + Q\ s) \rrbracket \Longrightarrow$
sub-add t
⟨*proof*⟩

lemma *sub-addD*[*dest*]:
$\llbracket\ sub\text{-}add\ t;\ sound\ P;\ sound\ Q\ \rrbracket \Longrightarrow t\ P\ s + t\ Q\ s \leq t\ (\lambda s'.\ P\ s' + Q\ s')\ s$
⟨*proof*⟩

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

Sublinearity and feasibility imply sub-additivity.

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

A few properties following from sub-additivity:

lemma *standard-negate*:
assumes *ht*: *healthy t*
and *sat*: *sub-add t*

shows $t \text{ «}P\text{» } s + t \text{ «}\mathcal{N}\ P\text{» } s \leq 1$
⟨*proof*⟩

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

lemma *sub-add-guard-split*:
fixes t::*'s::finite trans* **and** P::*'s expect* **and** s::*'s*
assumes *sat*: *sub-add t*
and *ht*: *healthy t*
and *sP*: *sound P*
shows $(\sum y{\in}\{s.\ G\ s\}.\ \ P\ y * t \text{ « } \lambda z.\ z = y \text{ » } s) +$
$(\sum y{\in}\{s.\ \neg G\ s\}.\ P\ y * t \text{ « } \lambda z.\ z = y \text{ » } s) \leq t\ P\ s$
⟨*proof*⟩

Sub-distributivity

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

lemma *sub-distribI*[*intro*]:
$[\![\ \bigwedge P\ s.\ sound\ P \Longrightarrow t\ P\ s \ominus 1 \leq t\ (\lambda s'.\ P\ s' \ominus 1)\ s\ ]\!] \Longrightarrow sub\text{-}distrib\ t$
⟨*proof*⟩

lemma *sub-distribI2*:
$[\![\ \bigwedge P.\ sound\ P \Longrightarrow \lambda s.\ t\ P\ s \ominus 1 \Vdash t\ (\lambda s.\ P\ s \ominus 1)\ ]\!] \Longrightarrow sub\text{-}distrib\ t$
⟨*proof*⟩

lemma *sub-distribD*[*dest*]:
$[\![\ sub\text{-}distrib\ t;\ sound\ P\ ]\!] \Longrightarrow t\ P\ s \ominus 1 \leq t\ (\lambda s'.\ P\ s' \ominus 1)\ s$
⟨*proof*⟩

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*
⟨*proof*⟩

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
$\mathit{sub\text{-}conj}\ t \equiv \forall\, P\ Q.\ (\mathit{sound}\ P \wedge \mathit{sound}\ Q) \longrightarrow$
$t\ P\ \&\&\ t\ Q \Vdash t\ (P\ \&\&\ Q)$

lemma *sub-conjI*[*intro*]:
$[\![\bigwedge P\ Q.\ [\![\mathit{sound}\ P;\ \mathit{sound}\ Q]\!] \Longrightarrow$
$t\ P\ \&\&\ t\ Q \Vdash t\ (P\ \&\&\ Q)]\!] \Longrightarrow \mathit{sub\text{-}conj}\ t$
$\langle proof \rangle$

lemma *sub-conjD*[*dest*]:
$[\![\mathit{sub\text{-}conj}\ t;\ \mathit{sound}\ P;\ \mathit{sound}\ Q]\!] \Longrightarrow t\ P\ \&\&\ t\ Q \Vdash t\ (P\ \&\&\ Q)$
$\langle proof \rangle$

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

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*:
$[\![\mathit{equiv\text{-}trans}\ t\ u;\ \mathit{sublinear}\ t;\ \mathit{healthy}\ t]\!] \Longrightarrow \mathit{sublinear}\ u$
$\langle proof \rangle$

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
$\quad \mathit{additive} :: (('a \Rightarrow \mathit{real}) \Rightarrow 'a \Rightarrow \mathit{real}) \Rightarrow \mathit{bool}$
where
$\quad \mathit{additive}\ t \equiv \forall\, P\ Q.\ (\mathit{sound}\ P \wedge \mathit{sound}\ Q) \longrightarrow$
$\qquad\qquad t\ (\lambda s.\ P\ s + Q\ s) = (\lambda s.\ t\ P\ s + t\ Q\ s)$

lemma *additiveD*:
$\quad [\![\ \mathit{additive}\ t;\ \mathit{sound}\ P;\ \mathit{sound}\ Q\ ]\!] \Longrightarrow t\ (\lambda s.\ P\ s + Q\ s) = (\lambda s.\ t\ P\ s + t\ Q\ s)$
$\quad \langle \mathit{proof} \rangle$

lemma *additiveI*[*intro*]:
$\quad [\![\ \bigwedge P\ Q\ s.\ [\![\ \mathit{sound}\ P;\ \mathit{sound}\ Q\ ]\!] \Longrightarrow t\ (\lambda s.\ P\ s + Q\ s)\ s = t\ P\ s + t\ Q\ s\ ]\!] \Longrightarrow$
$\quad\ \mathit{additive}\ t$
$\quad \langle \mathit{proof} \rangle$

Additivity is strictly stronger than sub-additivity.

lemma *additive-sub-add*:
$\quad \mathit{additive}\ t \Longrightarrow \mathit{sub\text{-}add}\ t$
$\quad \langle \mathit{proof} \rangle$

The additivity property extends to finite summation.

lemma *additive-sum*:
 fixes $S{::}'s\ \mathit{set}$
 assumes *additive*: $\mathit{additive}\ t$
  and *healthy*: $\mathit{healthy}\ t$
  and *finite*: $\mathit{finite}\ S$
  and *sPz*: $\bigwedge z.\ \mathit{sound}\ (P\ z)$
 shows $t\ (\lambda x.\ \sum y{\in}S.\ P\ y\ x) = (\lambda x.\ \sum y{\in}S.\ t\ (P\ y)\ x)$
$\langle \mathit{proof} \rangle$

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 $t{::}('s{::}\mathit{finite} \Rightarrow \mathit{real}) \Rightarrow 's \Rightarrow \mathit{real}$
 assumes *additive*: $\mathit{additive}\ t$
  and *ht*: $\mathit{healthy}\ t$

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

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 $t\ P\ x = (\sum y{\in}\{s.\quad G\ s\}.\ P\ y * t\ «\lambda z.\ z = y»\ x) +$
$(\sum y{\in}\{s.\ \neg\ G\ s\}.\ P\ y * t\ «\lambda z.\ z = y»\ x)$
$\langle proof \rangle$

Maximality

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

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

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

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*]:
$[\![\ additive\ t;\ maximal\ t\ ]\!] \Longrightarrow determ\ t$
$\langle proof \rangle$

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 «P» $s + t$ «$\mathcal{N}$ P» $s = 1$
⟨*proof*⟩

3.2.5 Modular Reasoning

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

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

lemma *entails-combine*:
 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 $P\ \&\&\ Q \Vdash t\ (R\ \&\&\ S)$
⟨*proof*⟩

These allow mismatched results to be composed

lemma *entails-strengthen-post*:
 $[\![\ P \Vdash t\ Q;\ healthy\ t;\ sound\ R;\ Q \Vdash R;\ sound\ Q\ ]\!] \Longrightarrow P \Vdash t\ R$
 ⟨*proof*⟩

lemma *entails-weaken-pre*:
 $[\![\ Q \Vdash t\ R;\ P \Vdash Q\ ]\!] \Longrightarrow P \Vdash t\ R$
 ⟨*proof*⟩

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

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

3.2.6 Transforming Standard Expectations

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

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

lemma *use-premise*:

assumes h: *healthy t* **and** wP: $\bigwedge s.\ P\ s \Longrightarrow 1 \leq t\ «Q»\ s$
shows $«P» \Vdash t\ «Q»$
$\langle proof\rangle$

The other direction works too.

lemma *fold-premise*:
assumes ht: *healthy t*
and wp: $«P» \Vdash t\ «Q»$
shows $\forall\, s.\ P\ s \longrightarrow 1 \leq t\ «Q»\ s$
$\langle proof\rangle$

Predicate conjunction behaves as expected:

lemma *conj-post*:
$[\![\ P \Vdash t\ «\lambda s.\ Q\ s \wedge R\ s»;\ healthy\ t\ ]\!] \Longrightarrow P \Vdash t\ «Q»$
$\langle proof\rangle$

Similar to $[\![healthy\ ?t;\ \bigwedge s.\ ?P\ s \Longrightarrow 1 \leq\ ?t\ «\ ?Q\ »\ s]\!] \Longrightarrow «\ ?P\ » \Vdash\ ?t\ «\ ?Q\ »$, but more general.

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

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 λ-. *0*), 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 $\forall P.\ \mathit{sound}\ P \longrightarrow \mathit{healthy}\ (t\ P)$ and $\forall Q.\ \mathit{sound}\ Q \longrightarrow \mathit{healthy}\ (\lambda P.\ t\ P\ Q)$. Thus by feasibility, $t\ P\ Q$ must be bounded by *bound-of* P. Thus, as by definition $x \leq t\ P\ x$ for any fixed point, all must lie in the set of sound expectations bounded above by λ-. *bound-of* P.

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

lemma *Inf-exp-lower*:
$[\![\ P \in S;\ \forall P{\in}S.\ \mathit{nneg}\ P\ ]\!] \Longrightarrow \mathit{Inf\text{-}exp}\ S \leq P$
$\langle \mathit{proof} \rangle$

lemma *Inf-exp-greatest*:
$[\![\ S \neq \{\};\ \forall P{\in}S.\ Q \leq P\ ]\!] \Longrightarrow Q \leq \mathit{Inf\text{-}exp}\ S$
$\langle \mathit{proof} \rangle$

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

lemma *Sup-exp-upper*:
$[\![\ P \in S;\ \forall P{\in}S.\ \mathit{bounded\text{-}by}\ b\ P\ ]\!] \Longrightarrow P \leq \mathit{Sup\text{-}exp}\ S$
$\langle \mathit{proof} \rangle$

lemma *Sup-exp-least*:
$[\![\ \forall P{\in}S.\ P \leq Q;\ \mathit{nneg}\ Q\ ]\!] \Longrightarrow \mathit{Sup\text{-}exp}\ S \leq Q$
$\langle \mathit{proof} \rangle$

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

definition *lfp-exp* :: *'s trans* $\Rightarrow$ *'s expect*
where *lfp-exp* $t = \mathit{Inf\text{-}exp}\ \{P.\ \mathit{sound}\ P \wedge t\ P \leq P\}$

lemma *lfp-exp-lowerbound*:
$[\![\ t\ P \leq P;\ \mathit{sound}\ P\ ]\!] \Longrightarrow \mathit{lfp\text{-}exp}\ t \leq P$
$\langle \mathit{proof} \rangle$

lemma *lfp-exp-greatest*:
$[\![\ \bigwedge P.\ [\![\ t\ P \leq P;\ \mathit{sound}\ P\ ]\!] \Longrightarrow Q \leq P;\ \mathit{sound}\ Q;\ t\ R \Vdash R;\ \mathit{sound}\ R\ ]\!] \Longrightarrow Q \leq \mathit{lfp\text{-}exp}\ t$
$\langle \mathit{proof} \rangle$

lemma *feasible-lfp-exp-sound*:
feasible $t \Longrightarrow$ *sound* (*lfp-exp* t)
$\langle \mathit{proof} \rangle$

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

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

lemma lfp-exp-unitary:
  (⋀P. unitary P ⟹ unitary (t P)) ⟹ unitary (lfp-exp t)
⟨proof⟩

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

lemma lfp-exp-lemma3:
  assumes st: ⋀P. sound P ⟹ sound (t P)
    and mt: mono-trans t
    and fR: t R ⊩ R and sR: sound R
  shows lfp-exp t ≤ t (lfp-exp t)
  ⟨proof⟩

lemma lfp-exp-unfold:
  assumes nt: ⋀P. sound P ⟹ sound (t P)
    and mt: mono-trans t
    and fR: t R ⊩ R and sR: sound R
  shows lfp-exp t = t (lfp-exp t)
  ⟨proof⟩

definition gfp-exp :: 's trans ⇒ 's expect
where gfp-exp t = Sup-exp {P. unitary P ∧ P ≤ t P}

lemma gfp-exp-upperbound:
  ⟦ P ≤ t P; unitary P ⟧ ⟹ P ≤ gfp-exp t
  ⟨proof⟩

lemma gfp-exp-least:
  ⟦ ⋀P. ⟦ P ≤ t P; unitary P ⟧ ⟹ P ≤ Q; unitary Q ⟧ ⟹ gfp-exp t ≤ Q
  ⟨proof⟩

