pGCL for Isabelle

David Cock

October 26, 2026

Contents

1 Overview **1**

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

3 Semantic Structures **13**
3.1 Expectations . . . 13
3.1.1 Bounded Functions . . . 14
3.1.2 Non-Negative Functions. . . . 16
3.1.3 Sound Expectations . . . 17
3.1.4 Unitary expectations . . . 19
3.1.5 Standard Expectations . . . 19
3.1.6 Entailment . . . 21
3.1.7 Expectation Conjunction . . . 22
3.1.8 Rules Involving Conjunction. . . . 23
3.1.9 Rules Involving Entailment and Conjunction Together . . . 25
3.2 Expectation Transformers . . . 26
3.2.1 Comparing Transformers . . . 28
3.2.2 Healthy Transformers . . . 32
3.2.3 Sublinearity . . . 37

3.2.4 Determinism . . . 41
3.2.5 Modular Reasoning . . . 44
3.2.6 Transforming Standard Expectations . . . 44
3.3 Induction . . . 45
3.3.1 The Lattice of Expectations . . . 45
3.3.2 The Lattice of Transformers . . . 48
3.3.3 Tail Recursion . . . 51

4 The pGCL Language **53**
4.1 A Shallow Embedding of pGCL in HOL . . . 53
4.1.1 Core Primitives and Syntax . . . 53
4.1.2 Unfolding rules for non-recursive primitives . . . 56
4.2 Healthiness . . . 59
4.2.1 The Healthiness of the Embedding . . . 59
4.2.2 Healthiness for Loops . . . 62
4.3 Continuity . . . 64
4.3.1 Continuity of Primitives . . . 65
4.3.2 Continuity of a Single Loop Step . . . 68
4.4 Continuity and Induction for Loops . . . 68
4.4.1 The Limit of Iterates . . . 69
4.5 Sublinearity . . . 71
4.5.1 Nonrecursive Primitives . . . 71
4.5.2 Sublinearity for Loops . . . 73
4.6 Determinism . . . 74
4.6.1 Additivity . . . 74
4.6.2 Maximality . . . 75
4.6.3 Determinism . . . 76
4.7 Well-Defined Programs. . . . 77
4.7.1 Strict Implies Liberal . . . 78
4.7.2 Sub-Distributivity of Conjunction . . . 80
4.7.3 The Well-Defined Predicate. . . . 82
4.8 The Loop Rules . . . 84
4.8.1 Liberal and Strict Invariants. . . . 84
4.8.2 Partial Correctness . . . 85
4.8.3 Total Correctness . . . 85
4.8.4 Unfolding . . . 85
4.9 The Algebra of pGCL . . . 86
4.9.1 Program Refinement . . . 86
4.9.2 Simple Identities . . . 87
4.9.3 Deterministic Programs are Maximal . . . 90
4.9.4 The Algebraic Structure of Refinement . . . 90
4.9.5 Data Refinement . . . 91
4.9.6 The Algebra of Data Refinement . . . 93
4.9.7 Structural Rules for Correspondence . . . 95

4.9.8 Structural Rules for Data Refinement 95
4.10 Structured Reasoning 96
4.10.1 Syntactic Decomposition 96
4.10.2 Algebraic Decomposition 100
4.10.3 Hoare triples 100
4.11 Loop Termination 101
4.11.1 Trivial Termination 101
4.11.2 Classical Termination 101
4.11.3 Probabilistic Termination 102
4.12 Automated Reasoning 102

Additional Material **105**
4.13 Miscellaneous Mathematics 105
4.13.1 Truncated Subtraction 107

Chapter 1

Overview

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

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

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

Chapter 2

Introduction to pGCL

2.1 Language Primitives

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

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

2.1.1 The Basics

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

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

record *coins* =
a :: *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 (λ*s*. *s* (| *a* := *Heads* |)) = (*a* := (λ*s*. *Heads*))
⟨*proof*⟩

We can treat the record's fields as the names of *variables*. Note that the right-hand side of an assignment is always a function of the current state. Thus we may use a record accessor directly, for example *Apply* (λ*s*. *s*(|*a* := *b s*|)), which updates a with the current value of b. If we wish to formally establish that the previous statement

is correct i.e. that in the final state, a really will have whatever value b had in the initial state, we must first introduce the assertion language.

2.1.2 Assertion and Annotation

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

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

lemma
$\bigwedge x.\ \langle\!\langle \lambda s.\ b\ s = x \rangle\!\rangle \Vdash wp\ (a := b)\ \langle\!\langle \lambda s.\ a\ s = x \rangle\!\rangle$
$\langle proof \rangle$

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
$\langle\!\langle \lambda s::real.\ s \neq 0 \rangle\!\rangle \Vdash wp\ (Apply\ ((*)\ 2)\ ;;\ Apply\ (\lambda s.\ s\ /\ s))\ \langle\!\langle \lambda s.\ s = 1 \rangle\!\rangle$
$\langle proof \rangle$

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 $1 - p$. The probability may depend on the state, and is therefore of type $'s \Rightarrow real$. The following annotation states that the probability of heads is exactly 1/2:

definition

flip-a :: *real* ⇒ *coins prog*
where
flip-a p = *a* := (λ-. *Heads*) $_{(\lambda s.\ p)}\oplus$ *a* := (λ-. *Tails*)

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

2.1.4 Nondeterminism

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

lemma
λ*s*. *1/3* ⊨ *wp* (*flip-a* (*2/3*) ⨅ *flip-a* (*1/3*)) «λ*s*. *a s* = *Heads*»
⟨*proof*⟩

2.1.5 Properties of Expectations

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

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

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

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

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

definition *Puniform* :: '*a set* ⇒ ('*a* ⇒ *real*)
where *Puniform S* = (λ*x*. *if x* ∈ *S then 1* / *card S else 0*)

lemma *Puniform-in*:
x ∈ *S* ⟹ *Puniform S x* = *1* / *card S*
⟨*proof*⟩

lemma *Puniform-out*:
$x \notin S \Longrightarrow$ *Puniform S x* $= 0$
⟨*proof*⟩

lemma *supp-Puniform*:
finite S $\Longrightarrow$ *supp* (*Puniform S*) $= S$
⟨*proof*⟩

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

lemma
$(\lambda s.\ 7/2) =$ *wp* (*bind v at* ($\lambda s.$ *Puniform* $\{1..6\}$ *v*) *in red* := (λ-. *v*)) *red*
⟨*proof*⟩

The expectations of independent variables add:

lemma
$(\lambda s.\ 7) =$ *wp* ((*bind v at* ($\lambda s.$ *Puniform* $\{1..6\}$ *v*) *in red* := ($\lambda s.$ *v*)) ;;
(*bind v at* ($\lambda s.$ *Puniform* $\{1..6\}$ *v*) *in blue* := ($\lambda s.$ *v*)))
($\lambda s.$ *red s* + *blue s*)
⟨*proof*⟩

end

2.2 Loops

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

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

2.2.1 Guaranteed Termination

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

definition *countdown* :: *int prog*
where
countdown = *do* ($\lambda x.\ 0 < x$) $\longrightarrow$ *Apply* ($\lambda s.\ s - 1$) *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* :: *int* $\Rightarrow$ *bool*
where
inv-count = ($\lambda x.\ 0 \leq x$)

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

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

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

lemma *term-countdown*:
«*inv-count*» $\Vdash$ *wp countdown* $(\lambda s.\ 1)$
⟨*proof*⟩

2.2.2 Probabilistic Termination

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

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

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

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

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

Nonetheless, we can show termination .

lemma *wait-for-heads-term*:
$\lambda s.\ 1 \Vdash$ *wp wait-for-heads* $(\lambda s.\ 1)$
⟨*proof*⟩

end

2.3 The Monty Hall Problem

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

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

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

The setting is a game show: There is a prize hidden behind one of three doors, and the contestant is invited to choose one. Once the guess is made, the host 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* (λ*x*. *D*))

Choose door *D*.

definition *guess-behind* :: *nat* ⇒ *game prog*
where *guess-behind D* ≡ *Apply* (*guess-update* (λ*x*. *D*))

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

definition *open-door* :: *nat* ⇒ *game prog*
where *open-door D* ≡ *Apply* (*clue-update* (λ*x*. *D*))

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

definition *hide-prize* :: *game prog*
where *hide-prize* ≡ *hide-behind 1* ⨅ *hide-behind 2* ⨅ *hide-behind 3*

Guess uniformly at random.

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

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

definition *reveal* :: *game prog*
where *reveal* ≡ ⨅*d*∈(λ*s*. {*1*,*2*,*3*} − {*prize s*, *guess s*}). *open-door d*

Switch your guess to the other unopened door.

definition *switch-guess* :: *game prog*
where *switch-guess* ≡ ⨅*d*∈(λ*s*. {*1*,*2*,*3*} − {*clue s*, *guess s*}). *guess-behind d*

The complete game, either with or without switching guesses.

definition *monty* :: *bool* ⇒ *game prog*
where
monty switch ≡ *hide-prize* ;;
make-guess ;;
reveal ;;
(*if switch then switch-guess else Skip*)

2.3.3 A Brute Force Solution

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

lemma *eval-win*[*simp*]:
p = *g* ⟹ «*player-wins*» (*s*(| *prize* := *p*, *guess* := *g*, *clue* := *c* |)) = *1*
⟨*proof*⟩

lemma *eval-loss*[*simp*]:
p ≠ *g* ⟹ «*player-wins*» (*s*(| *prize* := *p*, *guess* := *g*, *clue* := *c* |)) = *0*
⟨*proof*⟩

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*»
$\langle proof \rangle$

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

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

lemma *wp-monty-switch-bruteforce*:
$(\lambda s.\ 2/3) = wp\ (monty\ True)$ «*player-wins*»
$\langle proof \rangle$

2.3.4 A Modular Approach

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

Healthiness

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

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

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

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

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

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*:
$(\lambda s.\ 1) \Vdash$ *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*:
$(\lambda s.\ 2/3 * \text{«}\lambda g.\ \textit{inv-prize}\ g\text{»}\ s) \Vdash$
wp make-guess «λg. guess g ≠ prize g ∧ inv-prize g ∧ inv-guess g»
⟨*proof*⟩

lemma *last-one*:
assumes $a \neq b$ **and** $a \in \{1::nat,2,3\}$ **and** $b \in \{1,2,3\}$
shows $\exists!c.\ \{1,2,3\} - \{b,a\} = \{c\}$
⟨*proof*⟩

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

lemma *wp-reveal*:
«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*:
(λ*s*. *2*/*3*) ⊨ *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*:
(λ*s*. *2*/*3*) ⊨ *wp* (*monty True*) «*player-wins*»
⟨*proof*⟩

end

Chapter 3

Semantic Structures

3.1 Expectations

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

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

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

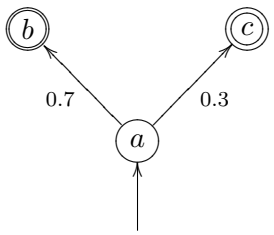

Figure 3.1: A probabilistic automaton

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

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

lemma *bounded-byI*[*intro*]:
⟦ ⋀*x*. *P x* ≤ *b* ⟧ ⟹ *bounded-by b P*
⟨*proof*⟩

lemma *bounded-byI2*[*intro*]:
P ≤ (λ*s*. *b*) ⟹ *bounded-by b P*
⟨*proof*⟩

lemma *bounded-byD*[*dest*]:
bounded-by b P ⟹ *P x* ≤ *b*
⟨*proof*⟩

lemma *bounded-byD2*[*dest*]:
bounded-by b P ⟹ *P* ≤ (λ*s*. *b*)
⟨*proof*⟩

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

definition *bounded* :: (′*a* ⇒ *real*) ⇒ *bool*
where *bounded P* ≡ (∃ *b*. *bounded-by b P*)

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

definition *bound-of* :: (′*a* ⇒ *real*) ⇒ *real*
where *bound-of P* ≡ *Sup* (*P* ‘ *UNIV*)

lemma *bounded-bdd-above*[*intro*]:
assumes *bP*: *bounded P*
shows *bdd-above* (*range P*)
⟨*proof*⟩

The least upper bound has the usual properties:

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

assumes *bP*: *bounded-by b P*
shows *bound-of P* $\leq$ *b*
⟨*proof*⟩

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

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

lemma *bounded-by-mono*:
⟦ *bounded-by a P*; *a* $\leq$ *b* ⟧ $\Longrightarrow$ *bounded-by b P*
⟨*proof*⟩

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

This is occasionally easier to apply:

lemma *bounded-by-bound-of-alt*:
⟦ *bounded P*; *bound-of P* = *a* ⟧ $\Longrightarrow$ *bounded-by a P*
⟨*proof*⟩

lemma *bounded-const*[*simp*]:
bounded (λ*x*. *c*)
⟨*proof*⟩

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

lemma *bounded-by-mono-alt*[*intro*]:
⟦ *bounded-by b Q*; *P* $\leq$ *Q* ⟧ $\Longrightarrow$ *bounded-by b P*
⟨*proof*⟩

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

lemma *bound-of-leI*:
assumes ⋀*x*. *P x* $\leq$ (*c*::*real*)
shows *bound-of P* $\leq$ *c*
⟨*proof*⟩

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

$[\![P \leq Q; \textit{bounded } P; \textit{bounded } Q]\!] \Longrightarrow \textit{bound-of } P \leq \textit{bound-of } Q$
$\langle proof \rangle$

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

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

3.1.2 Non-Negative Functions.

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

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

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

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

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

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

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

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

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

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

⟦ *bounded P*; *nneg P* ⟧ ⟹ *0* ≤ *bound-of P*
⟨*proof*⟩

lemma *nneg-bounded-by-nneg*[*dest*]:
⟦ *bounded-by b P*; *nneg P* ⟧ ⟹ *0* ≤ (*b*::*real*)
⟨*proof*⟩

lemma *bounded-by-nneg*[*dest*]:
fixes *P*::*'s* ⇒ *real*
shows ⟦ *bounded-by b P*; *nneg P* ⟧ ⟹ *0* ≤ *b*
⟨*proof*⟩

3.1.3 Sound Expectations

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

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

lemma *soundI*:
⟦ *bounded P*; *nneg P* ⟧ ⟹ *sound P*
⟨*proof*⟩

lemma *soundI2*[*intro*]:
⟦ *bounded-by b P*; *nneg P* ⟧ ⟹ *sound P*
⟨*proof*⟩

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

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

lemma *bound-of-sound*[*intro*]:
assumes *sP*: *sound P*
shows *0* ≤ *bound-of P*
⟨*proof*⟩

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

lemma *sound-sum*[*simp*,*intro*]:
assumes *sP*: *sound P* **and** *sQ*: *sound Q*
shows *sound* (λ*s*. *P s* + *Q s*)
⟨*proof*⟩

lemma *mult-sound*:
assumes *sP*: *sound P* **and** *sQ*: *sound Q*
shows *sound* $(\lambda s.\ P\ s * Q\ s)$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *sc-sound*:
⟦ *sound P*; $0 \leq c$ ⟧ $\Longrightarrow$ *sound* $(\lambda s.\ c * P\ s)$
⟨*proof*⟩

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

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* $(a + b)$ $(\lambda s.\ P\ s + Q\ s)$
⟨*proof*⟩

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

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

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

3.1.4 Unitary expectations

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

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

lemma *unitaryI*[*intro*]:
⟦ *sound P*; *bounded-by 1 P* ⟧ ⟹ *unitary P*
⟨*proof*⟩

lemma *unitaryI2*:
⟦ *nneg P*; *bounded-by 1 P* ⟧ ⟹ *unitary P*
⟨*proof*⟩

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

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

3.1.5 Standard Expectations

definition
embed-bool :: (*'s* ⇒ *bool*) ⇒ *'s* ⇒ *real* (‹« - »› *1000*)
where

«*P*» ≡ (λ*s*. *if P s then 1 else 0*)

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

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

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

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

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

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

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

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

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

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

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

lemma *expect-1-I*:
P x ⟹ *1* ≤ «*P*» *x*
⟨*proof*⟩

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 ⇒ bool) ⇒ 's ⇒ bool* (‹𝒩›)
where *negate P = (λs. ¬ P s)*

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

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

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

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

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

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

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

lemma *embed-bool-cancel*:
*«G» s * «𝒩 G» s = 0*
⟨*proof*⟩

3.1.6 Entailment

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

abbreviation *entails* :: *('s ⇒ real) ⇒ ('s ⇒ real) ⇒ bool* (‹- ⊨ -› *50*)

where $P \models Q \equiv P \leq Q$

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

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

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

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

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

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

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

3.1.7 Expectation Conjunction

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

definition
exp-conj :: ('*s* ⇒ *real*) ⇒ ('*s* ⇒ *real*) ⇒ ('*s* ⇒ *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*]:
$b \leq 1 \Longrightarrow 0$.& $b = 0$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

3.1.8 Rules Involving Conjunction.

lemma *exp-conj-mono-left*:

$P \Vdash Q \Longrightarrow P \;\&\&\; R \Vdash Q \;\&\&\; R$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

bounded-by 1 P $\Longrightarrow$ *P && (λs. 0) = (λs. 0)*
⟨*proof*⟩

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

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

3.1.9 Rules Involving Entailment and Conjunction Together

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

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

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

lemma *unitary-top*[*iff*]:
unitary (λs. 1::*real)*
⟨*proof*⟩

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

lemma *unitary-const*[*iff*]:
⟦ *0 ≤ c*; *c ≤ 1* ⟧ $\Longrightarrow$ *unitary (λs. c)*
⟨*proof*⟩

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

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

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

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

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

end

3.2 Expectation Transformers

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

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

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

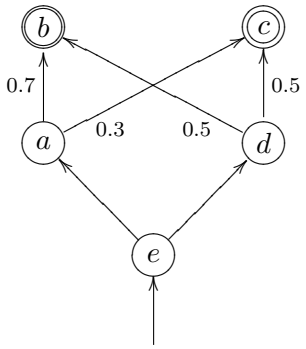

Figure 3.2: A nondeterministic-probabilistic automaton.

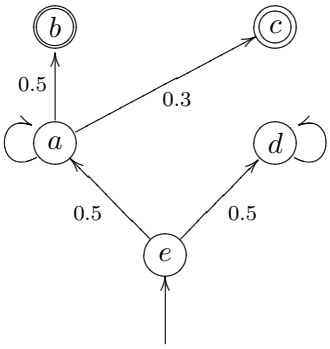

Figure 3.3: A diverging automaton.