lemma gfp-exp-bound:
  (⋀P. unitary P ⟹ unitary (t P)) ⟹ bounded-by 1 (gfp-exp t)
  ⟨proof⟩
```

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

lemma *gfp-exp-unitary*:
($\bigwedge P$. *unitary P* $\Longrightarrow$ *unitary* (*t P*)) $\Longrightarrow$ *unitary* (*gfp-exp t*)
⟨*proof*⟩

lemma *gfp-exp-lemma2*:
assumes *ft*: $\bigwedge P$. *unitary P* $\Longrightarrow$ *unitary* (*t P*)
and *mt*: $\bigwedge P\ Q$. $[\![$ *unitary P*; *unitary Q*; $P \Vdash Q$ $]\!] \Longrightarrow t\ P \Vdash t\ Q$
shows *gfp-exp t* $\leq$ *t* (*gfp-exp t*)
⟨*proof*⟩

lemma *gfp-exp-lemma3*:
assumes *ft*: $\bigwedge P$. *unitary P* $\Longrightarrow$ *unitary* (*t P*)
and *mt*: $\bigwedge P\ Q$. $[\![$ *unitary P*; *unitary Q*; $P \Vdash Q$ $]\!] \Longrightarrow t\ P \Vdash t\ Q$
shows *t* (*gfp-exp t*) $\leq$ *gfp-exp t*
⟨*proof*⟩

lemma *gfp-exp-unfold*:
($\bigwedge P$. *unitary P* $\Longrightarrow$ *unitary* (*t P*)) $\Longrightarrow$ ($\bigwedge P\ Q$. $[\![$ *unitary P*; *unitary Q*; $P \Vdash Q$ $]\!] \Longrightarrow t\ P \Vdash$
$t\ Q) \Longrightarrow$
gfp-exp t = *t* (*gfp-exp t*)
⟨*proof*⟩

3.3.2 The Lattice of Transformers

In addition to fixed points on expectations, we also need to reason about fixed points on expectation transformers. The interpretation of a recursive program in pGCL is as a fixed point of a function from transformers to transformers. In contrast to the case of expectations, *healthy* transformers do form a complete lattice, where the bottom element is λ- -. *0*, and the top element is the greatest allowed by feasibility: λP -. *bound-of P*.

definition *Inf-trans* :: *'s trans set* $\Rightarrow$ *'s trans*
where *Inf-trans S* = (λP. *Inf-exp* $\{t\ P\ |t.\ t \in S\}$)

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

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

definition *Sup-trans* :: *'s trans set* $\Rightarrow$ *'s trans*
where *Sup-trans S* = (λP. *Sup-exp* $\{t\ P\ |t.\ t \in S\}$)

lemma *Sup-trans-upper*:
$[\![\ t \in S;\ \forall\, u{\in}S.\ \forall\, P.\ \textit{unitary}\ P \longrightarrow \textit{unitary}\ (u\ P)\ ]\!] \Longrightarrow \textit{le-utrans}\ t\ (\textit{Sup-trans}\ S)$
$\langle \textit{proof} \rangle$

lemma *Sup-trans-upper2*:
$[\![\ t \in S;\ \forall\, u{\in}S.\ \forall\, P.\ (\textit{nneg}\ P \wedge \textit{bounded-by}\ b\ P) \longrightarrow (\textit{nneg}\ (u\ P) \wedge \textit{bounded-by}\ b\ (u\ P));$
$\textit{nneg}\ P;\ \textit{bounded-by}\ b\ P\ ]\!] \Longrightarrow t\ P \Vdash \textit{Sup-trans}\ S\ P$
$\langle \textit{proof} \rangle$

lemma *Sup-trans-least*:
$[\![\ \forall\, t{\in}S.\ \textit{le-utrans}\ t\ u;\ \bigwedge P.\ \textit{unitary}\ P \Longrightarrow \textit{unitary}\ (u\ P)\ ]\!] \Longrightarrow \textit{le-utrans}\ (\textit{Sup-trans}\ S)\ u$
$\langle \textit{proof} \rangle$

lemma *Sup-trans-least2*:
$[\![\ \forall\, t{\in}S.\ \forall\, P.\ \textit{nneg}\ P \longrightarrow \textit{bounded-by}\ b\ P \longrightarrow t\ P \Vdash u\ P;$
$\forall\, u{\in}S.\ \forall\, P.\ (\textit{nneg}\ P \wedge \textit{bounded-by}\ b\ P) \longrightarrow (\textit{nneg}\ (u\ P) \wedge \textit{bounded-by}\ b\ (u\ P));$
$\textit{nneg}\ P;\ \textit{bounded-by}\ b\ P;\ \bigwedge P.\ [\![\ \textit{nneg}\ P;\ \textit{bounded-by}\ b\ P\ ]\!] \Longrightarrow \textit{nneg}\ (u\ P)\ ]\!] \Longrightarrow$
$\textit{Sup-trans}\ S\ P \Vdash u\ P$
$\langle \textit{proof} \rangle$

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

definition $\textit{lfp-trans} :: ('s\ \textit{trans} \Rightarrow 's\ \textit{trans}) \Rightarrow 's\ \textit{trans}$
where $\textit{lfp-trans}\ T = \textit{Inf-trans}\ \{t.\ (\forall\, P.\ \textit{sound}\ P \longrightarrow \textit{sound}\ (t\ P)) \wedge \textit{le-trans}\ (T\ t)\ t\}$

lemma *lfp-trans-lowerbound*:
$[\![\ \textit{le-trans}\ (T\ t)\ t;\ \bigwedge P.\ \textit{sound}\ P \Longrightarrow \textit{sound}\ (t\ P)\ ]\!] \Longrightarrow \textit{le-trans}\ (\textit{lfp-trans}\ T)\ t$
$\langle \textit{proof} \rangle$

lemma *lfp-trans-greatest*:
$[\![\ \bigwedge t\ P.\ [\![\ \textit{le-trans}\ (T\ t)\ t;\ \bigwedge P.\ \textit{sound}\ P \Longrightarrow \textit{sound}\ (t\ P)\ ]\!] \Longrightarrow \textit{le-trans}\ u\ t;$
$\bigwedge P.\ \textit{sound}\ P \Longrightarrow \textit{sound}\ (v\ P);\ \textit{le-trans}\ (T\ v)\ v\ ]\!] \Longrightarrow$
$\textit{le-trans}\ u\ (\textit{lfp-trans}\ T)$
$\langle \textit{proof} \rangle$

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

lemma *lfp-trans-unitary*:

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

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

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

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

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

lemma *gfp-trans-upperbound*:
$[\![\ \textit{le-utrans}\ t\ (T\ t);\ \bigwedge P.\ \textit{unitary}\ P \Longrightarrow \textit{unitary}\ (t\ P)\ ]\!] \Longrightarrow \textit{le-utrans}\ t\ (\textit{gfp-trans}\ T)$
$\langle \textit{proof} \rangle$

lemma *gfp-trans-least*:
$[\![\ \bigwedge t.\ [\![\ \textit{le-utrans}\ t\ (T\ t);\ \bigwedge P.\ \textit{unitary}\ P \Longrightarrow \textit{unitary}\ (t\ P)\ ]\!] \Longrightarrow \textit{le-utrans}\ t\ u;$
$\bigwedge P.\ \textit{unitary}\ P \Longrightarrow \textit{unitary}\ (u\ P)\ ]\!] \Longrightarrow$
$\textit{le-utrans}\ (\textit{gfp-trans}\ T)\ u$

⟨*proof*⟩

lemma *gfp-trans-unitary*:
fixes $P :: {'s}\ \mathit{expect}$
assumes *uP*: $\mathit{unitary}\ P$
shows $\mathit{unitary}\ (\mathit{gfp\text{-}trans}\ T\ P)$
⟨*proof*⟩

lemma *gfp-trans-lemma2*:
assumes *mono*: $\bigwedge t\ u.\ [\![\ \mathit{le\text{-}utrans}\ t\ u;\ \bigwedge P.\ \mathit{unitary}\ P \Longrightarrow \mathit{unitary}\ (t\ P);$
$\bigwedge P.\ \mathit{unitary}\ P \Longrightarrow \mathit{unitary}\ (u\ P)\ ]\!] \Longrightarrow \mathit{le\text{-}utrans}\ (T\ t)\ (T\ u)$
and *hT*: $\bigwedge t\ P.\ [\![\ \bigwedge Q.\ \mathit{unitary}\ Q \Longrightarrow \mathit{unitary}\ (t\ Q);\ \mathit{unitary}\ P\ ]\!] \Longrightarrow \mathit{unitary}\ (T\ t\ P)$
shows $\mathit{le\text{-}utrans}\ (\mathit{gfp\text{-}trans}\ T)\ (T\ (\mathit{gfp\text{-}trans}\ T))$
⟨*proof*⟩

lemma *gfp-trans-lemma3*:
assumes *mono*: $\bigwedge t\ u.\ [\![\ \mathit{le\text{-}utrans}\ t\ u;\ \bigwedge P.\ \mathit{unitary}\ P \Longrightarrow \mathit{unitary}\ (t\ P);$
$\bigwedge P.\ \mathit{unitary}\ P \Longrightarrow \mathit{unitary}\ (u\ P)\ ]\!] \Longrightarrow \mathit{le\text{-}utrans}\ (T\ t)\ (T\ u)$
and *hT*: $\bigwedge t\ P.\ [\![\ \bigwedge Q.\ \mathit{unitary}\ Q \Longrightarrow \mathit{unitary}\ (t\ Q);\ \mathit{unitary}\ P\ ]\!] \Longrightarrow \mathit{unitary}\ (T\ t\ P)$
shows $\mathit{le\text{-}utrans}\ (T\ (\mathit{gfp\text{-}trans}\ T))\ (\mathit{gfp\text{-}trans}\ T)$
⟨*proof*⟩

lemma *gfp-trans-unfold*:
assumes *mono*: $\bigwedge t\ u.\ [\![\ \mathit{le\text{-}utrans}\ t\ u;\ \bigwedge P.\ \mathit{unitary}\ P \Longrightarrow \mathit{unitary}\ (t\ P);$
$\bigwedge P.\ \mathit{unitary}\ P \Longrightarrow \mathit{unitary}\ (u\ P)\ ]\!] \Longrightarrow \mathit{le\text{-}utrans}\ (T\ t)\ (T\ u)$
and *hT*: $\bigwedge t\ P.\ [\![\ \bigwedge Q.\ \mathit{unitary}\ Q \Longrightarrow \mathit{unitary}\ (t\ Q);\ \mathit{unitary}\ P\ ]\!] \Longrightarrow \mathit{unitary}\ (T\ t\ P)$
shows $\mathit{equiv\text{-}utrans}\ (\mathit{gfp\text{-}trans}\ T)\ (T\ (\mathit{gfp\text{-}trans}\ T))$
⟨*proof*⟩

3.3.3 Tail Recursion

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

lemma *gfp-pulldown*:
fixes $P :: {'s}\ \mathit{expect}$
assumes *tailcall*: $\bigwedge u\ P.\ \mathit{unitary}\ P \Longrightarrow T\ u\ P = t\ P\ (u\ P)$
and *fT*: $\bigwedge t\ P.\ [\![\ \bigwedge Q.\ \mathit{unitary}\ Q \Longrightarrow \mathit{unitary}\ (t\ Q);\ \mathit{unitary}\ P\ ]\!] \Longrightarrow \mathit{unitary}\ (T\ t\ P)$
and *ft*: $\bigwedge P\ Q.\ \mathit{unitary}\ P \Longrightarrow \mathit{unitary}\ Q \Longrightarrow \mathit{unitary}\ (t\ P\ Q)$
and *mt*: $\bigwedge P\ Q\ R.\ [\![\ \mathit{unitary}\ P;\ \mathit{unitary}\ Q;\ \mathit{unitary}\ R;\ Q \Vdash R\ ]\!] \Longrightarrow t\ P\ Q \Vdash t\ P\ R$
and *uP*: $\mathit{unitary}\ P$
and *monoT*: $\bigwedge t\ u.\ [\![\ \mathit{le\text{-}utrans}\ t\ u;\ \bigwedge P.\ \mathit{unitary}\ P \Longrightarrow \mathit{unitary}\ (t\ P);$
$\bigwedge P.\ \mathit{unitary}\ P \Longrightarrow \mathit{unitary}\ (u\ P)\ ]\!] \Longrightarrow \mathit{le\text{-}utrans}\ (T\ t)\ (T\ u)$
shows $\mathit{gfp\text{-}trans}\ T\ P = \mathit{gfp\text{-}exp}\ (t\ P)$ (**is** $?X\ P = ?Y\ P$)
⟨*proof*⟩

lemma *lfp-pulldown*:
fixes $P :: {'s}\ \mathit{expect}$ **and** $t :: {'s}\ \mathit{expect} \Rightarrow {'s}\ \mathit{trans}$

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

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

lemma *Inf-utrans-lower*:
$[\![\ t \in S;\ \forall\, t{\in}S.\ \forall\, P.\ unitary\ P \longrightarrow unitary\ (t\ P)\ ]\!] \Longrightarrow le\text{-}utrans\ (Inf\text{-}utrans\ S)\ t$
$\langle proof \rangle$

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

end

Chapter 4

The pGCL Language

4.1 A Shallow Embedding of pGCL in HOL

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

4.1.1 Core Primitives and Syntax

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

type-synonym *′s prog* = *bool* ⇒ (*′s* ⇒ *real*) ⇒ (*′s* ⇒ *real*)

Abort either always fails, *λP s. 0*, or always succeeds, *λP s. 1*.

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

Skip does nothing at all.

definition *Skip* :: *′s prog*
where *Skip* ≡ *λab P. P*

Apply lifts a state transformer into the space of programs.

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

Seq is sequential composition.

definition *Seq* :: *′s prog* ⇒ *′s prog* ⇒ *′s prog*
(**infixl** ‹;;› *59*)
where *Seq a b* ≡ (*λab. a ab o b ab*)

PC is *probabilistic* choice between programs.

definition *PC* :: *′s prog* ⇒ (*′s* ⇒ *real*) ⇒ *′s prog* ⇒ *′s prog*
(‹- _⊕ -› [*58*,*57*,*57*] *57*)

```
where   PC a P b ≡ λab Q s. P s ∗ a ab Q s + (1 − P s) ∗ b ab Q s
```

DC is *demonic* choice between programs.

```
definition DC :: 's prog ⇒ 's prog ⇒ 's prog (‹- ⊓ -› [58,57] 57)
where   DC a b ≡ λab Q s. min (a ab Q s) (b ab Q s)
```

AC is *angelic* choice between programs.