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 $a \to a \to \ldots$ 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
le-utrans t u $\longleftrightarrow$ $(\forall P.$ *unitary P* $\longrightarrow$ *t P* $\leq$ *u P*$)$

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

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

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

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

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

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

lemma *le-trans-refl*[*iff*]:
le-trans x x
⟨*proof*⟩

lemma *le-utrans-refl*[*iff*]:
le-utrans x x
⟨*proof*⟩

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

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

Transformer equivalence is induced by comparison:

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

definition
equiv-utrans :: *'s trans* ⇒ *'s trans* ⇒ *bool*
where
equiv-utrans t u ⟷ *le-utrans t u* ∧ *le-utrans u t*

lemma *equiv-transI*[*intro*]:
⟦ ⋀*P*. *sound P* ⟹ *t P* = *u P* ⟧ ⟹ *equiv-trans t u*
⟨*proof*⟩

lemma *equiv-utransI*[*intro*]:
⟦ ⋀*P*. *sound P* ⟹ *t P* = *u P* ⟧ ⟹ *equiv-utrans t u*
⟨*proof*⟩

lemma *equiv-transD*[*dest*]:
⟦ *equiv-trans t u*; *sound P* ⟧ ⟹ *t P* = *u P*
⟨*proof*⟩

lemma *equiv-utransD*[*dest*]:
⟦ *equiv-utrans t u*; *unitary P* ⟧ ⟹ *t P* = *u P*
⟨*proof*⟩

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

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

lemma *le-trans-antisym*:
⟦ *le-trans x y*; *le-trans y x* ⟧ ⟹ *equiv-trans x y*
⟨*proof*⟩

lemma *le-utrans-antisym*:
⟦ *le-utrans x y*; *le-utrans y x* ⟧ ⟹ *equiv-utrans x y*
⟨*proof*⟩

lemma *equiv-trans-comm*[*ac-simps*]:
equiv-trans t u ⟷ *equiv-trans u t*
⟨*proof*⟩

lemma *equiv-utrans-comm*[*ac-simps*]:
equiv-utrans t u ⟷ *equiv-utrans u t*
⟨*proof*⟩

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

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

lemma *equiv-imp-le-alt*:
equiv-trans t u $\Longrightarrow$ *le-trans u t*
⟨*proof*⟩

lemma *equiv-uimp-le-alt*:
equiv-utrans t u $\Longrightarrow$ *le-utrans u t*
⟨*proof*⟩

lemma *le-trans-equiv-rsp*[*simp*]:
equiv-trans t u $\Longrightarrow$ *le-trans t v* $\longleftrightarrow$ *le-trans u v*
⟨*proof*⟩

lemma *le-utrans-equiv-rsp*[*simp*]:
equiv-utrans t u $\Longrightarrow$ *le-utrans t v* $\longleftrightarrow$ *le-utrans u v*
⟨*proof*⟩

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

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

lemma *le-trans-equiv-rsp-right*[*simp*]:
equiv-trans t u $\Longrightarrow$ *le-trans v t* $\longleftrightarrow$ *le-trans v u*
⟨*proof*⟩

lemma *le-utrans-equiv-rsp-right*[*simp*]:
equiv-utrans t u $\Longrightarrow$ *le-utrans v t* $\longleftrightarrow$ *le-utrans v u*
⟨*proof*⟩

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

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

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

⟨*proof*⟩

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

lemma *equiv-trans-equiv-utrans*[*dest*]:
equiv-trans t u ⟹ *equiv-utrans t u*
⟨*proof*⟩

3.2.2 Healthy Transformers

Feasibility

definition *feasible* :: ((′*a* ⇒ *real*) ⇒ (′*a* ⇒ *real*)) ⇒ *bool*
where *feasible t* ⟷ (∀ *P b*. *bounded-by b P* ∧ *nneg P* ⟶ *bounded-by b* (*t P*) ∧ *nneg* (*t P*))

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

lemma *feasibleI*[*intro*]:
⟦ ⋀*b P*. ⟦ *bounded-by b P*; *nneg P* ⟧ ⟹ *bounded-by b* (*t P*);
⋀*b P*. ⟦ *bounded-by b P*; *nneg P* ⟧ ⟹ *nneg* (*t P*) ⟧ ⟹ *feasible t*
⟨*proof*⟩

lemma *feasible-boundedD*[*dest*]:
⟦ *feasible t*; *bounded-by b P*; *nneg P* ⟧ ⟹ *bounded-by b* (*t P*)
⟨*proof*⟩

lemma *feasible-nnegD*[*dest*]:
⟦ *feasible t*; *bounded-by b P*; *nneg P* ⟧ ⟹ *nneg* (*t P*)
⟨*proof*⟩

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

lemma *feasible-pr-0*[*simp*]:
fixes *t*::(′*s* ⇒ *real*) ⇒ ′*s* ⇒ *real*
assumes *ft*: *feasible t*
shows *t* (λ*x*. *0*) = (λ*x*. *0*)
⟨*proof*⟩

lemma *feasible-id*:

feasible $(\lambda x.\ x)$
⟨*proof*⟩

lemma *feasible-bounded-by*[*dest*]:
⟦ *feasible t*; *sound P*; *bounded-by b P* ⟧ ⟹ *bounded-by b* $(t\ P)$
⟨*proof*⟩

lemma *feasible-fixes-top*:
feasible t ⟹ $t\ (\lambda s.\ 1) \leq (\lambda s.\ (1{::}real))$
⟨*proof*⟩

lemma *feasible-fixes-bot*:
assumes *ft*: *feasible t*
shows $t\ (\lambda s.\ 0) = (\lambda s.\ 0)$
⟨*proof*⟩

lemma *feasible-unitaryD*[*dest*]:
assumes *ft*: *feasible t* **and** *uP*: *unitary P*
shows *unitary* $(t\ P)$
⟨*proof*⟩

Monotonicity

definition
mono-trans :: $((('s \Rightarrow real) \Rightarrow ('s \Rightarrow real)) \Rightarrow bool$
where
mono-trans t ≡ $\forall P\ Q.\ (sound\ P \wedge sound\ Q \wedge P \leq Q) \longrightarrow t\ P \leq t\ Q$

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

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

lemma *mono-transI*[*intro*]:
⟦ $\bigwedge P\ Q.$ ⟦ *sound P*; *sound Q*; $P \leq Q$ ⟧ ⟹ $t\ P \leq t\ Q$ ⟧ ⟹ *mono-trans t*
⟨*proof*⟩

lemma *mono-transD*[*dest*]:
⟦ *mono-trans t*; *sound P*; *sound Q*; $P \leq Q$ ⟧ ⟹ $t\ P \leq t\ Q$
⟨*proof*⟩

Scaling

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

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

The *scaling* and feasibility properties together allow us to treat transformers as a complete lattice, when operating on bounded expectations. The action of a transformer on such a bounded expectation is completely determined by its action on *unitary* expectations (those bounded by 1): $t\ P\ s = bound\text{-}of\ P * t\ (\lambda s.\ P\ s\ /\ bound\text{-}of\ P)\ s$. 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*]:
$[\![\bigwedge P\ c\ x.\ [\![sound\ P; 0 \leq c]\!] \Longrightarrow c * t\ P\ x = t\ (\lambda x.\ c * P\ x)\ x]\!] \Longrightarrow scaling\ t$
⟨*proof*⟩

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

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

Healthiness

Healthy transformers are feasible and monotonic, and respect scaling

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

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

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

lemma *healthy-monoD*[*dest*]:
$healthy\ t \Longrightarrow mono\text{-}trans\ t$
⟨*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*]:
healthy t $\Longrightarrow$ *le-trans t* (λ*P s*. *bound-of P*)
⟨*proof*⟩

lemma *healthy-pr-bot*[*simp*]:
healthy t $\Longrightarrow$ *t* (λ*s*. *0*) = (λ*s*. *0*)
⟨*proof*⟩

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

lemma *healthy-equivI*:
fixes *t*::(*'s* $\Rightarrow$ *real*) $\Rightarrow$ *'s* $\Rightarrow$ *real* **and** *u*
assumes *equiv*: *equiv-trans t u*
and *healthy*: *healthy t*
shows *healthy u*
⟨*proof*⟩

lemma *healthy-equiv*:
equiv-trans t u $\Longrightarrow$ *healthy t* $\longleftrightarrow$ *healthy u*
⟨*proof*⟩

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

lemma *healthy-top*[*iff*]:
healthy (λ*P s*. *bound-of P*)
⟨*proof*⟩

lemma *healthy-bot*[*iff*]:
healthy (λ*P s*. *0*)
⟨*proof*⟩

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

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

lemma *nearly-healthyI*[*intro*]:
⟦ $\bigwedge$*P*. *unitary P* $\Longrightarrow$ *unitary* (*t P*);
$\bigwedge$*P Q*. ⟦ *unitary P*; *unitary Q*; *P* $\models$ *Q* ⟧ $\Longrightarrow$ *t P* $\models$ *t Q* ⟧ $\Longrightarrow$ *nearly-healthy t*
⟨*proof*⟩

lemma *nearly-healthy-monoD*[*dest*]:
⟦ *nearly-healthy t*; *P* ⊨ *Q*; *unitary P*; *unitary Q* ⟧ ⟹ *t P* ⊨ *t Q*
⟨*proof*⟩

lemma *nearly-healthy-unitaryD*[*dest*]:
⟦ *nearly-healthy t*; *unitary P* ⟧ ⟹ *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* ⇒ *real*) ⇒ ('*s* ⇒ *real*)) ⇒ *bool*
where
sublinear t ⟷ (∀ *a b c P Q s*. (*sound P* ∧ *sound Q* ∧ *0* ≤ *a* ∧ *0* ≤ *b* ∧ *0* ≤ *c*) ⟶
a ∗ *t P s* + *b* ∗ *t Q s* ⊖ *c*
≤ *t* (λ*s'*. *a* ∗ *P s'* + *b* ∗ *Q s'* ⊖ *c*) *s*)

lemma *sublinearI*[*intro*]:
⟦ ⋀*a b c P Q s*. ⟦ *sound P*; *sound Q*; *0* ≤ *a*; *0* ≤ *b*; *0* ≤ *c* ⟧ ⟹
a ∗ *t P s* + *b* ∗ *t Q s* ⊖ *c* ≤
t (λ*s'*. *a* ∗ *P s'* + *b* ∗ *Q s'* ⊖ *c*) *s* ⟧ ⟹ *sublinear t*
⟨*proof*⟩

lemma *sublinearD*[*dest*]:
⟦ *sublinear t*; *sound P*; *sound Q*; *0* ≤ *a*; *0* ≤ *b*; *0* ≤ *c* ⟧ ⟹
a ∗ *t P s* + *b* ∗ *t Q s* ⊖ *c* ≤
t (λ*s'*. *a* ∗ *P s'* + *b* ∗ *Q s'* ⊖ *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* ::

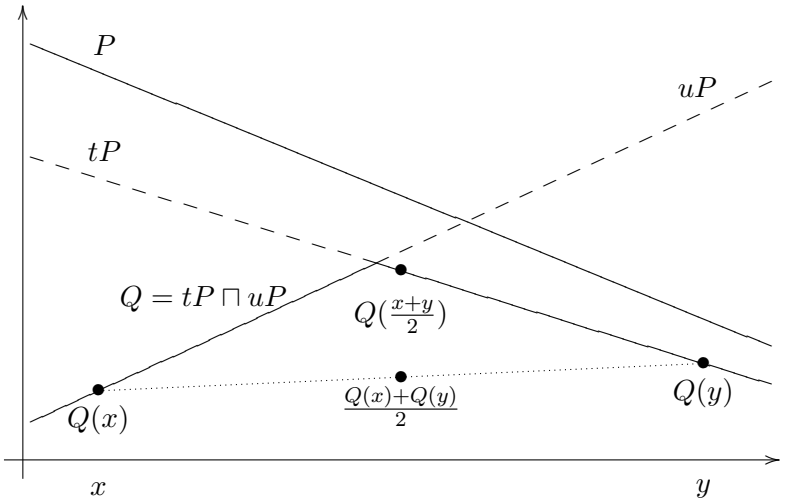

Figure 3.4: A graphical depiction of sub-additivity as convexity.

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

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

Here P is an affine function (expectation) *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*]:
⟦ ⋀*P Q s*. ⟦ *sound P*; *sound Q* ⟧ ⟹
t P s + *t Q s* ≤ *t* (λ*s*′. *P s*′ + *Q s*′) *s* ⟧ ⟹ *sub-add t*
⟨*proof*⟩

lemma *sub-addI2*:
⟦⋀*P Q*. ⟦ *sound P*; *sound Q* ⟧ ⟹
λ*s*. *t P s* + *t Q s* ⊨ *t* (λ*s*. *P s* + *Q s*)⟧ ⟹
sub-add t
⟨*proof*⟩

lemma *sub-addD*[*dest*]:
⟦ *sub-add t*; *sound P*; *sound Q* ⟧ ⟹ *t P s* + *t Q s* ≤ *t* (λ*s*′. *P s*′ + *Q s*′) *s*
⟨*proof*⟩

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

Sublinearity and feasibility imply sub-additivity.

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

A few properties following from sub-additivity:

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

shows $t\ «P»\ s + t\ «\mathcal{N}\ P»\ s \leq 1$
⟨*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 $(\sum y{\in}\{s.\ G\ s\}.\ \ P\ y * t\ « \lambda z.\ z = y\ » s) +$
$(\sum y{\in}\{s.\ \neg G\ s\}.\ P\ y * t\ « \lambda z.\ z = y\ » s) \leq t\ P\ s$
⟨*proof*⟩

Sub-distributivity

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

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-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-distrib t*
⟨*proof*⟩

lemma *sub-distribD*[*dest*]:
$[\![$ *sub-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*
⟨*proof*⟩

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

lemma *sd-sa-sublinear*:
fixes *t*::(*'s* ⇒ *real*) ⇒ *'s* ⇒ *real*
assumes *sdt*: *sub-distrib t* **and** *sat*: *sub-add t* **and** *ht*: *healthy t*
shows *sublinear t*
⟨*proof*⟩

Sub-conjunctivity

definition
sub-conj :: ((*'s* ⇒ *real*) ⇒ *'s* ⇒ *real*) ⇒ *bool*
where
sub-conj t ≡ ∀ *P Q*. (*sound P* ∧ *sound Q*) ⟶
t P && *t Q* ⊨ *t* (*P* && *Q*)

lemma *sub-conjI*[*intro*]:
⟦ ⋀*P Q*. ⟦ *sound P*; *sound Q* ⟧ ⟹
t P && *t Q* ⊨ *t* (*P* && *Q*) ⟧ ⟹ *sub-conj t*
⟨*proof*⟩

lemma *sub-conjD*[*dest*]:
⟦ *sub-conj t*; *sound P*; *sound Q* ⟧ ⟹ *t P* && *t Q* ⊨ *t* (*P* && *Q*)
⟨*proof*⟩

lemma *sub-conj-wp-twice*:
fixes *f*::*'s* ⇒ ((*'s* ⇒ *real*) ⇒ *'s* ⇒ *real*)
assumes *all*: ∀ *s*. *sub-conj* (*f s*)
shows *sub-conj* (λ*P s*. *f s P s*)
⟨*proof*⟩

Sublinearity implies sub-conjunctivity:

lemma *sublinear-sub-conj*:
fixes *t*::(*'s* ⇒ *real*) ⇒ *'s* ⇒ *real*
assumes *slt*: *sublinear t*
shows *sub-conj t*
⟨*proof*⟩

Sublinearity under equivalence

Sublinearity is preserved by equivalence.

lemma *equiv-sublinear*:
⟦ *equiv-trans t u*; *sublinear t*; *healthy t* ⟧ ⟹ *sublinear u*
⟨*proof*⟩

3.2.4 Determinism

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

Additivity

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

definition
$additive :: ((\prime a \Rightarrow real) \Rightarrow \prime a \Rightarrow real) \Rightarrow bool$
where
$additive\ t \equiv \forall P\ Q.\ (sound\ P \wedge sound\ Q) \longrightarrow$
$t\ (\lambda s.\ P\ s + Q\ s) = (\lambda s.\ t\ P\ s + t\ Q\ s)$

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

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

Additivity is strictly stronger than sub-additivity.

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

The additivity property extends to finite summation.

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

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::$(\prime s$::*finite* $\Rightarrow$ *real*$) \Rightarrow \prime s \Rightarrow$ *real*
assumes *additive*: *additive t*
and *ht*: *healthy t*

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

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

lemma *additive-guard-split*:
fixes *t*::$('s{::}finite \Rightarrow real) \Rightarrow\ 's \Rightarrow real$
assumes *additive*: *additive t*
and *ht*: *healthy t*
and *sP*: *sound P*
shows *t P x* = $(\sum y{\in}\{s.\ \ G\ s\}.\ P\ y * t\ «\lambda z.\ z = y»\ x) +$
$(\sum y{\in}\{s.\ \neg\ G\ s\}.\ P\ y * t\ «\lambda z.\ z = y»\ x)$
⟨*proof*⟩

Maximality

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

lemma *maximalI*[*intro*]:
⟦ $\bigwedge c.\ 0 \leq c \Longrightarrow t\ (\lambda\text{-}.\ c) = (\lambda\text{-}.\ c)$ ⟧ ⟹ *maximal t*
⟨*proof*⟩

lemma *maximalD*[*dest*]:
⟦ *maximal t*; $0 \leq c$ ⟧ ⟹ $t\ (\lambda\text{-}.\ c) = (\lambda\text{-}.\ c)$
⟨*proof*⟩

A transformer that is both additive and maximal is deterministic:

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

lemma *determI*[*intro*]:
⟦ *additive t*; *maximal t* ⟧ ⟹ *determ t*
⟨*proof*⟩

lemma *determ-additiveD*[*intro*]:
determ t ⟹ *additive t*
⟨*proof*⟩

lemma *determ-maximalD*[*intro*]:
determ t ⟹ *maximal t*
⟨*proof*⟩

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 «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 ⊨ t R*
and *wp2*: *Q ⊨ t S*
and *sc*: *sub-conj t*
and *sR*: *sound R*
and *sS*: *sound S*
shows *P && Q ⊨ t (R && S)*
⟨*proof*⟩

These allow mismatched results to be composed

lemma *entails-strengthen-post*:
⟦ *P ⊨ t Q*; *healthy t*; *sound R*; *Q ⊨ R*; *sound Q* ⟧ ⟹ *P ⊨ t R*
⟨*proof*⟩

lemma *entails-weaken-pre*:
⟦ *Q ⊨ t R*; *P ⊨ Q* ⟧ ⟹ *P ⊨ 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 ⊨ t Q* **and** *h*: *healthy t*
and *sQ*: *sound Q* **and** *pos*: *0 ≤ c*
shows *(λs. c * P s) ⊨ t (λs. 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*»
⟨*proof*⟩

The other direction works too.