```
definition AC :: 's prog ⇒ 's prog ⇒ 's prog (‹- ⊔ -› [58,57] 57)
where   AC a b ≡ λab Q s. max (a ab Q s) (b ab Q s)
```

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

```
definition Embed :: 's trans ⇒ 's prog
where   Embed t = (λab. t)
```

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

```
definition Mu :: ('s prog ⇒ 's prog) ⇒ 's prog (binder ‹μ› 50)
where   Mu(T) ≡ (λab. if ab then lfp-trans (λt. T (Embed t) ab)
                      else gfp-trans (λt. T (Embed t) ab))
```

repeat expresses finite repetition

```
primrec
  repeat :: nat ⇒ 'a prog ⇒ 'a prog
where
  repeat 0 p = Skip |
  repeat (Suc n) p = p ;; repeat n p
```

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

```
definition SetDC :: ('a ⇒ 's prog) ⇒ ('s ⇒ 'a set) ⇒ 's prog
  where SetDC f S ≡ λab P s. Inf ((λa. f a ab P s) ‘ S s)

syntax -SetDC :: pttrn => ('s => 'a set) => 's prog => 's prog
            (‹⊓-∈-./ -› 100)
syntax-consts -SetDC == SetDC
translations ⊓x∈S. p == CONST SetDC (%x. p) S
```

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

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

```
definition
  SetPC :: ('a ⇒ 's prog) ⇒ ('s ⇒ 'a ⇒ real) ⇒ 's prog
where
  SetPC f p ≡ λab P s. ∑a∈supp (p s). p s a ∗ f a ab P s
```

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

```
definition
  Bind :: ('s ⇒ 'a) ⇒ ('a ⇒ 's prog) ⇒ 's prog
where
  Bind g f ab ≡ λP s. let a = g s in f a ab P s
```

This gives us something like let syntax

```
syntax -Bind :: pttrn => ('s => 'a) => 's prog => 's prog
      (‹- is - in -› [55,55,55]55)
syntax-consts -Bind == Bind
translations x is f in a => CONST Bind f (%x. a)

definition flip :: ('a ⇒ 'b ⇒ 'c) ⇒ 'b ⇒ 'a ⇒ 'c
where [simp]: flip f = (λb a. f a b)
```

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

```
syntax -PBind :: pttrn => ('s => real) => 's prog => 's prog
            (‹bind - at - in -› [55,55,55]55)
syntax-consts -PBind == SetPC
translations bind x at p in a => CONST SetPC (%x. a) (CONST flip (%x. p))

syntax -DBind :: pttrn => ('s => 'a set) ⇒ 's prog => 's prog
            (‹bind - from - in -› [55,55,55]55)
syntax-consts -DBind == SetDC
translations bind x from S in a => CONST SetDC (%x. a) S
```

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

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

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

syntax
  -set-dc :: ident => ('s => 'a set) => 's prog (‹- :∈ -› [66,66]66)
syntax-consts
  -set-dc ⇌ SetDC
⟨ML⟩
```

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

```
syntax
```

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

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

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

If-Then-Else as a degenerate probabilistic choice.

```
abbreviation(input)
  if-then-else :: ['s ⇒ bool, 's prog, 's prog] ⇒ 's prog
     (‹If - Then - Else -› 58)
where
  If P Then a Else b == a «P»⊕ b
```

Syntax for loops

```
abbreviation
  do-while :: ['s ⇒ bool, 's prog] ⇒ 's prog
           (‹do - ⟶// (4 -) //od›)
where
  do-while P a ≡ μ x. If P Then a ;; x Else Skip
```

4.1.2 Unfolding rules for non-recursive primitives

```
lemma eval-wp-Abort:
  wp Abort P = (λs. 0)
  ⟨proof⟩

lemma eval-wlp-Abort:
  wlp Abort P = (λs. 1)
  ⟨proof⟩

lemma eval-wp-Skip:
  wp Skip P = P
  ⟨proof⟩

lemma eval-wlp-Skip:
  wlp Skip P = P
  ⟨proof⟩

lemma eval-wp-Apply:
```

$wp\ (Apply\,f)\ P = P\ o\ f$
$\langle proof\rangle$

lemma *eval-wlp-Apply*:
$wlp\ (Apply\,f)\ P = P\ o\ f$
$\langle proof\rangle$

lemma *eval-wp-Seq*:
$wp\ (a\ ;;\ b)\ P = (wp\ a\ o\ wp\ b)\ P$
$\langle proof\rangle$

lemma *eval-wlp-Seq*:
$wlp\ (a\ ;;\ b)\ P = (wlp\ a\ o\ wlp\ b)\ P$
$\langle proof\rangle$

lemma *eval-wp-PC*:
$wp\ (a\ {}_{Q}{\oplus}\ b)\ P = (\lambda s.\ Q\ s * wp\ a\ P\ s + (1 - Q\ s) * wp\ b\ P\ s)$
$\langle proof\rangle$

lemma *eval-wlp-PC*:
$wlp\ (a\ {}_{Q}{\oplus}\ b)\ P = (\lambda s.\ Q\ s * wlp\ a\ P\ s + (1 - Q\ s) * wlp\ b\ P\ s)$
$\langle proof\rangle$

lemma *eval-wp-DC*:
$wp\ (a \bigsqcap b)\ P = (\lambda s.\ min\ (wp\ a\ P\ s)\ (wp\ b\ P\ s))$
$\langle proof\rangle$

lemma *eval-wlp-DC*:
$wlp\ (a \bigsqcap b)\ P = (\lambda s.\ min\ (wlp\ a\ P\ s)\ (wlp\ b\ P\ s))$
$\langle proof\rangle$

lemma *eval-wp-AC*:
$wp\ (a \bigsqcup b)\ P = (\lambda s.\ max\ (wp\ a\ P\ s)\ (wp\ b\ P\ s))$
$\langle proof\rangle$

lemma *eval-wlp-AC*:
$wlp\ (a \bigsqcup b)\ P = (\lambda s.\ max\ (wlp\ a\ P\ s)\ (wlp\ b\ P\ s))$
$\langle proof\rangle$

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*:
$wlp\ (SetDC\ p\ S)\ R\ s = Inf\ ((\lambda a.\ wlp\ (p\ a)\ R\ s)\ \text{‘}\ S\ s)$
$\langle proof \rangle$

lemma *eval-wp-SetPC*:
$wp\ (SetPC\ f\ p)\ P = (\lambda s.\ \sum a{\in}supp\ (p\ s).\ p\ s\ a * wp\ (f\ a)\ P\ s)$
$\langle proof \rangle$

lemma *eval-wlp-SetPC*:
$wlp\ (SetPC\ f\ p)\ P = (\lambda s.\ \sum a{\in}supp\ (p\ s).\ p\ s\ a * wlp\ (f\ a)\ P\ s)$
$\langle proof \rangle$

lemma *eval-wp-Mu*:
$wp\ (\mu\ t.\ T\ t) = lfp\text{-}trans\ (\lambda t.\ wp\ (T\ (Embed\ t)))$
$\langle proof \rangle$

lemma *eval-wlp-Mu*:
$wlp\ (\mu\ t.\ T\ t) = gfp\text{-}trans\ (\lambda t.\ wlp\ (T\ (Embed\ t)))$
$\langle proof \rangle$

lemma *eval-wp-Bind*:
$wp\ (Bind\ g\ f) = (\lambda P\ s.\ wp\ (f\ (g\ s))\ P\ s)$
$\langle proof \rangle$

lemma *eval-wlp-Bind*:
$wlp\ (Bind\ g\ f) = (\lambda P\ s.\ wlp\ (f\ (g\ s))\ P\ s)$
$\langle proof \rangle$

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*:
$Skip\ ;;\ A = A$
$\langle proof \rangle$

lemma *Seq-Skip*:
$A\ ;;\ Skip = A$
$\langle proof \rangle$

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* ${}_{P}\oplus$ *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: healthy (wp f) and hg: healthy (wp g)
 shows healthy (wp (f ⨅ g))
⟨proof⟩

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

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

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

lemma healthy-wp-Embed:
 healthy t ⟹ healthy (wp (Embed t))
 ⟨proof⟩

lemma nearly-healthy-wlp-Embed:
 nearly-healthy t ⟹ nearly-healthy (wlp (Embed t))
 ⟨proof⟩

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

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

```
lemma healthy-wp-SetDC:
  fixes prog::'b ⇒ 'a prog and S::'a ⇒ 'b set
  assumes healthy: ⋀x s. x ∈ S s ⟹ healthy (wp (prog x))
     and nonempty: ⋀s. ∃x. x ∈ S s
  shows healthy (wp (SetDC prog S)) (is healthy ?T)
⟨proof⟩

lemma nearly-healthy-wlp-SetDC:
  fixes prog::'b ⇒ 'a prog and S::'a ⇒ 'b set
  assumes healthy: ⋀x s. x ∈ S s ⟹ nearly-healthy (wlp (prog x))
     and nonempty: ⋀s. ∃x. x ∈ S s
  shows nearly-healthy (wlp (SetDC prog S)) (is nearly-healthy ?T)
⟨proof⟩

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

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

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

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

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

lemma nearly-healthy-wlp-Bind:
  fixes f::'s ⇒ 'a
```

assumes *hsub*: $\bigwedge s.$ *nearly-healthy* (*wlp* (*p* (*f s*)))
shows *nearly-healthy* (*wlp* (*Bind f p*))
⟨*proof*⟩

4.2.2 Healthiness for Loops

lemma *wp-loop-step-mono*:
fixes *t u*::*'s trans*
assumes *hb*: *healthy* (*wp body*)
and *le*: *le-trans t u*
and *ht*: $\bigwedge P.$ *sound P* $\Longrightarrow$ *sound* (*t P*)
and *hu*: $\bigwedge P.$ *sound P* $\Longrightarrow$ *sound* (*u P*)
shows *le-trans* (*wp* (*body* ;; *Embed t* « *G* »$\oplus$ *Skip*))
(*wp* (*body* ;; *Embed u* « *G* »$\oplus$ *Skip*))
⟨*proof*⟩

lemma *wlp-loop-step-mono*:
fixes *t u*::*'s trans*
assumes *mb*: *nearly-healthy* (*wlp body*)
and *le*: *le-utrans t u*
and *ht*: $\bigwedge P.$ *unitary P* $\Longrightarrow$ *unitary* (*t P*)
and *hu*: $\bigwedge P.$ *unitary P* $\Longrightarrow$ *unitary* (*u P*)
shows *le-utrans* (*wlp* (*body* ;; *Embed t* « *G* »$\oplus$ *Skip*))
(*wlp* (*body* ;; *Embed u* « *G* »$\oplus$ *Skip*))
⟨*proof*⟩

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

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

lemma *lfp-loop-greatest*:
fixes *P*::*'s expect*
assumes *lb*: $\bigwedge R.$ $\lambda s.$ «*G*» *s* $*$ *wp body R s* $+$ «$\mathcal{N}$ *G*» *s* $*$ *P s* $\Vdash$ *R* $\Longrightarrow$ *sound R* $\Longrightarrow$ *Q* $\Vdash$ *R*
and *hb*: *healthy* (*wp body*)
and *sP*: *sound P*
and *sQ*: *sound Q*
shows *Q* $\Vdash$ *lfp-exp* (λQ *s*. «*G*» *s* $*$ *wp body Q s* $+$ «$\mathcal{N}$ *G*» *s* $*$ *P s*)
⟨*proof*⟩

lemma *lfp-loop-sound*:
fixes *P*::*'s expect*
assumes *hb*: *healthy* (*wp body*)
and *sP*: *sound P*
shows *sound* (*lfp-exp* (λQ *s*. «*G*» *s* $*$ *wp body Q s* $+$ «$\mathcal{N}$ *G*» *s* $*$ *P s*))
⟨*proof*⟩

```
lemma wlp-loop-step-unitary:
 fixes t u::'s trans
 assumes hb: nearly-healthy (wlp body)
   and ht: ⋀P. unitary P ⟹ unitary (t P)
   and uP: unitary P
 shows unitary (wlp (body ;; Embed t « G »⊕ Skip) P)
⟨proof⟩

lemma wp-loop-step-sound:
 fixes t u::'s trans
 assumes hb: healthy (wp body)
   and ht: ⋀P. sound P ⟹ sound (t P)
   and sP: sound P
 shows sound (wp (body ;; Embed t « G »⊕ Skip) P)
⟨proof⟩
```

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

```
lemma wlp-Loop1:
 fixes body :: 's prog
 assumes unitary: unitary P
   and healthy: nearly-healthy (wlp body)
 shows wlp (do G ⟶ body od) P =
  gfp-exp (λQ s. «G» s ∗ wlp body Q s + «𝒩 G» s ∗ P s)
 (is ?X = gfp-exp (?Y P))
⟨proof⟩

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

Likewise, we can rewrite strict loops.

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

lemma nearly-healthy-wlp-loop:
 fixes body::'s prog
 assumes hb: nearly-healthy (wlp body)
 shows nearly-healthy (wlp (do G ⟶ body od))
⟨proof⟩
```

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

lemma *healthy-wp-loop*:
fixes $body{::}'s\ prog$
assumes hb: $healthy\ (wp\ body)$
shows $healthy\ (wp\ (do\ G \longrightarrow body\ od))$
$\langle proof \rangle$

Use 'simp add:healthy_intros' or 'blast intro:healthy_intros' as appropriate to discharge healthiness side-contitions 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 seperately, 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 $bd\text{-}cts :: 's\ trans \Rightarrow bool$
where $bd\text{-}cts\ t = (\forall M.\ (\forall i.\ (M\ i \Vdash M\ (Suc\ i)) \wedge sound\ (M\ i)) \longrightarrow$
$(\exists b.\ \forall i.\ bounded\text{-}by\ b\ (M\ i)) \longrightarrow$
$t\ (Sup\text{-}exp\ (range\ M)) = Sup\text{-}exp\ (range\ (t\ o\ M)))$

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

lemma *bd-ctsI*:
$(\bigwedge b\ M.\ (\bigwedge i.\ M\ i \Vdash M\ (Suc\ i)) \Longrightarrow (\bigwedge i.\ sound\ (M\ i)) \Longrightarrow (\bigwedge i.\ bounded\text{-}by\ b\ (M\ i)) \Longrightarrow$
$t\ (Sup\text{-}exp\ (range\ M)) = Sup\text{-}exp\ (range\ (t\ o\ M))) \Longrightarrow bd\text{-}cts\ t$
$\langle proof \rangle$

A generalised property for transformers of transformers.

definition $bd\text{-}cts\text{-}tr :: ('s\ trans \Rightarrow 's\ trans) \Rightarrow bool$

where $\mathit{bd\text{-}cts\text{-}tr}\ T = (\forall M.\ (\forall i.\ \mathit{le\text{-}trans}\ (M\ i)\ (M\ (\mathit{Suc}\ i)) \wedge \mathit{feasible}\ (M\ i)) \longrightarrow$
$\quad\mathit{equiv\text{-}trans}\ (T\ (\mathit{Sup\text{-}trans}\ (M\ `\ \mathit{UNIV})))\ (\mathit{Sup\text{-}trans}\ ((T\ o\ M)\ `\ \mathit{UNIV})))$

lemma *bd-cts-trD*:
$[\![\ \mathit{bd\text{-}cts\text{-}tr}\ T;\ \bigwedge i.\ \mathit{le\text{-}trans}\ (M\ i)\ (M\ (\mathit{Suc}\ i));\ \bigwedge i.\ \mathit{feasible}\ (M\ i)\ ]\!] \Longrightarrow$
$\mathit{equiv\text{-}trans}\ (T\ (\mathit{Sup\text{-}trans}\ (M\ `\ \mathit{UNIV})))\ (\mathit{Sup\text{-}trans}\ ((T\ o\ M)\ `\ \mathit{UNIV}))$
$\langle \mathit{proof} \rangle$

lemma *bd-cts-trI*:
$(\bigwedge M.\ (\bigwedge i.\ \mathit{le\text{-}trans}\ (M\ i)\ (M\ (\mathit{Suc}\ i))) \Longrightarrow (\bigwedge i.\ \mathit{feasible}\ (M\ i)) \Longrightarrow$
$\quad\mathit{equiv\text{-}trans}\ (T\ (\mathit{Sup\text{-}trans}\ (M\ `\ \mathit{UNIV})))\ (\mathit{Sup\text{-}trans}\ ((T\ o\ M)\ `\ \mathit{UNIV}))) \Longrightarrow \mathit{bd\text{-}cts\text{-}tr}$
T
$\langle \mathit{proof} \rangle$

4.3.1 Continuity of Primitives

lemma *cts-wp-Abort*:
$\mathit{bd\text{-}cts}\ (\mathit{wp}\ (\mathit{Abort}::'s\ \mathit{prog}))$
$\langle \mathit{proof} \rangle$

lemma *cts-wp-Skip*:
$\mathit{bd\text{-}cts}\ (\mathit{wp}\ \mathit{Skip})$
$\langle \mathit{proof} \rangle$

lemma *cts-wp-Apply*:
$\mathit{bd\text{-}cts}\ (\mathit{wp}\ (\mathit{Apply}\ f))$
$\langle \mathit{proof} \rangle$

lemma *cts-wp-Bind*:
fixes $a::'a \Rightarrow\ 's\ \mathit{prog}$
assumes ca: $\bigwedge s.\ \mathit{bd\text{-}cts}\ (\mathit{wp}\ (a\ (f\ s)))$
shows $\mathit{bd\text{-}cts}\ (\mathit{wp}\ (\mathit{Bind}\ f\ a))$
$\langle \mathit{proof} \rangle$

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 $a\ b::'s\ \mathit{prog}$
assumes ca: $\mathit{bd\text{-}cts}\ (\mathit{wp}\ a)$
and cb: $\mathit{bd\text{-}cts}\ (\mathit{wp}\ b)$
and ha: $\mathit{healthy}\ (\mathit{wp}\ a)$
and hb: $\mathit{healthy}\ (\mathit{wp}\ b)$
shows $\mathit{bd\text{-}cts}\ (\mathit{wp}\ (a \bigsqcap b))$
$\langle \mathit{proof} \rangle$

lemma *cts-wp-Seq*:
fixes $a\ b::'s\ \mathit{prog}$
assumes ca: $\mathit{bd\text{-}cts}\ (\mathit{wp}\ a)$

and *cb*: *bd-cts* (*wp b*)
and *hb*: *healthy* (*wp b*)
shows *bd-cts* (*wp* (*a* ;; *b*))
⟨*proof*⟩

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

Both set-based choice operators are only continuous for finite sets (probabilistic choice *can* be extended infinitely, but we have not done so). The proofs for both are inductive, and rely on the above results on binary operators.

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

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

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

lemma *wp-SetPC-nil*:
wp (*SetPC a* (λ*s a*. *0*)) = (λ*P s*. *0*)
⟨*proof*⟩

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

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

lemma *cts-wp-SetPC-const*:

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

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

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

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

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

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

lemma cts-wp-SetDC:
  fixes a::'a ⇒ 's prog
  assumes ca: ⋀x s. x ∈ S s ⟹ bd-cts (wp (a x))
    and ha: ⋀x s. x ∈ S s ⟹ healthy (wp (a x))
    and fS: ⋀s. finite (S s)
    and neS: ⋀s. S s ≠ {}
  shows bd-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*:
$bd\text{-}cts\ t \Longrightarrow bd\text{-}cts\ (wp\ (Embed\ t))$
$\langle proof \rangle$

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 $lfp\text{-}exp\ (\lambda Q\ s.\ \ll G \gg s * wp\ body\ Q\ s + \ll \mathcal{N}\ G \gg s * P\ s) =$
$(\lambda s.\ \ll G \gg s * wp\ body\ (lfp\text{-}exp\ (\lambda Q\ s.\ \ll G \gg s * wp\ body\ Q\ s + \ll \mathcal{N}\ G \gg s * P\ s))\ s\ +$

```
        «𝒩 G» s ∗ P s)
⟨proof⟩
```

```
lemma wp-loop-step-unitary:
  fixes body::'s prog
  assumes hb: healthy (wp body)
    and uP: unitary P and uQ: unitary Q
  shows unitary (λs. «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s)
⟨proof⟩
```

```
lemma lfp-loop-unitary:
  fixes body::'s prog
  assumes hb: healthy (wp body)
    and uP: unitary P
  shows unitary (lfp-exp (λQ s. «G» s ∗ wp body Q s + «𝒩 G» s ∗ P s))
  ⟨proof⟩
```

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

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

4.4.1 The Limit of Iterates

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

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

where *iterates body G i* = ((λ*x*. *wp* (*body* ;; *Embed x* $_{«G»}$⊕ *Skip*)) ^^ *i*) (λ*P s*. *0*)

lemma *iterates-0*[*simp*]:
iterates body G 0 = (λ*P s*. *0*)
⟨*proof*⟩

lemma *iterates-Suc*[*simp*]:
iterates body G (*Suc i*) = *wp* (*body* ;; *Embed* (*iterates body G i*) $_{«G»}$⊕ *Skip*)
⟨*proof*⟩

All iterates are healthy.

lemma *iterates-healthy*:
healthy (*wp body*) ⟹ *healthy* (*iterates body G i*)
⟨*proof*⟩

The iterates are an ascending chain.

lemma *iterates-increasing*:
fixes *body*::′*s prog*
assumes *hb*: *healthy* (*wp body*)
shows *le-trans* (*iterates body G i*) (*iterates body G* (*Suc i*))
⟨*proof*⟩

lemma *wp-loop-step-bounded*:
fixes *t*::′*s trans* **and** *Q*::′*s expect*
assumes *nQ*: *nneg Q*
and *bQ*: *bounded-by b Q*
and *ht*: *healthy t*
and *hb*: *healthy* (*wp body*)
shows *bounded-by b* (*wp* (*body* ;; *Embed t* $_{«G»}$⊕ *Skip*) *Q*)
⟨*proof*⟩

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

lemma *lfp-iterates*:
fixes *body*::′*s prog*
assumes *hb*: *healthy* (*wp body*)
and *cb*: *bd-cts* (*wp body*)
shows *equiv-trans* (*wp* (*do G* ⟶ *body od*)) (*Sup-trans* (*range* (*iterates body G*)))
(**is** *equiv-trans ?X ?Y*)
⟨*proof*⟩

Therefore, evaluated at a given point (state), the sequence of iterates gives a sequence of real values that converges on that of the loop itself.

corollary *loop-iterates*:
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*) $\longrightarrow$ *wp* (*do G* $\longrightarrow$ *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 alebraic 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*))

$\langle proof \rangle$

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* $(wp\ (x\ ;;\ y))$
$\langle proof \rangle$

lemma *sublinear-wp-PC*:
fixes x::*'s prog*
assumes *slx*: *sublinear* $(wp\ x)$ **and** *sly*: *sublinear* $(wp\ y)$
and *uP*: *unitary P*
shows *sublinear* $(wp\ (x\ {}_P{\oplus}\ y))$
$\langle proof \rangle$

lemma *sublinear-wp-DC*:
fixes x::*'s prog*
assumes *slx*: *sublinear* $(wp\ x)$ **and** *sly*: *sublinear* $(wp\ y)$
shows *sublinear* $(wp\ (x \bigsqcap y))$
$\langle proof \rangle$

As for continuity, we insist on a finite support.

lemma *sublinear-wp-SetPC*:
fixes p::*'a ⇒ 's prog*
assumes *slp*: $\bigwedge s\ a.\ a \in supp\ (P\ s) \Longrightarrow sublinear\ (wp\ (p\ a))$
and *sum*: $\bigwedge s.\ (\sum a{\in}supp\ (P\ s).\ P\ s\ a) \leq 1$
and *nnP*: $\bigwedge s\ a.\ 0 \leq P\ s\ a$
and *fin*: $\bigwedge s.\ finite\ (supp\ (P\ s))$
shows *sublinear* $(wp\ (SetPC\ p\ P))$
$\langle proof \rangle$

lemma *sublinear-wp-SetDC*:
fixes p::*'a ⇒ 's prog*
assumes *slp*: $\bigwedge s\ a.\ a \in S\ s \Longrightarrow sublinear\ (wp\ (p\ a))$
and *hp*: $\bigwedge s\ a.\ a \in S\ s \Longrightarrow healthy\ (wp\ (p\ a))$
and *ne*: $\bigwedge s.\ S\ s \neq \{\}$
shows *sublinear* $(wp\ (SetDC\ p\ S))$
$\langle proof \rangle$

lemma *sublinear-wp-Embed*:
$sublinear\ t \Longrightarrow sublinear\ (wp\ (Embed\ t))$
$\langle proof \rangle$

lemma *sublinear-wp-repeat*:
$[\![\ sublinear\ (wp\ p);\ healthy\ (wp\ p)\ ]\!] \Longrightarrow sublinear\ (wp\ (repeat\ n\ p))$
$\langle proof \rangle$

lemma *sublinear-wp-Bind*:

```
⟦ ⋀s. sublinear (wp (a (f s))) ⟧ ⟹ sublinear (wp (Bind f a))
⟨proof⟩
```

4.5.2 Sublinearity for Loops

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

```
lemma sub-distrib-wp-loop:
 fixes body::'s prog
 assumes sdb: sub-distrib (wp body)
   and hb: healthy (wp body)
   and nhb: nearly-healthy (wlp body)
 shows sub-distrib (wp (do G ⟶ body od))
⟨proof⟩
```

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

```
lemma sublinear-iterates:
 assumes hb: healthy (wp body)
   and sb: sublinear (wp body)
 shows sublinear (iterates body G i)
 ⟨proof⟩
```

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

```
lemma sub-add-wp-loop:
 fixes body::'s prog
 assumes sb: sublinear (wp body)
   and cb: bd-cts (wp body)
   and hwp: healthy (wp body)
 shows sub-add (wp (do G ⟶ body od))
⟨proof⟩

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

lemmas sublinear-intros =
 sublinear-wp-Abort
 sublinear-wp-Skip
 sublinear-wp-Apply
 sublinear-wp-Seq
 sublinear-wp-PC
```

sublinear-wp-DC
sublinear-wp-SetPC
sublinear-wp-SetDC
sublinear-wp-Embed
sublinear-wp-repeat
sublinear-wp-Bind
sublinear-wp-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*:
$\quad additive\ (wp\ (Abort))$
$\quad \langle proof\rangle$

wlp Abort is not additive.

lemma *additive-wp-Skip*:
$\quad additive\ (wp\ (Skip))$
$\quad \langle proof\rangle$

lemma *additive-wp-Apply*:
$\quad additive\ (wp\ (Apply\ f))$
$\quad \langle proof\rangle$

lemma *additive-wp-Seq*:
 fixes $a{::}'s\ prog$
 assumes $adda$: $additive\ (wp\ a)$
  and $addb$: $additive\ (wp\ b)$
  and wb: $well\text{-}def\ b$
 shows $additive\ (wp\ (a\ ;;\ b))$
$\langle proof\rangle$

lemma *additive-wp-PC*:
$\quad [\![\ additive\ (wp\ a);\ additive\ (wp\ b)\ ]\!] \Longrightarrow additive\ (wp\ (a\ {}_{P}{\oplus}\ b))$
$\quad \langle proof\rangle$

DC is not additive.

lemma *additive-wp-SetPC*:
$\quad [\![\ \bigwedge x\ s.\ x \in supp\ (p\ s) \Longrightarrow additive\ (wp\ (a\ x));\ \bigwedge s.\ finite\ (supp\ (p\ s))\ ]\!] \Longrightarrow$

$\mathit{additive}\ (\mathit{wp}\ (\mathit{SetPC}\ a\ p))$
$\langle \mathit{proof} \rangle$

lemma *additive-wp-Bind*:
$\llbracket\ \bigwedge x.\ \mathit{additive}\ (\mathit{wp}\ (a\ (f\,x)))\ \rrbracket \Longrightarrow \mathit{additive}\ (\mathit{wp}\ (\mathit{Bind}\,f\,a))$
$\langle \mathit{proof} \rangle$

lemma *additive-wp-Embed*:
$\llbracket\ \mathit{additive}\ t\ \rrbracket \Longrightarrow \mathit{additive}\ (\mathit{wp}\ (\mathit{Embed}\ t))$
$\langle \mathit{proof} \rangle$

lemma *additive-wp-repeat*:
$\mathit{additive}\ (\mathit{wp}\ a) \Longrightarrow \mathit{well\text{-}def}\ a \Longrightarrow \mathit{additive}\ (\mathit{wp}\ (\mathit{repeat}\ n\ a))$
$\langle \mathit{proof} \rangle$

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*:
$\mathit{maximal}\ (\mathit{wp}\ \mathit{Skip})$
$\langle \mathit{proof} \rangle$

lemma *max-wp-Apply*:
$\mathit{maximal}\ (\mathit{wp}\ (\mathit{Apply}\,f))$
$\langle \mathit{proof} \rangle$

lemma *max-wp-Seq*:
$\llbracket\ \mathit{maximal}\ (\mathit{wp}\ a);\ \mathit{maximal}\ (\mathit{wp}\ b)\ \rrbracket \Longrightarrow \mathit{maximal}\ (\mathit{wp}\ (a\ ;;\ b))$
$\langle \mathit{proof} \rangle$

lemma *max-wp-PC*:
$\llbracket\ \mathit{maximal}\ (\mathit{wp}\ a);\ \mathit{maximal}\ (\mathit{wp}\ b)\ \rrbracket \Longrightarrow \mathit{maximal}\ (\mathit{wp}\ (a\ {}_{P}{\oplus}\ b))$
$\langle \mathit{proof} \rangle$

lemma *max-wp-DC*:
$\llbracket\ \mathit{maximal}\ (\mathit{wp}\ a);\ \mathit{maximal}\ (\mathit{wp}\ b)\ \rrbracket \Longrightarrow \mathit{maximal}\ (\mathit{wp}\ (a \bigsqcap b))$
$\langle \mathit{proof} \rangle$

lemma *max-wp-SetPC*:
$\llbracket\ \bigwedge s\ a.\ a \in \mathit{supp}\ (P\ s) \Longrightarrow \mathit{maximal}\ (\mathit{wp}\ (p\ a));\ \bigwedge s.\ (\sum a{\in}\mathit{supp}\ (P\ s).\ P\ s\ a) = \mathit{1}\ \rrbracket \Longrightarrow$
$\mathit{maximal}\ (\mathit{wp}\ (\mathit{SetPC}\ p\ P))$
$\langle \mathit{proof} \rangle$

```
lemma max-wp-SetDC:
  fixes p::'a ⇒ 's prog
  assumes mp: ⋀s a. a ∈ S s ⟹ maximal (wp (p a))
    and ne: ⋀s. S s ≠ {}
  shows maximal (wp (SetDC p S))
⟨proof⟩

lemma max-wp-Embed:
  maximal t ⟹ maximal (wp (Embed t))
  ⟨proof⟩

lemma max-wp-repeat:
  maximal (wp a) ⟹ maximal (wp (repeat n a))
  ⟨proof⟩

lemma max-wp-Bind:
  assumes ma: ⋀s. maximal (wp (a (f s)))
  shows maximal (wp (Bind f a))
⟨proof⟩

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

A healthy transformer that terminates is maximal.