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

Predicate conjunction behaves as expected:

lemma *conj-post*:
$[\![\ P \Vdash t\ «\lambda s.\ Q\ s \wedge R\ s»;\ \textit{healthy}\ t\ ]\!] \Longrightarrow P \Vdash t\ «Q»$
⟨*proof*⟩

Similar to $[\![\textit{healthy}\ ?t;\ \bigwedge s.\ ?P\ s \Longrightarrow 1 \leq ?t\ «\ ?Q\ »\ s]\!] \Longrightarrow «\ ?P\ » \Vdash ?t\ «\ ?Q\ »$, 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*
⟨*proof*⟩

end

3.3 Induction

theory *Induction*
imports *Expectations Transformers*
begin

3.3.1 The Lattice of Expectations

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

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

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

wish to find the least fixed point of $t\ P$. In practice, t is generally doubly healthy, that is $\forall P.\ sound\ P \longrightarrow healthy\ (t\ P)$ and $\forall Q.\ sound\ Q \longrightarrow healthy\ (\lambda P.\ t\ P\ Q)$. 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* ⇒ *'s expect*
where *Inf-exp S* = $(\lambda s.\ Inf\ \{f\ s\ |f.\ f \in S\})$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *gfp-exp-lemma2*:
assumes *ft*: ⋀*P*. *unitary P* ⟹ *unitary* (*t P*)
and *mt*: ⋀*P Q*. ⟦ *unitary P*; *unitary Q*; *P* ⊨ *Q* ⟧ ⟹ *t P* ⊨ *t Q*
shows *gfp-exp t* ≤ *t* (*gfp-exp t*)
⟨*proof*⟩

lemma *gfp-exp-lemma3*:
assumes *ft*: ⋀*P*. *unitary P* ⟹ *unitary* (*t P*)
and *mt*: ⋀*P Q*. ⟦ *unitary P*; *unitary Q*; *P* ⊨ *Q* ⟧ ⟹ *t P* ⊨ *t Q*
shows *t* (*gfp-exp t*) ≤ *gfp-exp t*
⟨*proof*⟩

lemma *gfp-exp-unfold*:
(⋀*P*. *unitary P* ⟹ *unitary* (*t P*)) ⟹ (⋀*P Q*. ⟦ *unitary P*; *unitary Q*; *P* ⊨ *Q* ⟧ ⟹ *t P* ⊨ *t Q*) ⟹
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: λ*P* -. *bound-of P*.

definition *Inf-trans* :: *'s trans set* ⇒ *'s trans*
where *Inf-trans S* = (λ*P*. *Inf-exp* {*t P* |*t*. *t* ∈ *S*})

lemma *Inf-trans-lower*:
⟦ *t* ∈ *S*; ∀ *u*∈*S*. ∀ *P*. *sound P* ⟶ *sound* (*u P*) ⟧ ⟹ *le-trans* (*Inf-trans S*) *t*
⟨*proof*⟩

lemma *Inf-trans-greatest*:
⟦ *S* ≠ {}; ∀ *t*∈*S*. ∀ *P*. *le-trans u t* ⟧ ⟹ *le-trans u* (*Inf-trans S*)
⟨*proof*⟩

definition *Sup-trans* :: *'s trans set* ⇒ *'s trans*
where *Sup-trans S* = (λ*P*. *Sup-exp* {*t P* |*t*. *t* ∈ *S*})

lemma *Sup-trans-upper*:
⟦ $t \in S$; $\forall u{\in}S.\ \forall P.$ *unitary* $P \longrightarrow$ *unitary* $(u\ P)$ ⟧ $\Longrightarrow$ *le-utrans* t (*Sup-trans* S)
⟨*proof*⟩

lemma *Sup-trans-upper2*:
⟦ $t \in S$; $\forall u{\in}S.\ \forall P.$ (*nneg* $P \wedge$ *bounded-by* $b\ P$) $\longrightarrow$ (*nneg* $(u\ P) \wedge$ *bounded-by* $b\ (u\ P)$);
nneg P; *bounded-by* $b\ P$ ⟧ $\Longrightarrow$ $t\ P \Vdash$ *Sup-trans* $S\ P$
⟨*proof*⟩

lemma *Sup-trans-least*:
⟦ $\forall t{\in}S.$ *le-utrans* $t\ u$; $\bigwedge P.$ *unitary* $P \Longrightarrow$ *unitary* $(u\ P)$ ⟧ $\Longrightarrow$ *le-utrans* (*Sup-trans* S) u
⟨*proof*⟩

lemma *Sup-trans-least2*:
⟦ $\forall t{\in}S.\ \forall P.$ *nneg* $P \longrightarrow$ *bounded-by* $b\ P \longrightarrow t\ P \Vdash u\ P$;
$\forall u{\in}S.\ \forall P.$ (*nneg* $P \wedge$ *bounded-by* $b\ P$) $\longrightarrow$ (*nneg* $(u\ P) \wedge$ *bounded-by* $b\ (u\ P)$);
nneg P; *bounded-by* $b\ P$; $\bigwedge P.$ ⟦ *nneg* P; *bounded-by* $b\ P$ ⟧ $\Longrightarrow$ *nneg* $(u\ P)$ ⟧ $\Longrightarrow$
Sup-trans $S\ P \Vdash u\ P$
⟨*proof*⟩

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

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

lemma *lfp-trans-lowerbound*:
⟦ *le-trans* $(T\ t)\ t$; $\bigwedge P.$ *sound* $P \Longrightarrow$ *sound* $(t\ P)$ ⟧ $\Longrightarrow$ *le-trans* (*lfp-trans* T) t
⟨*proof*⟩

lemma *lfp-trans-greatest*:
⟦ $\bigwedge t\ P.$ ⟦ *le-trans* $(T\ t)\ t$; $\bigwedge P.$ *sound* $P \Longrightarrow$ *sound* $(t\ P)$ ⟧ $\Longrightarrow$ *le-trans* $u\ t$;
$\bigwedge P.$ *sound* $P \Longrightarrow$ *sound* $(v\ P)$; *le-trans* $(T\ v)\ v$ ⟧ $\Longrightarrow$
le-trans u (*lfp-trans* T)
⟨*proof*⟩

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

lemma *lfp-trans-unitary*:

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

lemma *lfp-trans-lemma2*:
fixes *v*::'*s trans*
assumes *mono*: $\bigwedge t\ u$. ⟦ *le-trans t u*; $\bigwedge P$. *sound P* $\Longrightarrow$ *sound* (*t P*);
$\bigwedge P$. *sound P* $\Longrightarrow$ *sound* (*u P*) ⟧ $\Longrightarrow$ *le-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-trans* (*T v*) *v*
and *sv*: $\bigwedge P$. *sound P* $\Longrightarrow$ *sound* (*v P*)
shows *le-trans* (*T* (*lfp-trans T*)) (*lfp-trans T*)
⟨*proof*⟩

lemma *lfp-trans-lemma3*:
fixes *v*::'*s trans*
assumes *mono*: $\bigwedge t\ u$. ⟦ *le-trans t u*; $\bigwedge P$. *sound P* $\Longrightarrow$ *sound* (*t P*);
$\bigwedge P$. *sound P* $\Longrightarrow$ *sound* (*u P*) ⟧ $\Longrightarrow$ *le-trans* (*T t*) (*T u*)
and *sT*: $\bigwedge t\ P$. ⟦ $\bigwedge Q$. *sound Q* $\Longrightarrow$ *sound* (*t Q*); *sound P* ⟧ $\Longrightarrow$ *sound* (*T t P*)
and *fv*: *le-trans* (*T v*) *v*
and *sv*: $\bigwedge P$. *sound P* $\Longrightarrow$ *sound* (*v P*)
shows *le-trans* (*lfp-trans T*) (*T* (*lfp-trans T*))
⟨*proof*⟩

lemma *lfp-trans-unfold*:
fixes *P*::'*s expect*
assumes *mono*: $\bigwedge t\ u$. ⟦ *le-trans t u*; $\bigwedge P$. *sound P* $\Longrightarrow$ *sound* (*t P*);
$\bigwedge P$. *sound P* $\Longrightarrow$ *sound* (*u P*) ⟧ $\Longrightarrow$ *le-trans* (*T t*) (*T u*)
and *sT*: $\bigwedge t\ P$. ⟦ $\bigwedge Q$. *sound Q* $\Longrightarrow$ *sound* (*t Q*); *sound P* ⟧ $\Longrightarrow$ *sound* (*T t P*)
and *fv*: *le-trans* (*T v*) *v*
and *sv*: $\bigwedge P$. *sound P* $\Longrightarrow$ *sound* (*v P*)
shows *equiv-trans* (*lfp-trans T*) (*T* (*lfp-trans T*))
⟨*proof*⟩

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

lemma *gfp-trans-upperbound*:
⟦ *le-utrans t* (*T t*); $\bigwedge P$. *unitary P* $\Longrightarrow$ *unitary* (*t P*) ⟧ $\Longrightarrow$ *le-utrans t* (*gfp-trans T*)
⟨*proof*⟩

lemma *gfp-trans-least*:
⟦ $\bigwedge t$. ⟦ *le-utrans t* (*T t*); $\bigwedge P$. *unitary P* $\Longrightarrow$ *unitary* (*t P*) ⟧ $\Longrightarrow$ *le-utrans t u*;
$\bigwedge P$. *unitary P* $\Longrightarrow$ *unitary* (*u P*) ⟧ $\Longrightarrow$
le-utrans (*gfp-trans T*) *u*

⟨*proof*⟩

lemma *gfp-trans-unitary*:
fixes *P*::*'s expect*
assumes *uP*: *unitary P*
shows *unitary* (*gfp-trans T P*)
⟨*proof*⟩

lemma *gfp-trans-lemma2*:
assumes *mono*: ⋀*t u*. ⟦ *le-utrans t u*; ⋀*P*. *unitary P* ⟹ *unitary* (*t P*);
⋀*P*. *unitary P* ⟹ *unitary* (*u P*) ⟧ ⟹ *le-utrans* (*T t*) (*T u*)
and *hT*: ⋀*t P*. ⟦ ⋀*Q*. *unitary Q* ⟹ *unitary* (*t Q*); *unitary P* ⟧ ⟹ *unitary* (*T t P*)
shows *le-utrans* (*gfp-trans T*) (*T* (*gfp-trans T*))
⟨*proof*⟩

lemma *gfp-trans-lemma3*:
assumes *mono*: ⋀*t u*. ⟦ *le-utrans t u*; ⋀*P*. *unitary P* ⟹ *unitary* (*t P*);
⋀*P*. *unitary P* ⟹ *unitary* (*u P*) ⟧ ⟹ *le-utrans* (*T t*) (*T u*)
and *hT*: ⋀*t P*. ⟦ ⋀*Q*. *unitary Q* ⟹ *unitary* (*t Q*); *unitary P* ⟧ ⟹ *unitary* (*T t P*)
shows *le-utrans* (*T* (*gfp-trans T*)) (*gfp-trans T*)
⟨*proof*⟩

lemma *gfp-trans-unfold*:
assumes *mono*: ⋀*t u*. ⟦ *le-utrans t u*; ⋀*P*. *unitary P* ⟹ *unitary* (*t P*);
⋀*P*. *unitary P* ⟹ *unitary* (*u P*) ⟧ ⟹ *le-utrans* (*T t*) (*T u*)
and *hT*: ⋀*t P*. ⟦ ⋀*Q*. *unitary Q* ⟹ *unitary* (*t Q*); *unitary P* ⟧ ⟹ *unitary* (*T t P*)
shows *equiv-utrans* (*gfp-trans T*) (*T* (*gfp-trans T*))
⟨*proof*⟩

3.3.3 Tail Recursion

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

lemma *gfp-pulldown*:
fixes *P*::*'s expect*
assumes *tailcall*: ⋀*u P*. *unitary P* ⟹ *T u P* = *t P* (*u P*)
and *fT*: ⋀*t P*. ⟦ ⋀*Q*. *unitary Q* ⟹ *unitary* (*t Q*); *unitary P* ⟧ ⟹ *unitary* (*T t P*)
and *ft*: ⋀*P Q*. *unitary P* ⟹ *unitary Q* ⟹ *unitary* (*t P Q*)
and *mt*: ⋀*P Q R*. ⟦ *unitary P*; *unitary Q*; *unitary R*; *Q* ⊨ *R* ⟧ ⟹ *t P Q* ⊨ *t P R*
and *uP*: *unitary P*
and *monoT*: ⋀*t u*. ⟦ *le-utrans t u*; ⋀*P*. *unitary P* ⟹ *unitary* (*t P*);
⋀*P*. *unitary P* ⟹ *unitary* (*u P*) ⟧ ⟹ *le-utrans* (*T t*) (*T u*)
shows *gfp-trans T P* = *gfp-exp* (*t P*) (**is** *?X P* = *?Y P*)
⟨*proof*⟩

lemma *lfp-pulldown*:
fixes *P*::*'s expect* **and** *t*::*'s expect* ⇒ *'s 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-trans T P* = *lfp-exp* $(t\ P)$ (**is** *?X P* = *?Y P*)
$\langle proof \rangle$

definition *Inf-utrans* :: *'s trans set* $\Rightarrow$ *'s trans*
where *Inf-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 $'s\ prog = bool \Rightarrow ('s \Rightarrow real) \Rightarrow ('s \Rightarrow real)$

Abort either always fails, $\lambda P\ s.\ 0$, or always succeeds, $\lambda P\ s.\ 1$.

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

Skip does nothing at all.

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

Apply lifts a state transformer into the space of programs.

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

Seq is sequential composition.

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

PC is *probabilistic* choice between programs.

definition *PC* :: $'s\ prog \Rightarrow ('s \Rightarrow real) \Rightarrow 's\ prog \Rightarrow 's\ prog$
(‹- _⊕ -› [*58*,*57*,*57*] *57*)

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

DC is *demonic* choice between programs.

definition $DC :: 's\ prog \Rightarrow 's\ prog \Rightarrow 's\ prog$ (‹- $\bigsqcap$ -› [58,57] 57)
where $DC\ a\ b \equiv \lambda ab\ Q\ s.\ min\ (a\ ab\ Q\ s)\ (b\ ab\ Q\ s)$

AC is *angelic* choice between programs.

definition $AC :: 's\ prog \Rightarrow 's\ prog \Rightarrow 's\ prog$ (‹- $\bigsqcup$ -› [58,57] 57)
where $AC\ a\ b \equiv \lambda ab\ Q\ s.\ max\ (a\ ab\ Q\ s)\ (b\ ab\ Q\ s)$

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

definition $Embed :: 's\ trans \Rightarrow 's\ prog$
where $Embed\ t = (\lambda ab.\ t)$

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

definition $Mu :: ('s\ prog \Rightarrow 's\ prog) \Rightarrow 's\ prog$ (**binder** ‹μ› 50)
where $Mu(T) \equiv (\lambda ab.\ if\ ab\ then\ lfp\text{-}trans\ (\lambda t.\ T\ (Embed\ t)\ ab)$
$else\ gfp\text{-}trans\ (\lambda t.\ T\ (Embed\ t)\ ab))$

repeat expresses finite repetition

primrec
$repeat :: nat \Rightarrow 'a\ prog \Rightarrow 'a\ prog$
where
$repeat\ 0\ p = Skip\ |$
$repeat\ (Suc\ n)\ p = p\ ;;\ repeat\ n\ p$

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

definition $SetDC :: ('a \Rightarrow 's\ prog) \Rightarrow ('s \Rightarrow 'a\ set) \Rightarrow 's\ prog$
where $SetDC\ f\ S \equiv \lambda ab\ P\ s.\ Inf\ ((\lambda a.\ f\ a\ ab\ P\ s)\ `\ S\ s)$

syntax $-SetDC :: pttrn => ('s => 'a\ set) => 's\ prog => 's\ prog$
$(‹\bigsqcap -\in -./ -› 100)$
syntax-consts $-SetDC == SetDC$
translations $\bigsqcap x{\in}S.\ p == CONST\ SetDC\ (\%x.\ p)\ S$

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

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

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

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

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

This gives us something like let syntax

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

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

syntax *-PBind* :: *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*$) \Rightarrow$ *'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*)
$\langle M\ L\rangle$

syntax
-SetPC :: *ident* => $('s$ => $'a$ => *real*$)$ => *'s prog*
(‹*choose* - *at* -› [*66*,*66*]*66*)
syntax-consts
-SetPC $\rightleftharpoons$ *SetPC*
$\langle M\ L\rangle$

syntax
-set-dc :: *ident* => $('s$ => $'a$ *set*$)$ => *'s prog* (‹- :$\in$ -› [*66*,*66*]*66*)
syntax-consts
-set-dc $\rightleftharpoons$ *SetDC*
$\langle M\ L\rangle$

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

syntax

-set-dc-UNIV :: *ident* => *'s prog* (‹*any* -› [*66*]*66*)
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 = (λ*s*. *0*)
⟨*proof*⟩

lemma *eval-wlp-Abort*:
wlp Abort P = (λ*s*. *1*)
⟨*proof*⟩

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

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

lemma *eval-wp-Apply*:

wp (*Apply f*) *P* = *P o f*
⟨*proof*⟩

lemma *eval-wlp-Apply*:
wlp (*Apply f*) *P* = *P o f*
⟨*proof*⟩

lemma *eval-wp-Seq*:
wp (*a* ;; *b*) *P* = (*wp a o wp b*) *P*
⟨*proof*⟩

lemma *eval-wlp-Seq*:
wlp (*a* ;; *b*) *P* = (*wlp a o wlp b*) *P*
⟨*proof*⟩

lemma *eval-wp-PC*:
wp (*a* $\underset{Q}{\oplus}$ *b*) *P* = (λ*s*. *Q s* ∗ *wp a P s* + (*1* − *Q s*) ∗ *wp b P s*)
⟨*proof*⟩

lemma *eval-wlp-PC*:
wlp (*a* $\underset{Q}{\oplus}$ *b*) *P* = (λ*s*. *Q s* ∗ *wlp a P s* + (*1* − *Q s*) ∗ *wlp b P s*)
⟨*proof*⟩

lemma *eval-wp-DC*:
wp (*a* ⨅ *b*) *P* = (λ*s*. *min* (*wp a P s*) (*wp b P s*))
⟨*proof*⟩

lemma *eval-wlp-DC*:
wlp (*a* ⨅ *b*) *P* = (λ*s*. *min* (*wlp a P s*) (*wlp b P s*))
⟨*proof*⟩

lemma *eval-wp-AC*:
wp (*a* ⨆ *b*) *P* = (λ*s*. *max* (*wp a P s*) (*wp b P s*))
⟨*proof*⟩

lemma *eval-wlp-AC*:
wlp (*a* ⨆ *b*) *P* = (λ*s*. *max* (*wlp a P s*) (*wlp b P s*))
⟨*proof*⟩

lemma *eval-wp-Embed*:
wp (*Embed t*) = *t*
⟨*proof*⟩

lemma *eval-wlp-Embed*:
wlp (*Embed t*) = *t*
⟨*proof*⟩

lemma *eval-wp-SetDC*:
wp (*SetDC p S*) *R s* = *Inf* ((λ*a*. *wp* (*p a*) *R s*) ‘ *S s*)

⟨*proof*⟩

lemma *eval-wlp-SetDC*:
wlp (*SetDC p S*) *R s* = *Inf* ((λ*a*. *wlp* (*p a*) *R s*) ‘ *S s*)
⟨*proof*⟩

lemma *eval-wp-SetPC*:
wp (*SetPC f p*) *P* = (λ*s*. ∑ *a*∈*supp* (*p s*). *p s a* ∗ *wp* (*f a*) *P s*)
⟨*proof*⟩

lemma *eval-wlp-SetPC*:
wlp (*SetPC f p*) *P* = (λ*s*. ∑ *a*∈*supp* (*p s*). *p s a* ∗ *wlp* (*f a*) *P s*)
⟨*proof*⟩

lemma *eval-wp-Mu*:
wp (μ *t*. *T t*) = *lfp-trans* (λ*t*. *wp* (*T* (*Embed t*)))
⟨*proof*⟩

lemma *eval-wlp-Mu*:
wlp (μ *t*. *T t*) = *gfp-trans* (λ*t*. *wlp* (*T* (*Embed t*)))
⟨*proof*⟩

lemma *eval-wp-Bind*:
wp (*Bind g f*) = (λ*P s*. *wp* (*f* (*g s*)) *P s*)
⟨*proof*⟩

lemma *eval-wlp-Bind*:
wlp (*Bind g f*) = (λ*P s*. *wlp* (*f* (*g s*)) *P s*)
⟨*proof*⟩