```
lemma healthy-term-max:
  assumes ht: healthy t
    and trm: λs. 1 ⊩ t (λs. 1)
  shows maximal t
⟨proof⟩
```

4.6.3 Determinism

```
lemma det-wp-Skip:
  determ (wp Skip)
  ⟨proof⟩

lemma det-wp-Apply:
  determ (wp (Apply f))
  ⟨proof⟩

lemma det-wp-Seq:
  determ (wp a) ⟹ determ (wp b) ⟹ well-def b ⟹ determ (wp (a ;; b))
  ⟨proof⟩

lemma det-wp-PC:
```

determ (*wp a*) ⟹ *determ* (*wp b*) ⟹ *determ* (*wp* (*a* ${}_{p}\oplus$ *b*))
⟨*proof*⟩

lemma *det-wp-SetPC*:
(⋀*x s*. *x* ∈ *supp* (*p s*) ⟹ *determ* (*wp* (*a x*))) ⟹
(⋀*s*. *finite* (*supp* (*p s*))) ⟹
(⋀*s*. *sum* (*p s*) (*supp* (*p s*)) = *1*) ⟹
determ (*wp* (*SetPC a p*))
⟨*proof*⟩

lemma *det-wp-Bind*:
(⋀*x*. *determ* (*wp* (*a* (*f x*)))) ⟹ *determ* (*wp* (*Bind f a*))
⟨*proof*⟩

lemma *det-wp-Embed*:
determ t ⟹ *determ* (*wp* (*Embed t*))
⟨*proof*⟩

lemma *det-wp-repeat*:
determ (*wp a*) ⟹ *well-def a* ⟹ *determ* (*wp* (*repeat n a*))
⟨*proof*⟩

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

end

4.7 Well-Defined Programs.

theory *WellDefined* **imports**
Healthiness
Sublinearity
LoopInduction
begin

The definition of a well-defined program collects the various notions of healthiness and well-behavedness that we have so far established: healthiness of the strict and liberal transformers, continuity and sublinearity of the strict transformers, and two new properties. These are that the strict transformer always lies below the liberal one (i.e. that it is at least as *strict*, recalling the standard embedding of a predicate), and that expectation 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
$\mathit{wp\text{-}under\text{-}wlp} :: {}'s\ \mathit{prog} \Rightarrow \mathit{bool}$
where
$\mathit{wp\text{-}under\text{-}wlp}\ \mathit{prog} \equiv \forall P.\ \mathit{unitary}\ P \longrightarrow \mathit{wp}\ \mathit{prog}\ P \Vdash \mathit{wlp}\ \mathit{prog}\ P$

lemma $\mathit{wp\text{-}under\text{-}wlpI}[\mathit{intro}]$:
$[\![\bigwedge P.\ \mathit{unitary}\ P \Longrightarrow \mathit{wp}\ \mathit{prog}\ P \Vdash \mathit{wlp}\ \mathit{prog}\ P\ ]\!] \Longrightarrow \mathit{wp\text{-}under\text{-}wlp}\ \mathit{prog}$
$\langle \mathit{proof} \rangle$

lemma $\mathit{wp\text{-}under\text{-}wlpD}[\mathit{dest}]$:
$[\![\mathit{wp\text{-}under\text{-}wlp}\ \mathit{prog};\ \mathit{unitary}\ P\ ]\!] \Longrightarrow \mathit{wp}\ \mathit{prog}\ P \Vdash \mathit{wlp}\ \mathit{prog}\ P$
$\langle \mathit{proof} \rangle$

lemma $\mathit{wp\text{-}under\text{-}le\text{-}trans}$:
$\mathit{wp\text{-}under\text{-}wlp}\ a \Longrightarrow \mathit{le\text{-}utrans}\ (\mathit{wp}\ a)\ (\mathit{wlp}\ a)$
$\langle \mathit{proof} \rangle$

lemma $\mathit{wp\text{-}under\text{-}wlp\text{-}Abort}$:
$\mathit{wp\text{-}under\text{-}wlp}\ \mathit{Abort}$
$\langle \mathit{proof} \rangle$

lemma $\mathit{wp\text{-}under\text{-}wlp\text{-}Skip}$:
$\mathit{wp\text{-}under\text{-}wlp}\ \mathit{Skip}$
$\langle \mathit{proof} \rangle$

lemma $\mathit{wp\text{-}under\text{-}wlp\text{-}Apply}$:
$\mathit{wp\text{-}under\text{-}wlp}\ (\mathit{Apply}\ f)$
$\langle \mathit{proof} \rangle$

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

lemma $\mathit{wp\text{-}under\text{-}wlp\text{-}PC}$:
assumes $\mathit{h\text{-}wp\text{-}a}$: $\mathit{healthy}\ (\mathit{wp}\ a)$
and $\mathit{h\text{-}wlp\text{-}a}$: $\mathit{nearly\text{-}healthy}\ (\mathit{wlp}\ a)$
and $\mathit{h\text{-}wp\text{-}b}$: $\mathit{healthy}\ (\mathit{wp}\ b)$
and $\mathit{h\text{-}wlp\text{-}b}$: $\mathit{nearly\text{-}healthy}\ (\mathit{wlp}\ b)$
and $\mathit{wp\text{-}u\text{-}a}$: $\mathit{wp\text{-}under\text{-}wlp}\ a$
and $\mathit{wp\text{-}u\text{-}b}$: $\mathit{wp\text{-}under\text{-}wlp}\ b$

```
    and uP:     unitary P
  shows wp-under-wlp (a P⊕ b)
⟨proof⟩

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

lemma wp-under-wlp-SetPC:
  assumes wp-u-f: ⋀s a. a ∈ supp (P s) ⟹ wp-under-wlp (f a)
    and nP:     ⋀s a. a ∈ supp (P s) ⟹ 0 ≤ P s a
  shows wp-under-wlp (SetPC f P)
⟨proof⟩

lemma wp-under-wlp-SetDC:
  assumes wp-u-f: ⋀s a. a ∈ S s ⟹ wp-under-wlp (f a)
    and hf:     ⋀s a. a ∈ S s ⟹ healthy (wp (f a))
    and nS:     ⋀s. S s ≠ {}
  shows wp-under-wlp (SetDC f S)
⟨proof⟩

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

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

lemma wp-under-wlp-repeat:
  ⟦ healthy (wp a); nearly-healthy (wlp a); wp-under-wlp a ⟧ ⟹
   wp-under-wlp (repeat n a)
  ⟨proof⟩

lemma wp-under-wlp-Bind:
  ⟦ ⋀s. wp-under-wlp (a (f s)) ⟧ ⟹ wp-under-wlp (Bind f a)
  ⟨proof⟩

lemmas wp-under-wlp-intros =
  wp-under-wlp-Abort wp-under-wlp-Skip
  wp-under-wlp-Apply wp-under-wlp-Seq
  wp-under-wlp-PC    wp-under-wlp-DC
  wp-under-wlp-SetPC wp-under-wlp-SetDC
```

wp-under-wlp-Embed wp-under-wlp-loop
wp-under-wlp-repeat wp-under-wlp-Bind

4.7.2 Sub-Distributivity of Conjunction

definition
sub-distrib-pconj :: *'s prog* ⇒ *bool*
where
sub-distrib-pconj prog ≡
∀ *P Q*. *unitary P* ⟶ *unitary Q* ⟶
wlp prog P && *wp prog Q* ⊩ *wp prog* (*P* && *Q*)

lemma *sub-distrib-pconjI*[*intro*]:
⟦⋀*P Q*. ⟦ *unitary P*; *unitary Q* ⟧ ⟹ *wlp prog P* && *wp prog Q* ⊩ *wp prog* (*P* && *Q*) ⟧
⟹
sub-distrib-pconj prog
⟨*proof*⟩

lemma *sub-distrib-pconjD*[*dest*]:
⋀*P Q*. ⟦ *sub-distrib-pconj prog*; *unitary P*; *unitary Q* ⟧ ⟹
wlp prog P && *wp prog Q* ⊩ *wp prog* (*P* && *Q*)
⟨*proof*⟩

lemma *sdp-Abort*:
sub-distrib-pconj Abort
⟨*proof*⟩

lemma *sdp-Skip*:
sub-distrib-pconj Skip
⟨*proof*⟩

lemma *sdp-Seq*:
fixes *a* **and** *b*
assumes *sdp-a*: *sub-distrib-pconj a*
and *sdp-b*: *sub-distrib-pconj b*
and *h-wp-a*: *healthy* (*wp a*)
and *h-wp-b*: *healthy* (*wp b*)
and *h-wlp-b*: *nearly-healthy* (*wlp b*)
shows *sub-distrib-pconj* (*a* ;; *b*)
⟨*proof*⟩

lemma *sdp-Apply*:
sub-distrib-pconj (*Apply f*)
⟨*proof*⟩

lemma *sdp-DC*:
fixes *a*::*'s prog* **and** *b*
assumes *sdp-a*: *sub-distrib-pconj a*
and *sdp-b*: *sub-distrib-pconj b*

```
    and h-wp-a:  healthy (wp a)
    and h-wp-b:  healthy (wp b)
    and h-wlp-b: nearly-healthy (wlp b)
  shows sub-distrib-pconj (a ⨅ b)
⟨proof⟩

lemma sdp-PC:
  fixes a::'s prog and b
  assumes sdp-a:  sub-distrib-pconj a
    and sdp-b:  sub-distrib-pconj b
    and h-wp-a:  healthy (wp a)
    and h-wp-b:  healthy (wp b)
    and h-wlp-b: nearly-healthy (wlp b)
    and uP:     unitary P
  shows sub-distrib-pconj (a P⊕ b)
⟨proof⟩

lemma sdp-Embed:
  ⟦ ⋀P Q. ⟦ unitary P; unitary Q ⟧ ⟹ t P && t Q ⊩ t (P && Q) ⟧ ⟹
   sub-distrib-pconj (Embed t)
  ⟨proof⟩

lemma sdp-repeat:
  fixes a::'s prog
  assumes sdpa: sub-distrib-pconj a
    and hwp: healthy (wp a) and hwlp: nearly-healthy (wlp a)
  shows sub-distrib-pconj (repeat n a) (is ?X n)
⟨proof⟩

lemma sdp-SetPC:
  fixes p::'a ⇒ 's prog
  assumes sdp: ⋀s a. a ∈ supp (P s) ⟹ sub-distrib-pconj (p a)
    and fin: ⋀s. finite (supp (P s))
    and nnp: ⋀s a. 0 ≤ P s a
    and sub: ⋀s. sum (P s) (supp (P s)) ≤ 1
  shows sub-distrib-pconj (SetPC p P)
⟨proof⟩

lemma sdp-SetDC:
  fixes p::'a ⇒ 's prog
  assumes sdp: ⋀s a. a ∈ S s ⟹ sub-distrib-pconj (p a)
    and hwp: ⋀s a. a ∈ S s ⟹ healthy (wp (p a))
    and hwlp: ⋀s a. a ∈ S s ⟹ nearly-healthy (wlp (p a))
    and ne:  ⋀s. S s ≠ {}
  shows sub-distrib-pconj (SetDC p S)
⟨proof⟩

lemma sdp-Bind:
  ⟦ ⋀s. sub-distrib-pconj (p (f s)) ⟧ ⟹ sub-distrib-pconj (Bind f p)
```

```
 ⟨proof⟩
```

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

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

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

4.7.3 The Well-Defined Predicate.

```
definition
  well-def :: 's prog ⇒ bool
where
  well-def prog ≡ healthy (wp prog) ∧ nearly-healthy (wlp prog)
                ∧ wp-under-wlp prog ∧ sub-distrib-pconj prog
                ∧ sublinear (wp prog) ∧ bd-cts (wp prog)

lemma well-defI[intro]:
  ⟦ 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 $\Longrightarrow$ *sublinear* (*wp prog*)
⟨*proof*⟩

lemma *well-def-wp-cts*[*dest*]:
well-def prog $\Longrightarrow$ *bd-cts* (*wp prog*)
⟨*proof*⟩

lemmas *wd-dests* =
well-def-wp-healthy well-def-wlp-nearly-healthy
well-def-wp-under well-def-sdp
well-def-wp-sublinear well-def-wp-cts

lemma *wd-Abort*:
well-def Abort
⟨*proof*⟩

lemma *wd-Skip*:
well-def Skip
⟨*proof*⟩

lemma *wd-Apply*:
well-def (*Apply f*)
⟨*proof*⟩

lemma *wd-Seq*:
⟦ *well-def a*; *well-def b* ⟧ $\Longrightarrow$ *well-def* (*a* ;; *b*)
⟨*proof*⟩

lemma *wd-PC*:
⟦ *well-def a*; *well-def b*; *unitary P* ⟧ $\Longrightarrow$ *well-def* ($a\ {}_P\!\oplus\ b$)
⟨*proof*⟩

lemma *wd-DC*:
⟦ *well-def a*; *well-def b* ⟧ $\Longrightarrow$ *well-def* ($a \sqcap b$)
⟨*proof*⟩

lemma *wd-SetDC*:
⟦ $\bigwedge x\ s.\ x \in S\ s \Longrightarrow$ *well-def* (*a x*); $\bigwedge s.\ S\ s \neq \{\}$;
$\bigwedge s.$ *finite* (*S s*) ⟧ $\Longrightarrow$ *well-def* (*SetDC a S*)
⟨*proof*⟩

lemma *wd-SetPC*:
⟦ $\bigwedge x\ s.\ x \in$ (*supp* (*p s*)) $\Longrightarrow$ *well-def* (*a x*); $\bigwedge s.$ *unitary* (*p s*); $\bigwedge s.$ *finite* (*supp* (*p s*));
$\bigwedge s.$ *sum* (*p s*) (*supp* (*p s*)) $\leq$ *1* ⟧ $\Longrightarrow$ *well-def* (*SetPC a p*)
⟨*proof*⟩

lemma *wd-Embed*:
fixes *t*::*'s trans*
assumes *ht*: *healthy t* **and** *st*: *sublinear t* **and** *ct*: *bd-cts t*
shows *well-def* (*Embed t*)
⟨*proof*⟩

lemma *wd-repeat*:
well-def a ⟹ *well-def* (*repeat n a*)
⟨*proof*⟩

lemma *wd-Bind*:
⟦ ⋀*s*. *well-def* (*a* (*f s*)) ⟧ ⟹ *well-def* (*Bind f a*)
⟨*proof*⟩

lemma *wd-loop*:
well-def body ⟹ *well-def* (*do G* ⟶ *body od*)
⟨*proof*⟩

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* ⇒ *bool*) ⇒ *'s prog* ⇒ (*'s* ⇒ *real*) ⇒ *bool*
where
wp-inv G body I ⟷ (∀ *s*. «*G*» *s* ∗ *I s* ≤ *wp body I s*)

lemma *wp-invI*:
⋀*I*. (⋀*s*. «*G*» *s* ∗ *I s* ≤ *wp body I s*) ⟹ *wp-inv G body I*
⟨*proof*⟩

definition
wlp-inv :: (*'s* ⇒ *bool*) ⇒ *'s prog* ⇒ (*'s* ⇒ *real*) ⇒ *bool*

where
$\mathit{wlp\text{-}inv}\ G\ \mathit{body}\ I \longleftrightarrow (\forall\, s.\ «G»\ s * I\ s \leq \mathit{wlp}\ \mathit{body}\ I\ s)$

lemma *wlp-invI*:
$\bigwedge I.\ (\bigwedge s.\ «G»\ s * I\ s \leq \mathit{wlp}\ \mathit{body}\ I\ s) \Longrightarrow \mathit{wlp\text{-}inv}\ G\ \mathit{body}\ I$
⟨*proof*⟩

lemma *wlp-invD*:
$\mathit{wlp\text{-}inv}\ G\ \mathit{body}\ I \Longrightarrow «G»\ s * I\ s \leq \mathit{wlp}\ \mathit{body}\ I\ s$
⟨*proof*⟩