Use simp add:wp_eval to fully unfold a program fragment

lemmas *wp-eval* = *eval-wp-Abort eval-wlp-Abort eval-wp-Skip eval-wlp-Skip*
eval-wp-Apply eval-wlp-Apply eval-wp-Seq eval-wlp-Seq
eval-wp-PC eval-wlp-PC eval-wp-DC eval-wlp-DC
eval-wp-AC eval-wlp-AC
eval-wp-Embed eval-wlp-Embed eval-wp-SetDC eval-wlp-SetDC
eval-wp-SetPC eval-wlp-SetPC eval-wp-Mu eval-wlp-Mu
eval-wp-Bind eval-wlp-Bind

lemma *Skip-Seq*:
Skip ;; *A* = *A*
⟨*proof*⟩

lemma *Seq-Skip*:
A ;; *Skip* = *A*
⟨*proof*⟩

Use these as simp rules to clear out Skips

lemmas *skip-simps* = *Skip-Seq Seq-Skip*

end

4.2 Healthiness

theory *Healthiness* **imports** *Embedding* **begin**

4.2.1 The Healthiness of the Embedding

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

lemma *healthy-wp-Abort*:
healthy (*wp Abort*)
⟨*proof*⟩

lemma *nearly-healthy-wlp-Abort*:
nearly-healthy (*wlp Abort*)
⟨*proof*⟩

lemma *healthy-wp-Skip*:
healthy (*wp Skip*)
⟨*proof*⟩

lemma *nearly-healthy-wlp-Skip*:
nearly-healthy (*wlp Skip*)
⟨*proof*⟩

lemma *healthy-wp-Seq*:
fixes *t*::'*s prog* **and** *u*
assumes *ht*: *healthy* (*wp t*) **and** *hu*: *healthy* (*wp u*)
shows *healthy* (*wp* (*t* ;; *u*))
⟨*proof*⟩

lemma *nearly-healthy-wlp-Seq*:
fixes *t*::'*s prog* **and** *u*
assumes *ht*: *nearly-healthy* (*wlp t*) **and** *hu*: *nearly-healthy* (*wlp u*)
shows *nearly-healthy* (*wlp* (*t* ;; *u*))
⟨*proof*⟩

lemma *healthy-wp-PC*:
fixes *f*::'*s prog*
assumes *hf*: *healthy* (*wp f*) **and** *hg*: *healthy* (*wp g*)
and *uP*: *unitary P*
shows *healthy* (*wp* (*f* $_P\oplus$ *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* $_P\oplus$ *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* ⊨ $\lambda s.$ *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* ⊨ *R* $\Longrightarrow$ *sound R* $\Longrightarrow$ *Q* ⊨ *R*
and *hb*: *healthy* (*wp body*)
and *sP*: *sound P*
and *sQ*: *sound Q*
shows *Q* ⊨ *lfp-exp* ($\lambda Q\ s.$ «*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* ($\lambda Q\ s.$ «*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*: $\bigwedge P$. *unitary P* $\Longrightarrow$ *unitary* (*t P*)
and *uP*: *unitary P*
shows *unitary* (*wlp* (*body* ;; *Embed t* « *G* »$\oplus$ *Skip*) *P*)
⟨*proof*⟩

lemma *wp-loop-step-sound*:
fixes *t u*::*'s trans*
assumes *hb*: *healthy* (*wp body*)
and *ht*: $\bigwedge P$. *sound P* $\Longrightarrow$ *sound* (*t P*)
and *sP*: *sound P*
shows *sound* (*wp* (*body* ;; *Embed t* « *G* »$\oplus$ *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* $\longrightarrow$ *body od*) *P* =
gfp-exp ($\lambda Q\ s$. «*G*» *s* $*$ *wlp body Q s* + «$\mathcal{N}$ *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* $\longrightarrow$ *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* $\longrightarrow$ *body od*) *P* =
lfp-exp ($\lambda Q\ s$. «*G*» *s* $*$ *wp body Q s* + «$\mathcal{N}$ *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* $\longrightarrow$ *body od*))
⟨*proof*⟩

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

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

Use 'simp add:healthy_intros' or 'blast intro:healthy_intros' as appropriate to discharge healthiness side-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-cts* :: *'s trans* ⇒ *bool*
where *bd-cts t* = (∀ *M*. (∀ *i*. (*M i* ⊩ *M* (*Suc i*)) ∧ *sound* (*M i*)) ⟶
(∃ *b*. ∀ *i*. *bounded-by b* (*M i*)) ⟶
t (*Sup-exp* (*range M*)) = *Sup-exp* (*range* (*t o M*)))

lemma *bd-ctsD*:
⟦ *bd-cts t*; ⋀*i*. *M i* ⊩ *M* (*Suc i*); ⋀*i*. *sound* (*M i*); ⋀*i*. *bounded-by b* (*M i*) ⟧ ⟹
t (*Sup-exp* (*range M*)) = *Sup-exp* (*range* (*t o M*))
⟨*proof*⟩

lemma *bd-ctsI*:
(⋀*b M*. (⋀*i*. *M i* ⊩ *M* (*Suc i*)) ⟹ (⋀*i*. *sound* (*M i*)) ⟹ (⋀*i*. *bounded-by b* (*M i*)) ⟹
t (*Sup-exp* (*range M*)) = *Sup-exp* (*range* (*t o M*))) ⟹ *bd-cts t*
⟨*proof*⟩

A generalised property for transformers of transformers.

definition *bd-cts-tr* :: (*'s trans* ⇒ *'s trans*) ⇒ *bool*

where *bd-cts-tr T* = (∀ *M*. (∀ *i*. *le-trans* (*M i*) (*M* (*Suc i*)) ∧ *feasible* (*M i*)) ⟶
equiv-trans (*T* (*Sup-trans* (*M* ‘ *UNIV*))) (*Sup-trans* ((*T o M*) ‘ *UNIV*)))

lemma *bd-cts-trD*:
⟦ *bd-cts-tr T*; ⋀*i*. *le-trans* (*M i*) (*M* (*Suc i*)); ⋀*i*. *feasible* (*M i*) ⟧ ⟹
equiv-trans (*T* (*Sup-trans* (*M* ‘ *UNIV*))) (*Sup-trans* ((*T o M*) ‘ *UNIV*))
⟨*proof*⟩

lemma *bd-cts-trI*:
(⋀*M*. (⋀*i*. *le-trans* (*M i*) (*M* (*Suc i*))) ⟹ (⋀*i*. *feasible* (*M i*)) ⟹
equiv-trans (*T* (*Sup-trans* (*M* ‘ *UNIV*))) (*Sup-trans* ((*T o M*) ‘ *UNIV*))) ⟹ *bd-cts-tr T*
⟨*proof*⟩

4.3.1 Continuity of Primitives

lemma *cts-wp-Abort*:
bd-cts (*wp* (*Abort*::′*s prog*))
⟨*proof*⟩

lemma *cts-wp-Skip*:
bd-cts (*wp Skip*)
⟨*proof*⟩

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

lemma *cts-wp-Bind*:
fixes *a*::′*a* ⇒ ′*s prog*
assumes *ca*: ⋀*s*. *bd-cts* (*wp* (*a* (*f s*)))
shows *bd-cts* (*wp* (*Bind f a*))
⟨*proof*⟩

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 prog*
assumes *ca*: *bd-cts* (*wp a*)
and *cb*: *bd-cts* (*wp b*)
and *ha*: *healthy* (*wp a*)
and *hb*: *healthy* (*wp b*)
shows *bd-cts* (*wp* (*a* ⨅ *b*))
⟨*proof*⟩

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

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

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

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

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

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

lemma *cts-wp-SetDC*:
fixes $a::'a \Rightarrow 's\ prog$
assumes ca: $\bigwedge x\ s.\ x \in S\ s \Longrightarrow bd\text{-}cts\ (wp\ (a\ x))$
and ha: $\bigwedge x\ s.\ x \in S\ s \Longrightarrow healthy\ (wp\ (a\ x))$
and fS: $\bigwedge s.\ finite\ (S\ s)$
and neS: $\bigwedge s.\ S\ s \neq \{\}$
shows $bd\text{-}cts\ (wp\ (SetDC\ a\ S))$

⟨*proof*⟩

lemma *cts-wp-repeat*:
bd-cts (*wp a*) ⟹ *healthy* (*wp a*) ⟹ *bd-cts* (*wp* (*repeat n a*))
⟨*proof*⟩

lemma *cts-wp-Embed*:
bd-cts t ⟹ *bd-cts* (*wp* (*Embed t*))
⟨*proof*⟩

4.3.2 Continuity of a Single Loop Step

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

lemma *cts-wp-loopstep*:
fixes *body*::*'s prog*
assumes *hb*: *healthy* (*wp body*)
and *cb*: *bd-cts* (*wp body*)
shows *bd-cts-tr* (λ*x*. *wp* (*body* ;; *Embed x* « *G* »⊕ *Skip*)) (**is** *bd-cts-tr ?F*)
⟨*proof*⟩

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-trans* (λ*Q s*. « *G* » *s* ∗ *wp body Q s* + « *N G* » *s* ∗ *P s*)
⟨*proof*⟩

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-exp* (λ*Q s*. «*G*» *s* ∗ *wp body Q s* + «*N G*» *s* ∗ *P s*) =
(λ*s*. «*G*» *s* ∗ *wp body* (*lfp-exp* (λ*Q s*. «*G*» *s* ∗ *wp body Q s* + «*N G*» *s* ∗ *P s*)) *s* +