For standard invariants, the multiplication reduces to conjunction.

lemma *wp-inv-stdD*:
assumes *inv*: $\mathit{wp\text{-}inv}\ G\ \mathit{body}\ «I»$
and *hb*: $\mathit{healthy}\ (\mathit{wp}\ \mathit{body})$
shows $«G»\ \&\&\ «I» \Vdash \mathit{wp}\ \mathit{body}\ «I»$
⟨*proof*⟩

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 *uI*: *unitary I*
and *inv*: *wlp-inv G body I*
shows $I \leq \mathit{wlp}\ \mathit{do}\ G \longrightarrow \mathit{body}\ \mathit{od}\ (\lambda s.\ «\mathcal{N}\ G»\ s * I\ s)$
(**is** $I \leq \mathit{wlp}\ \mathit{do}\ G \longrightarrow \mathit{body}\ \mathit{od}\ ?P$)
⟨*proof*⟩

4.8.3 Total Correctness

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

lemma *wp-Loop*:
assumes *wd*: *well-def body*
and *inv*: *wlp-inv G body I*
and *unit*: *unitary I*
shows $I\ \&\&\ \mathit{wp}\ (\mathit{do}\ G \longrightarrow \mathit{body}\ \mathit{od})\ (\lambda s.\ 1) \Vdash \mathit{wp}\ (\mathit{do}\ G \longrightarrow \mathit{body}\ \mathit{od})\ (\lambda s.\ «\mathcal{N}\ G»\ s * I$
$s)$
(**is** $I\ \&\&\ ?T \Vdash \mathit{wp}\ ?\mathit{loop}\ ?X$)
⟨*proof*⟩

4.8.4 Unfolding

lemma *wp-loop-unfold*:
fixes $\mathit{body} :: {'s}\ \mathit{prog}$
assumes *sP*: *sound P*
and *h*: $\mathit{healthy}\ (\mathit{wp}\ \mathit{body})$

shows $wp\ (do\ G \longrightarrow body\ od)\ P =$
$(\lambda s.\ «\mathcal{N}\ G»\ s * P\ s + «G»\ s * wp\ body\ (wp\ (do\ G \longrightarrow body\ od)\ P)\ s)$
$\langle proof \rangle$

lemma *wp-loop-nguard*:
$[\![\ healthy\ (wp\ body);\ sound\ P;\ \neg\ G\ s\ ]\!] \Longrightarrow wp\ do\ G \longrightarrow body\ od\ P\ s = P\ s$
$\langle proof \rangle$

lemma *wp-loop-guard*:
$[\![\ healthy\ (wp\ body);\ sound\ P;\ G\ s\ ]\!] \Longrightarrow$
$wp\ do\ G \longrightarrow body\ od\ P\ s = wp\ (body\ ;;\ do\ G \longrightarrow body\ od)\ P\ s$
$\langle proof \rangle$

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 $a \bigsqcap b$ and $a \bigsqcup b$ as the meet and join operators, respectively. We also take advantage of the algebraic structure to establish a framwork 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** ‹⊑› *70*)
where
$prog \sqsubseteq prog' \equiv \forall P.\ sound\ P \longrightarrow wp\ prog\ P \Vdash wp\ prog'\ P$

lemma *refinesI*[*intro*]:
$[\![\ \bigwedge P.\ sound\ P \Longrightarrow wp\ prog\ P \Vdash wp\ prog'\ P\ ]\!] \Longrightarrow prog \sqsubseteq prog'$
$\langle proof \rangle$

lemma *refinesD*[*dest*]:
$[\![\ prog \sqsubseteq prog';\ sound\ P\ ]\!] \Longrightarrow wp\ prog\ P \Vdash wp\ prog'\ P$
$\langle proof \rangle$

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** ‹≃› *70*)
where

$prog \simeq prog' \equiv \forall P.\ sound\ P \longrightarrow wp\ prog\ P = wp\ prog'\ P$

lemma *pequivI*[*intro*]:
$[\![\bigwedge P.\ sound\ P \Longrightarrow wp\ prog\ P = wp\ prog'\ P\ ]\!] \Longrightarrow prog \simeq prog'$
⟨*proof*⟩

lemma *pequivD*[*dest*,*simp*]:
$[\![\ prog \simeq prog';\ sound\ P\ ]\!] \Longrightarrow wp\ prog\ P = wp\ prog'\ P$
⟨*proof*⟩

lemma *pequiv-equiv-trans*:
$a \simeq b \longleftrightarrow equiv\text{-}trans\ (wp\ a)\ (wp\ b)$
⟨*proof*⟩

4.9.2 Simple Identities

The following identities involve only the primitive operations as defined in Section 4.1.1, and refinement as defined above.

Laws following from the basic arithmetic of the operators seperately

lemma *DC-comm*[*ac-simps*]:
$a \bigsqcap b = b \bigsqcap a$
⟨*proof*⟩

lemma *DC-assoc*[*ac-simps*]:
$a \bigsqcap (b \bigsqcap c) = (a \bigsqcap b) \bigsqcap c$
⟨*proof*⟩

lemma *DC-idem*:
$a \bigsqcap a = a$
⟨*proof*⟩

lemma *AC-comm*[*ac-simps*]:
$a \bigsqcup b = b \bigsqcup a$
⟨*proof*⟩

lemma *AC-assoc*[*ac-simps*]:
$a \bigsqcup (b \bigsqcup c) = (a \bigsqcup b) \bigsqcup c$
⟨*proof*⟩

lemma *AC-idem*:
$a \bigsqcup a = a$
⟨*proof*⟩

lemma *PC-quasi-comm*:
$a\ {}_{p}{\oplus}\ b = b\ {}_{(\lambda s.\ 1 - p\ s)}{\oplus}\ a$
⟨*proof*⟩

lemma *PC-idem*:
$a\ {}_{p}{\oplus}\ a = a$
$\langle \mathit{proof} \rangle$

lemma *Seq-assoc*[*ac-simps*]:
$A\ ;;\ (B\ ;;\ C) = A\ ;;\ B\ ;;\ C$
$\langle \mathit{proof} \rangle$

lemma *Abort-refines*[*intro*]:
$\mathit{well\text{-}def}\ a \Longrightarrow \mathit{Abort} \sqsubseteq a$
$\langle \mathit{proof} \rangle$

Laws relating demonic choice and refinement

lemma *left-refines-DC*:
$(a \bigsqcap b) \sqsubseteq a$
$\langle \mathit{proof} \rangle$

lemma *right-refines-DC*:
$(a \bigsqcap b) \sqsubseteq b$
$\langle \mathit{proof} \rangle$

lemma *DC-refines*:
fixes *a*::*'s prog* **and** *b* **and** *c*
assumes *rab*: $a \sqsubseteq b$ **and** *rac*: $a \sqsubseteq c$
shows $a \sqsubseteq (b \bigsqcap c)$
$\langle \mathit{proof} \rangle$

lemma *DC-mono*:
fixes *a*::*'s prog*
assumes *rab*: $a \sqsubseteq b$ **and** *rcd*: $c \sqsubseteq d$
shows $(a \bigsqcap c) \sqsubseteq (b \bigsqcap d)$
$\langle \mathit{proof} \rangle$

Laws relating angelic choice and refinement

lemma *left-refines-AC*:
$a \sqsubseteq (a \bigsqcup b)$
$\langle \mathit{proof} \rangle$

lemma *right-refines-AC*:
$b \sqsubseteq (a \bigsqcup b)$
$\langle \mathit{proof} \rangle$

lemma *AC-refines*:
fixes *a*::*'s prog* **and** *b* **and** *c*
assumes *rac*: $a \sqsubseteq c$ **and** *rbc*: $b \sqsubseteq c$
shows $(a \bigsqcup b) \sqsubseteq c$
$\langle \mathit{proof} \rangle$

lemma *AC-mono*:
fixes $a{::}'s\ prog$
assumes *rab*: $a \sqsubseteq b$ **and** *rcd*: $c \sqsubseteq d$
shows $(a \bigsqcup c) \sqsubseteq (b \bigsqcup d)$
$\langle proof \rangle$

Laws depending on the arithmetic of $a\ {}_{p}{\oplus}\ b$ and $a \bigsqcap b$ together

lemma *DC-refines-PC*:
assumes *unit*: *unitary p*
shows $(a \bigsqcap b) \sqsubseteq (a\ {}_{p}{\oplus}\ b)$
$\langle proof \rangle$

Laws depending on the arithmetic of $a\ {}_{p}{\oplus}\ b$ and $a \bigsqcup b$ together

lemma *PC-refines-AC*:
assumes *unit*: *unitary p*
shows $(a\ {}_{p}{\oplus}\ b) \sqsubseteq (a \bigsqcup b)$
$\langle proof \rangle$

Laws depending on the arithmetic of $a \bigsqcup b$ and $a \bigsqcap b$ together

lemma *DC-refines-AC*:
$(a \bigsqcap b) \sqsubseteq (a \bigsqcup b)$
$\langle proof \rangle$

Laws Involving Refinement and Equivalence

lemma *pr-trans*[*trans*]:
fixes $A{::}'a\ prog$
assumes *prAB*: $A \sqsubseteq B$
and *prBC*: $B \sqsubseteq C$
shows $A \sqsubseteq C$
$\langle proof \rangle$

lemma *pequiv-refl*[*intro*!,*simp*]:
$a \simeq a$
$\langle proof \rangle$

lemma *pequiv-comm*[*ac-simps*]:
$a \simeq b \longleftrightarrow b \simeq a$
$\langle proof \rangle$

lemma *pequiv-pr*[*dest*]:
$a \simeq b \Longrightarrow a \sqsubseteq b$
$\langle proof \rangle$

lemma *pequiv-trans*[*intro*,*trans*]:
$[\![\ a \simeq b;\ b \simeq c\ ]\!] \Longrightarrow a \simeq c$
$\langle proof \rangle$

lemma *pequiv-pr-trans*[*intro*,*trans*]:
$[\![\, a \simeq b;\ b \sqsubseteq c \,]\!] \Longrightarrow a \sqsubseteq c$
⟨*proof*⟩

lemma *pr-pequiv-trans*[*intro*,*trans*]:
$[\![\, a \sqsubseteq b;\ b \simeq c \,]\!] \Longrightarrow a \sqsubseteq c$
⟨*proof*⟩

Refinement induces equivalence by antisymmetry:

lemma *pequiv-antisym*:
$[\![\, a \sqsubseteq b;\ b \sqsubseteq a \,]\!] \Longrightarrow a \simeq b$
⟨*proof*⟩

lemma *pequiv-DC*:
$[\![\, a \simeq c;\ b \simeq d \,]\!] \Longrightarrow (a \bigsqcap b) \simeq (c \bigsqcap d)$
⟨*proof*⟩

lemma *pequiv-AC*:
$[\![\, a \simeq c;\ b \simeq d \,]\!] \Longrightarrow (a \bigsqcup b) \simeq (c \bigsqcup d)$
⟨*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 $a \bigsqcap b$ 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* : $\lambda a\ b.\ a \simeq b \wedge \textit{well-def}\ a \wedge \textit{well-def}\ b$

$\langle proof \rangle$

instantiation *program* :: (*type*) *semilattice-inf* **begin**
lift-definition
  less-eq-program :: *'a program* ⇒ *'a program* ⇒ *bool* **is** *refines*
$\langle proof \rangle$

lift-definition
  less-program :: *'a program* ⇒ *'a program* ⇒ *bool*
  is $\lambda a\ b.\ a \sqsubseteq b \wedge \neg\ b \sqsubseteq a$
$\langle proof \rangle$

lift-definition
  inf-program :: *'a program* ⇒ *'a program* ⇒ *'a program* **is** *DC*
$\langle proof \rangle$

instance
$\langle proof \rangle$
end

instantiation *program* :: (*type*) *bot* **begin**
lift-definition
  bot-program :: *'a program* **is** *Abort*
  $\langle proof \rangle$

instance $\langle proof \rangle$
end

lemma *eq-det*: $\bigwedge a\ b{::}'s\ prog.\ [\![\ a \simeq b;\ determ\ (wp\ a)\ ]\!] \Longrightarrow determ\ (wp\ b)$
$\langle proof \rangle$

lift-definition
  pdeterm :: *'s program* ⇒ *bool*
  is $\lambda a.\ determ\ (wp\ a)$
$\langle proof \rangle$

lemma *determ-maximal*:
  $[\![\ pdeterm\ a;\ a \leq x\ ]\!] \Longrightarrow a = x$
  $\langle proof \rangle$

4.9.5 Data Refinement

A projective data refinement construction for pGCL. By projective, we mean that the abstract state is always a function (φ) 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 φ G A B ≡ ∀ P Q. (unitary P ∧ unitary Q ∧ (P ⊩ wp A Q)) ⟶
                 («G» && (P o φ) ⊩ wp B (Q o φ))
```

```
lemma drefinesD[dest]:
 ⟦ drefines φ G A B; unitary P; unitary Q; P ⊩ wp A Q ⟧ ⟹
  «G» && (P o φ) ⊩ wp B (Q o φ)
 ⟨proof⟩
```

We can alternatively use G as an assumption:

```
lemma drefinesD2:
 assumes dr: drefines φ G A B
   and uP: unitary P
   and uQ: unitary Q
   and wpA: P ⊩ wp A Q
   and G: G s
 shows (P o φ) s ≤ wp B (Q o φ) s
⟨proof⟩
```

This additional form is sometimes useful:

```
lemma drefinesD3:
 assumes dr: drefines φ G a b
   and G: G s
   and uQ: unitary Q
   and wa: well-def a
 shows wp a Q (φ s) ≤ wp b (Q o φ) s
⟨proof⟩
```

```
lemma drefinesI[intro]:
 ⟦ ⋀P Q. ⟦ unitary P; unitary Q; P ⊩ wp A Q ⟧ ⟹
      «G» && (P o φ) ⊩ wp B (Q o φ) ⟧ ⟹
  drefines φ G A B
 ⟨proof⟩
```

Use G as an assumption, when showing refinement:

```
lemma drefinesI2:
 fixes A::'a prog
  and B::'b prog
  and φ::'b ⇒ 'a
  and G::'b ⇒ bool
 assumes wB: well-def B
   and withAs:
    ⋀P Q s. ⟦ unitary P; unitary Q;
         G s; P ⊩ wp A Q ⟧ ⟹ (P o φ) s ≤ wp B (Q o φ) s
 shows drefines φ G A B
⟨proof⟩
```

```
lemma dr-strengthen-guard:
 fixes a::'s prog and b::'t prog
 assumes fg: ⋀s. F s ⟹ G s
```

  and *drab*: *drefines* φ *G a b*
 shows *drefines* φ *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 φ *G A B* $\longleftrightarrow$
  $(\forall\, Q.$ *unitary Q* $\longrightarrow$ «*G*» && (*wp A Q o* φ) = «*G*» && *wp B* (*Q o* φ))

lemma *pcorresI*:
 $[\![\, \bigwedge Q.$ *unitary Q* $\Longrightarrow$ «*G*» && (*wp A Q o* φ) = «*G*» && *wp B* (*Q o* φ) $]\!] \Longrightarrow$
  pcorres φ *G A B*
 $\langle$*proof*$\rangle$

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

lemma *pcorresI2*[*intro*]:
 fixes *A*::*'a prog* **and** *B*::*'b prog*
 assumes *withG*: $\bigwedge Q\ s.$ $[\![$ *unitary Q*; *G s* $]\!] \Longrightarrow$ *wp A Q* (φ *s*)= *wp B* (*Q o* φ) *s*
  and *wA*: *well-def A*
  and *wB*: *well-def B*
 shows *pcorres* φ *G A B*
$\langle$*proof*$\rangle$

lemma *pcorresD*:
 $[\![$ *pcorres* φ *G A B*; *unitary Q* $]\!] \Longrightarrow$ «*G*» && (*wp A Q o* φ) = «*G*» && *wp B* (*Q o* φ)
 $\langle$*proof*$\rangle$

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

lemma *pcorresD2*:
 assumes *pc*: *pcorres* φ *G A B*
  and *uQ*: *unitary Q*
  and *wA*: *well-def A* **and** *wB*: *well-def B*
  and *G*: *G s*
 shows *wp A Q* (φ *s*) = *wp B* (*Q o* φ) *s*
$\langle$*proof*$\rangle$

4.9.6 The Algebra of Data Refinement

Program refinement implies a trivial data refinement:

lemma *refines-drefines*:
 fixes *a*::*'s prog*
 assumes *rab*: $a \sqsubseteq b$ **and** *wb*: *well-def b*
 shows *drefines* $(\lambda s.\ s)$ *G a b*
$\langle$*proof*$\rangle$

Data refinement is transitive:

lemma *dr-trans*[*trans*]:
fixes *A*::*'a prog* **and** *B*::*'b prog* **and** *C*::*'c prog*
assumes *drAB*: *drefines* φ *G A B*
and *drBC*: *drefines* φ' G' *B C*
and *Gimp*: $\bigwedge s.\ G'\ s \Longrightarrow G\ (\varphi'\ s)$
shows *drefines* $(\varphi\ o\ \varphi')$ G' *A C*
$\langle proof \rangle$

Data refinement composes with program refinement:

lemma *pr-dr-trans*[*trans*]:
assumes *prAB*: $A \sqsubseteq B$
and *drBC*: *drefines* φ *G B C*
shows *drefines* φ *G A C*
$\langle proof \rangle$

lemma *dr-pr-trans*[*trans*]:
assumes *drAB*: *drefines* φ *G A B*
assumes *prBC*: $B \sqsubseteq C$
shows *drefines* φ *G A C*
$\langle proof \rangle$

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

lemma *dr-refl*:
assumes *wa*: *well-def a*
and *comm*: $\bigwedge Q.\ unitary\ Q \Longrightarrow wp\ a\ Q\ o\ \varphi \Vdash wp\ a\ (Q\ o\ \varphi)$
shows *drefines* φ *G a a*
$\langle proof \rangle$

Correspondence implies data refinement

lemma *pcorres-drefine*:
assumes *corres*: *pcorres* φ *G A C*
and *wC*: *well-def C*
shows *drefines* φ *G A C*
$\langle proof \rangle$

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* φ *G a b*
shows *pcorres* φ *G a b*

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

⟨*proof*⟩

4.9.7 Structural Rules for Correspondence

lemma *pcorres-Skip*:
  pcorres φ *G Skip Skip*
  ⟨*proof*⟩

Correspondence composes over sequential composition.

lemma *pcorres-Seq*:
  fixes *A*::*'b prog* **and** *B*::*'c prog*
   and *C*::*'b prog* **and** *D*::*'c prog*
   and φ::*'c* ⇒ *'b*
  assumes *pcAB*: *pcorres* φ *G A B*
    and *pcCD*: *pcorres* φ *H C D*
    and *wA*: *well-def A* **and** *wB*: *well-def B*
    and *wC*: *well-def C* **and** *wD*: *well-def D*
    and *p3p2*: $\bigwedge Q.$ *unitary Q* $\Longrightarrow$ «*I*» && *wp B Q* = *wp B* («*H*» && *Q*)
    and *p1p3*: $\bigwedge s.$ *G s* $\Longrightarrow$ *I s*
  shows *pcorres* φ *G* (*A*;;*C*) (*B*;;*D*)
⟨*proof*⟩

4.9.8 Structural Rules for Data Refinement

lemma *dr-Skip*:
  fixes φ::*'c* ⇒ *'b*
  shows *drefines* φ *G Skip Skip*
⟨*proof*⟩

lemma *dr-Abort*:
  fixes φ::*'c* ⇒ *'b*
  shows *drefines* φ *G Abort Abort*
⟨*proof*⟩

lemma *dr-Apply*:
  fixes φ::*'c* ⇒ *'b*
  assumes *commutes*: *f o* φ = φ *o g*
  shows *drefines* φ *G* (*Apply f*) (*Apply g*)
⟨*proof*⟩

lemma *dr-Seq*:
  assumes *drAB*: *drefines* φ *P A B*
    and *drBC*: *drefines* φ *Q C D*
    and *wpB*: «*P*» ⊩ *wp B* «*Q*»
    and *wB*: *well-def B*
    and *wC*: *well-def C*

  and *wD*: *well-def D*
 shows *drefines* φ *P* (*A*;;*C*) (*B*;;*D*)
$\langle$*proof*$\rangle$

lemma *dr-repeat*:
 fixes φ :: *'a* ⇒ *'b*
 assumes *dr-ab*: *drefines* φ *G a b*
  and *Gpr*: «*G*» $\Vdash$ *wp b* «*G*»
  and *wa*: *well-def a*
  and *wb*: *well-def b*
 shows *drefines* φ *G* (*repeat n a*) (*repeat n b*) (**is** *?X n*)
$\langle$*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*:
 (λ*s. 0*) $\Vdash$ *wp Abort Q*
 $\langle$*proof*$\rangle$

lemma *wlp-Abort*:
 (λ*s. 1*) $\Vdash$ *wlp Abort Q*
 $\langle$*proof*$\rangle$

lemma *wp-Skip*:
 P $\Vdash$ *wp Skip P*
 $\langle$*proof*$\rangle$

lemma *wlp-Skip*:
 P $\Vdash$ *wlp Skip P*
 $\langle$*proof*$\rangle$

lemma *wp-Apply*:
 Q o f $\Vdash$ *wp* (*Apply f*) *Q*
 $\langle$*proof*$\rangle$

lemma *wlp-Apply*:
 Q o f $\Vdash$ *wlp* (*Apply f*) *Q*

⟨*proof*⟩

lemma *wp-Seq*:
assumes *ent-a*: $P \Vdash wp\ a\ Q$
and *ent-b*: $Q \Vdash wp\ b\ R$
and *wa*: *well-def a*
and *wb*: *well-def b*
and *s-Q*: *sound Q*
and *s-R*: *sound R*
shows $P \Vdash wp\ (a\ ;;\ b)\ R$
⟨*proof*⟩

lemma *wlp-Seq*:
assumes *ent-a*: $P \Vdash wlp\ a\ Q$
and *ent-b*: $Q \Vdash wlp\ b\ R$
and *wa*: *well-def a*
and *wb*: *well-def b*
and *u-Q*: *unitary Q*
and *u-R*: *unitary R*
shows $P \Vdash wlp\ (a\ ;;\ b)\ R$
⟨*proof*⟩

lemma *wp-PC*:
$(\lambda s.\ P\ s * wp\ a\ Q\ s + (1 - P\ s) * wp\ b\ Q\ s) \Vdash wp\ (a\ {}_P{\oplus}\ b)\ Q$
⟨*proof*⟩

lemma *wlp-PC*:
$(\lambda s.\ P\ s * wlp\ a\ Q\ s + (1 - P\ s) * wlp\ b\ Q\ s) \Vdash wlp\ (a\ {}_P{\oplus}\ b)\ Q$
⟨*proof*⟩

A simpler rule for when the probability does not depend on the state.

lemma *PC-fixed*:
assumes *wpa*: $P \Vdash a\ ab\ R$
and *wpb*: $Q \Vdash b\ ab\ R$
and *np*: $0 \le p$ **and** *bp*: $p \le 1$
shows $(\lambda s.\ p * P\ s + (1 - p) * Q\ s) \Vdash (a\ {}_{(\lambda s.\ p)}{\oplus}\ b)\ ab\ R$
⟨*proof*⟩

lemma *wp-PC-fixed*:
$[\![\, P \Vdash wp\ a\ R;\ Q \Vdash wp\ b\ R;\ 0 \le p;\ p \le 1 \,]\!] \Longrightarrow$
$(\lambda s.\ p * P\ s + (1 - p) * Q\ s) \Vdash wp\ (a\ {}_{(\lambda s.\ p)}{\oplus}\ b)\ R$
⟨*proof*⟩

lemma *wlp-PC-fixed*:
$[\![\, P \Vdash wlp\ a\ R;\ Q \Vdash wlp\ b\ R;\ 0 \le p;\ p \le 1 \,]\!] \Longrightarrow$
$(\lambda s.\ p * P\ s + (1 - p) * Q\ s) \Vdash wlp\ (a\ {}_{(\lambda s.\ p)}{\oplus}\ b)\ R$
⟨*proof*⟩

lemma *wp-DC*:

$(\lambda s.\ min\ (wp\ a\ Q\ s)\ (wp\ b\ Q\ s)) \Vdash wp\ (a \bigsqcap b)\ Q$
$\langle proof \rangle$

lemma *wlp-DC*:
$(\lambda s.\ min\ (wlp\ a\ Q\ s)\ (wlp\ b\ Q\ s)) \Vdash wlp\ (a \bigsqcap b)\ Q$
$\langle proof \rangle$

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 $(\lambda s.\ min\ (P\ s)\ (Q\ s)) \Vdash (a \bigsqcap b)\ ab\ R$
$\langle proof \rangle$

lemma *wp-DC-split*:
$\llbracket P \Vdash wp\ prog\ R;\ Q \Vdash wp\ prog'\ R \rrbracket \Longrightarrow$
$(\lambda s.\ min\ (P\ s)\ (Q\ s)) \Vdash wp\ (prog \bigsqcap prog')\ R$
$\langle proof \rangle$

lemma *wlp-DC-split*:
$\llbracket P \Vdash wlp\ prog\ R;\ Q \Vdash wlp\ prog'\ R \rrbracket \Longrightarrow$
$(\lambda s.\ min\ (P\ s)\ (Q\ s)) \Vdash wlp\ (prog \bigsqcap prog')\ R$
$\langle proof \rangle$

lemma *wp-DC-split-same*:
$\llbracket P \Vdash wp\ prog\ Q;\ P \Vdash wp\ prog'\ Q \rrbracket \Longrightarrow P \Vdash wp\ (prog \bigsqcap prog')\ Q$
$\langle proof \rangle$

lemma *wlp-DC-split-same*:
$\llbracket P \Vdash wlp\ prog\ Q;\ P \Vdash wlp\ prog'\ Q \rrbracket \Longrightarrow P \Vdash wlp\ (prog \bigsqcap prog')\ Q$
$\langle proof \rangle$

lemma *SetPC-split*:
fixes $f{::}'x \Rightarrow\ 'y\ prog$
and $p{::}'y \Rightarrow\ 'x \Rightarrow real$
assumes rec: $\bigwedge x\ s.\ x \in supp\ (p\ s) \Longrightarrow P\ x \Vdash f\ x\ ab\ Q$
and nnp: $\bigwedge s.\ nneg\ (p\ s)$
shows $(\lambda s.\ \sum x \in supp\ (p\ s).\ p\ s\ x * P\ x\ s) \Vdash SetPC\ f\ p\ ab\ Q$
$\langle proof \rangle$

lemma *wp-SetPC-split*:
$\llbracket \bigwedge x\ s.\ x \in supp\ (p\ s) \Longrightarrow P\ x \Vdash wp\ (f\ x)\ Q;\ \bigwedge s.\ nneg\ (p\ s) \rrbracket \Longrightarrow$
$(\lambda s.\ \sum x \in supp\ (p\ s).\ p\ s\ x * P\ x\ s) \Vdash wp\ (SetPC\ f\ p)\ Q$
$\langle proof \rangle$

lemma *wlp-SetPC-split*:
$\llbracket \bigwedge x\ s.\ x \in supp\ (p\ s) \Longrightarrow P\ x \Vdash wlp\ (f\ x)\ Q;\ \bigwedge s.\ nneg\ (p\ s) \rrbracket \Longrightarrow$
$(\lambda s.\ \sum x \in supp\ (p\ s).\ p\ s\ x * P\ x\ s) \Vdash wlp\ (SetPC\ f\ p)\ Q$

$\langle \mathit{proof} \rangle$

lemma *wp-SetDC-split*:
$[\![\ \bigwedge s\ x.\ x \in S\ s \Longrightarrow P \Vdash wp\ (f\ x)\ Q;\ \bigwedge s.\ S\ s \neq \{\}\ ]\!] \Longrightarrow$
$P \Vdash wp\ (SetDC\ f\ S)\ Q$
$\langle \mathit{proof} \rangle$

lemma *wlp-SetDC-split*:
$[\![\ \bigwedge s\ x.\ x \in S\ s \Longrightarrow P \Vdash wlp\ (f\ x)\ Q;\ \bigwedge s.\ S\ s \neq \{\}\ ]\!] \Longrightarrow$
$P \Vdash wlp\ (SetDC\ f\ S)\ Q$
$\langle \mathit{proof} \rangle$

lemma *wp-SetDC*:
assumes *wp*: $\bigwedge s\ x.\ x \in S\ s \Longrightarrow P\ x \Vdash wp\ (f\ x)\ Q$
and *ne*: $\bigwedge s.\ S\ s \neq \{\}$
and *sP*: $\bigwedge x.\ sound\ (P\ x)$
shows $(\lambda s.\ Inf\ ((\lambda x.\ P\ x\ s)\ \text{‘}\ S\ s)) \Vdash wp\ (SetDC\ f\ S)\ Q$
$\langle \mathit{proof} \rangle$

lemma *wlp-SetDC*:
assumes *wp*: $\bigwedge s\ x.\ x \in S\ s \Longrightarrow P\ x \Vdash wlp\ (f\ x)\ Q$
and *ne*: $\bigwedge s.\ S\ s \neq \{\}$
and *sP*: $\bigwedge x.\ sound\ (P\ x)$
shows $(\lambda s.\ Inf\ ((\lambda x.\ P\ x\ s)\ \text{‘}\ S\ s)) \Vdash wlp\ (SetDC\ f\ S)\ Q$
$\langle \mathit{proof} \rangle$

lemma *wp-Embed*:
$P \Vdash t\ Q \Longrightarrow P \Vdash wp\ (Embed\ t)\ Q$
$\langle \mathit{proof} \rangle$

lemma *wlp-Embed*:
$P \Vdash t\ Q \Longrightarrow P \Vdash wlp\ (Embed\ t)\ Q$
$\langle \mathit{proof} \rangle$

lemma *wp-Bind*:
$[\![\ \bigwedge s.\ P\ s \leq wp\ (a\ (f\ s))\ Q\ s\ ]\!] \Longrightarrow P \Vdash wp\ (Bind\ f\ a)\ Q$
$\langle \mathit{proof} \rangle$

lemma *wlp-Bind*:
$[\![\ \bigwedge s.\ P\ s \leq wlp\ (a\ (f\ s))\ Q\ s\ ]\!] \Longrightarrow P \Vdash wlp\ (Bind\ f\ a)\ Q$
$\langle \mathit{proof} \rangle$

lemma *wp-repeat*:
$[\![\ P \Vdash wp\ a\ Q;\ Q \Vdash wp\ (repeat\ n\ a)\ R;$
$\text{well-def}\ a;\ sound\ Q;\ sound\ R\ ]\!] \Longrightarrow P \Vdash wp\ (repeat\ (Suc\ n)\ a)\ R$
$\langle \mathit{proof} \rangle$

lemma *wlp-repeat*:
$[\![\ P \Vdash wlp\ a\ Q;\ Q \Vdash wlp\ (repeat\ n\ a)\ R;$

well-def a; *unitary Q*; *unitary R* ⟧ ⟹ *P* ⊩ *wlp* (*repeat* (*Suc n*) *a*) *R*
⟨*proof*⟩

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 ⊩ *wlp a Q* ⟹ *nearly-healthy* (*wlp a*) ⟹ *unitary R* ⟹ *Q* ⊩ *R* ⟹ *unitary Q* ⟹
P ⊩ *wlp a R*
⟨*proof*⟩

lemmas *wp-weaken-pre*=
entails-weaken-pre[**where** *t*=*wp a* **for** *a*]
lemmas *wlp-weaken-pre*=
entails-weaken-pre[**where** *t*=*wlp a* **for** *a*]

lemmas *wp-scale*=
entails-scale[**where** *t*=*wp a* **for** *a*, *OF* - *well-def-wp-healthy*]

4.10.2 Algebraic Decomposition

Refinement is a powerful tool for decomposition, belied by the simplicity of the rule. This is an *axiomatic* formulation of refinement (all annotations of the *a* are annotations of *b*), rather than an operational version (all traces of *b* are traces of *a*.

lemma *wp-refines*:
⟦ *a* ⊑ *b*; *P* ⊩ *wp a Q*; *sound Q* ⟧ ⟹ *P* ⊩ *wp b Q*
⟨*proof*⟩

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-valid :: (*'a* ⇒ *real*) ⇒ *'a prog* ⇒ (*'a* ⇒ *real*) ⇒ *bool* (‹⦃-⦄ - ⦃-⦄*p*›)
where
wp-valid P prog Q ≡ *P* ⊩ *wp prog Q*

lemma *wp-validI*:
P ⊩ *wp prog Q* ⟹ ⦃*P*⦄ *prog* ⦃*Q*⦄*p*

```
  ⟨proof⟩

lemma wp-validD:
  ⦃P⦄ prog ⦃Q⦄p ⟹ P ⊩ wp prog Q
  ⟨proof⟩

lemma valid-Seq:
  ⟦ ⦃P⦄ a ⦃Q⦄p; ⦃Q⦄ b ⦃R⦄p; well-def a; well-def b; sound Q; sound R ⟧ ⟹
  ⦃P⦄ a ;; b ⦃R⦄p
  ⟨proof⟩
```

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 prog and s::'s prog
  assumes mr: maximal (wp r)
    and ws: well-def s
    and ts: (λs. 1) ⊩ wp s (λs. 1)
  shows (λs. 1) ⊩ wp (r ;; s) (λs. 1)
⟨proof⟩
```

From any state where the guard does not hold, a loop terminates in a single step.

```
lemma term-onestep:
  assumes wb: well-def body
  shows «𝒩 G» ⊩ wp do G ⟶ body od (λs. 1)
⟨proof⟩
```

4.11.2 Classical Termination

The first non-trivial termination result is quite standard: If we can provide a natural-number-valued measure, that decreases on every iteration, and implies termination on reaching zero, the loop terminates.

lemma *loop-term-nat-measure-noinv*:
fixes $m :: {}'s \Rightarrow nat$ **and** $body :: {}'s\ prog$
assumes *wb*: *well-def body*
and *guard*: $\bigwedge s.\ m\ s = 0 \longrightarrow \neg\ G\ s$
and *variant*: $\bigwedge n.\ «\lambda s.\ m\ s = Suc\ n» \Vdash wp\ body\ «\lambda s.\ m\ s = n»$
shows $\lambda s.\ 1 \Vdash wp\ do\ G \longrightarrow body\ od\ (\lambda s.\ 1)$
$\langle proof \rangle$

This version allows progress to depend on an invariant. Termination is then determined by the invariant's value in the initial state.

lemma *loop-term-nat-measure*:
fixes $m :: {}'s \Rightarrow nat$ **and** $body :: {}'s\ prog$
assumes *wb*: *well-def body*
and *guard*: $\bigwedge s.\ m\ s = 0 \longrightarrow \neg\ G\ s$
and *variant*: $\bigwedge n.\ «\lambda s.\ m\ s = Suc\ n» \ \&\&\ «I» \Vdash wp\ body\ «\lambda s.\ m\ s = n»$
and *inv*: *wp-inv G body «I»*
shows $«I» \Vdash wp\ do\ G \longrightarrow body\ od\ (\lambda s.\ 1)$
$\langle proof \rangle$

4.11.3 Probabilistic Termination

Any loop that has a non-zero chance of terminating after each step terminates with probability 1.

lemma *termination-0-1*:
fixes $body :: {}'s\ prog$
assumes *wb*: *well-def body*
— The loop terminates in one step with nonzero probability
and *onestep*: $(\lambda s.\ p) \Vdash wp\ body\ «\mathcal{N}\ G»$
and *nzp*: $0 < p$
— The body is maximal i.e. it terminates absolutely.
and *mb*: $maximal\ (wp\ body)$
shows $\lambda s.\ 1 \Vdash wp\ do\ G \longrightarrow body\ od\ (\lambda s.\ 1)$
$\langle proof \rangle$

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 *wd*
theorems to automatically establish well−definedness
named-theorems *pwp-core*
core probabilistic wp rules, for evaluating primitive terms

named-theorems *pwp*
user−supplied probabilistic wp rules
named-theorems *pwlp*
user−supplied probabilistic wlp rules

⟨*ML*⟩

declare *wd-intros*[*wd*]

lemmas *core-wp-rules* =
wp-Skip wlp-Skip
wp-Abort wlp-Abort
wp-Apply wlp-Apply
wp-Seq wlp-Seq
wp-DC-split wlp-DC-split
wp-PC-fixed wlp-PC-fixed
wp-SetDC wlp-SetDC
wp-SetPC-split wlp-SetPC-split

declare *core-wp-rules*[*pwp-core*]

end

Additional Material

4.13 Miscellaneous Mathematics

```
theory Misc
imports
  HOL−Analysis.Multivariate-Analysis
begin lemma sum-UNIV:
  fixes S::'a::finite set
  assumes complete: ⋀x. x∉S ⟹ f x = 0
  shows sum f S = sum f UNIV
⟨proof⟩

lemma cInf-mono:
  fixes A::'a::conditionally-complete-lattice set
  assumes lower: ⋀b. b ∈ B ⟹ ∃ a∈A. a ≤ b
    and bounded: ⋀a. a ∈ A ⟹ c ≤ a
    and ne: B ≠ {}
  shows Inf A ≤ Inf B
⟨proof⟩

lemma max-distrib:
  fixes c::real
  assumes nn: 0 ≤ c
  shows c ∗ max a b = max (c ∗ a) (c ∗ b)
⟨proof⟩

lemma mult-div-mono-left:
  fixes c::real
  assumes nnc: 0 ≤ c and nzc: c ≠ 0
    and inv: a ≤ inverse c ∗ b
  shows c ∗ a ≤ b
⟨proof⟩

lemma mult-div-mono-right:
  fixes c::real
  assumes nnc: 0 ≤ c and nzc: c ≠ 0
    and inv: inverse c ∗ a ≤ b
  shows a ≤ c ∗ b
⟨proof⟩
```

```
lemma min-distrib:
  fixes c::real
  assumes nnc: 0 ≤ c
  shows c ∗ min a b = min (c ∗ a) (c ∗ b)
⟨proof⟩

lemma finite-set-least:
  fixes S::'a::linorder set
  assumes finite: finite S
    and ne: S ≠ {}
  shows ∃x∈S. ∀y∈S. x ≤ y
⟨proof⟩

lemma cSup-add:
  fixes c::real
  assumes ne: S ≠ {}
    and bS: ⋀x. x∈S ⟹ x ≤ b
  shows Sup S + c = Sup {x + c |x. x ∈ S}
⟨proof⟩

lemma cSup-mult:
  fixes c::real
  assumes ne: S ≠ {}
    and bS: ⋀x. x∈S ⟹ x ≤ b
    and nnc: 0 ≤ c
  shows c ∗ Sup S = Sup {c ∗ x |x. x ∈ S}
⟨proof⟩

lemma closure-contains-Sup:
  fixes S :: real set
  assumes neS: S ≠ {} and bS: ∀x∈S. x ≤ B
  shows Sup S ∈ closure S
⟨proof⟩

lemma tendsto-min:
  fixes x y::real
  assumes ta: a ⟶ x
    and tb: b ⟶ y
  shows (λi. min (a i) (b i)) ⟶ min x y
⟨proof⟩

definition supp :: ('s ⇒ real) ⇒ 's set
where supp f = {x. f x ≠ 0}

definition dist-remove :: ('s ⇒ real) ⇒ 's ⇒ 's ⇒ real
where dist-remove p x = (λy. if y=x then 0 else p y / (1 − p x))

lemma supp-dist-remove:
```

$p\ x \neq 0 \Longrightarrow p\ x \neq 1 \Longrightarrow \mathit{supp}\ (\mathit{dist\text{-}remove}\ p\ x) = \mathit{supp}\ p - \{x\}$
$\langle \mathit{proof} \rangle$

lemma *supp-empty*:
$\mathit{supp}\ f = \{\} \Longrightarrow f\ x = 0$
$\langle \mathit{proof} \rangle$

lemma *nsupp-zero*:
$x \notin \mathit{supp}\ f \Longrightarrow f\ x = 0$
$\langle \mathit{proof} \rangle$

lemma *sum-supp*:
fixes $f{::}'a{::}\mathit{finite} \Rightarrow \mathit{real}$
shows $\mathit{sum}\ f\ (\mathit{supp}\ f) = \mathit{sum}\ f\ \mathit{UNIV}$
$\langle \mathit{proof} \rangle$

4.13.1 Truncated Subtraction

definition
$\mathit{tminus} :: \mathit{real} \Rightarrow \mathit{real} \Rightarrow \mathit{real}$ (**infixl** $\langle\ominus\rangle$ *60*)
where
$x \ominus y = \mathit{max}\ (x - y)\ 0$

lemma *minus-le-tminus*[*intro*!,*simp*]:
$a - b \leq a \ominus b$
$\langle \mathit{proof} \rangle$

lemma *tminus-cancel-1*:
$0 \leq a \Longrightarrow a + 1 \ominus 1 = a$
$\langle \mathit{proof} \rangle$

lemma *tminus-zero-imp-le*:
$x \ominus y \leq 0 \Longrightarrow x \leq y$
$\langle \mathit{proof} \rangle$

lemma *tminus-zero*[*simp*]:
$0 \leq x \Longrightarrow x \ominus 0 = x$
$\langle \mathit{proof} \rangle$

lemma *tminus-left-mono*:
$a \leq b \Longrightarrow a \ominus c \leq b \ominus c$
$\langle \mathit{proof} \rangle$

lemma *tminus-less*:
$[\![\, 0 \leq a;\ 0 \leq b \,]\!] \Longrightarrow a \ominus b \leq a$
$\langle \mathit{proof} \rangle$

lemma *tminus-left-distrib*:
assumes *nna*: $0 \leq a$

shows $a * (b \ominus c) = a * b \ominus a * c$
⟨*proof*⟩

lemma *tminus-le*[*simp*]:
$b \leq a \Longrightarrow a \ominus b = a - b$
⟨*proof*⟩

lemma *tminus-le-alt*[*simp*]:
$a \leq b \Longrightarrow a \ominus b = 0$
⟨*proof*⟩

lemma *tminus-nle*[*simp*]:
$\neg b \leq a \Longrightarrow a \ominus b = 0$
⟨*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 $sum\, f\, S \ominus sum\, g\, S \leq sum\ (\lambda x.\, f\, x \ominus g\, x)\ S$
(**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.