«$\mathcal{N}$ 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* (λ*s*. «*G*» *s* ∗ *wp body Q s* + «$\mathcal{N}$ *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* (λ*Q s*. «*G*» *s* ∗ *wp body Q s* + «$\mathcal{N}$ *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 $(\lambda i.\ iterates\ body\ G\ i\ P\ s) \longrightarrow wp\ (do\ G \longrightarrow body\ od)\ P\ s$
⟨*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* ⟶ *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*))

⟨*proof*⟩

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

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

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

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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

lemma *sublinear-wp-repeat*:
⟦ *sublinear* (*wp p*); *healthy* (*wp p*) ⟧ ⟹ *sublinear* (*wp* (*repeat n p*))
⟨*proof*⟩

lemma *sublinear-wp-Bind*:

$[\![\bigwedge s.\ \mathit{sublinear}\ (\mathit{wp}\ (a\ (f\ s)))]\!] \Longrightarrow \mathit{sublinear}\ (\mathit{wp}\ (\mathit{Bind}\ f\ a))$
$\langle proof \rangle$

4.5.2 Sublinearity for Loops

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

lemma *sub-distrib-wp-loop*:
fixes *body*::*'s prog*
assumes *sdb*: *sub-distrib* (*wp body*)
and *hb*: *healthy* (*wp body*)
and *nhb*: *nearly-healthy* (*wlp body*)
shows *sub-distrib* (*wp* (*do G* ⟶ *body od*))
$\langle proof \rangle$

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*)
$\langle proof \rangle$

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*))
$\langle proof \rangle$

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*))
$\langle proof \rangle$

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

sublinear-wp-DC
sublinear-wp-SetPC
sublinear-wp-SetDC
sublinear-wp-Embed
sublinear-wp-repeat
sublinear-wp-Bind
sublinear-wp-loop

end

4.6 Determinism

theory *Determinism* **imports** *WellDefined* **begin**

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

4.6.1 Additivity

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

wlp Abort is not additive.

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

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

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

lemma *additive-wp-PC*:
⟦ *additive* (*wp a*); *additive* (*wp b*) ⟧ ⟹ *additive* (*wp* (*a* $_P\oplus$ *b*))
⟨*proof*⟩

DC is not additive.

lemma *additive-wp-SetPC*:
⟦ ⋀*x s*. *x* ∈ *supp* (*p s*) ⟹ *additive* (*wp* (*a x*)); ⋀*s*. *finite* (*supp* (*p s*)) ⟧ ⟹

additive (wp (SetPC a p))
⟨*proof*⟩

lemma *additive-wp-Bind*:
⟦ ⋀*x*. *additive (wp (a (f x)))* ⟧ ⟹ *additive (wp (Bind f a))*
⟨*proof*⟩

lemma *additive-wp-Embed*:
⟦ *additive t* ⟧ ⟹ *additive (wp (Embed t))*
⟨*proof*⟩

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

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

4.6.2 Maximality

lemma *max-wp-Skip*:
maximal (wp Skip)
⟨*proof*⟩

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

lemma *max-wp-Seq*:
⟦ *maximal (wp a)*; *maximal (wp b)* ⟧ ⟹ *maximal (wp (a ;; b))*
⟨*proof*⟩

lemma *max-wp-PC*:
⟦ *maximal (wp a)*; *maximal (wp b)* ⟧ ⟹ *maximal (wp (a* $_P\oplus$ *b))*
⟨*proof*⟩

lemma *max-wp-DC*:
⟦ *maximal (wp a)*; *maximal (wp b)* ⟧ ⟹ *maximal (wp (a* ⨅ *b))*
⟨*proof*⟩

lemma *max-wp-SetPC*:
⟦ ⋀*s a*. *a* ∈ *supp (P s)* ⟹ *maximal (wp (p a))*; ⋀*s*. (∑ *a*∈*supp (P s)*. *P s a*) = *1* ⟧ ⟹
maximal (wp (SetPC p P))
⟨*proof*⟩

lemma *max-wp-SetDC*:
fixes p::'$a \Rightarrow$'s *prog*
assumes *mp*: $\bigwedge s\ a.\ a \in S\ s \Longrightarrow$ *maximal* (*wp* (*p a*))
and *ne*: $\bigwedge s.\ S\ s \neq \{\}$
shows *maximal* (*wp* (*SetDC p S*))
⟨*proof*⟩

lemma *max-wp-Embed*:
maximal t $\Longrightarrow$ *maximal* (*wp* (*Embed t*))
⟨*proof*⟩

lemma *max-wp-repeat*:
maximal (*wp a*) $\Longrightarrow$ *maximal* (*wp* (*repeat n a*))
⟨*proof*⟩

lemma *max-wp-Bind*:
assumes *ma*: $\bigwedge s.$ *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*: $\lambda s.\ 1 \Vdash t\ (\lambda s.\ 1)$
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*) $\Longrightarrow$ *determ* (*wp b*) $\Longrightarrow$ *well-def b* $\Longrightarrow$ *determ* (*wp* (*a* ;; *b*))
⟨*proof*⟩

lemma *det-wp-PC*:

determ (*wp a*) $\Longrightarrow$ *determ* (*wp b*) $\Longrightarrow$ *determ* (*wp* (*a* $_P\oplus$ *b*))
⟨*proof*⟩

lemma *det-wp-SetPC*:
($\bigwedge x\ s.\ x \in supp\ (p\ s) \Longrightarrow determ\ (wp\ (a\ x))) \Longrightarrow$
($\bigwedge s.\ finite\ (supp\ (p\ s))) \Longrightarrow$
($\bigwedge s.\ sum\ (p\ s)\ (supp\ (p\ s)) = 1) \Longrightarrow$
determ (*wp* (*SetPC a p*))
⟨*proof*⟩

lemma *det-wp-Bind*:
($\bigwedge x.\ determ\ (wp\ (a\ (f\ x)))) \Longrightarrow determ\ (wp\ (Bind\ f\ a))$
⟨*proof*⟩

lemma *det-wp-Embed*:
determ t $\Longrightarrow$ *determ* (*wp* (*Embed t*))
⟨*proof*⟩

lemma *det-wp-repeat*:
determ (*wp a*) $\Longrightarrow$ *well-def a* $\Longrightarrow$ *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
wp-under-wlp :: *'s prog* ⇒ *bool*
where
wp-under-wlp prog ≡ ∀ *P*. *unitary P* ⟶ *wp prog P* ⊨ *wlp prog P*

lemma *wp-under-wlpI*[*intro*]:
⟦ ⋀*P*. *unitary P* ⟹ *wp prog P* ⊨ *wlp prog P* ⟧ ⟹ *wp-under-wlp prog*
⟨*proof*⟩

lemma *wp-under-wlpD*[*dest*]:
⟦ *wp-under-wlp prog*; *unitary P* ⟧ ⟹ *wp prog P* ⊨ *wlp prog P*
⟨*proof*⟩

lemma *wp-under-le-trans*:
wp-under-wlp a ⟹ *le-utrans* (*wp a*) (*wlp a*)
⟨*proof*⟩

lemma *wp-under-wlp-Abort*:
wp-under-wlp Abort
⟨*proof*⟩

lemma *wp-under-wlp-Skip*:
wp-under-wlp Skip
⟨*proof*⟩

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

lemma *wp-under-wlp-Seq*:
assumes *h-wlp-a*: *nearly-healthy* (*wlp a*)
and *h-wp-b*: *healthy* (*wp b*)
and *h-wlp-b*: *nearly-healthy* (*wlp b*)
and *wp-u-a*: *wp-under-wlp a*
and *wp-u-b*: *wp-under-wlp b*
shows *wp-under-wlp* (*a* ;; *b*)
⟨*proof*⟩

lemma *wp-under-wlp-PC*:
assumes *h-wp-a*: *healthy* (*wp a*)
and *h-wlp-a*: *nearly-healthy* (*wlp a*)
and *h-wp-b*: *healthy* (*wp b*)
and *h-wlp-b*: *nearly-healthy* (*wlp b*)
and *wp-u-a*: *wp-under-wlp a*
and *wp-u-b*: *wp-under-wlp b*

and *uP*: *unitary P*
shows *wp-under-wlp* (*a* $_P\oplus$ *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* $\bigsqcap$ *b*)
⟨*proof*⟩

lemma *wp-under-wlp-SetPC*:
assumes *wp-u-f*: $\bigwedge s\ a.\ a \in supp\ (P\ s) \Longrightarrow$ *wp-under-wlp* (*f a*)
and *nP*: $\bigwedge s\ a.\ a \in supp\ (P\ s) \Longrightarrow 0 \leq P\ s\ a$
shows *wp-under-wlp* (*SetPC f P*)
⟨*proof*⟩

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

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

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

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

lemma *wp-under-wlp-Bind*:
⟦ $\bigwedge s$. *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* $_P\oplus$ *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* $\bigsqcap$ *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*)) ≤ 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
wlp-inv G body I $\longleftrightarrow$ $(\forall s.\ «G»\ s * I\ s \leq wlp\ body\ I\ s)$

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

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

For standard invariants, the multiplication reduces to conjunction.

lemma *wp-inv-stdD*:
assumes *inv*: *wp-inv G body «I»*
and *hb*: *healthy* (*wp body*)
shows $«G»\ \&\&\ «I» \Vdash wp\ body\ «I»$
⟨*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 $I \leq wlp\ do\ G \longrightarrow body\ od\ (\lambda s.\ «\mathcal{N}\ G»\ s * I\ s)$
(**is** $I \leq wlp\ do\ G \longrightarrow body\ od\ ?P$)
⟨*proof*⟩

4.8.3 Total Correctness

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

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

4.8.4 Unfolding

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

shows *wp* (*do G* ⟶ *body od*) *P* =
(λ*s*. «$\mathcal{N}$ *G*» *s* ∗ *P s* + «*G*» *s* ∗ *wp body* (*wp* (*do G* ⟶ *body od*) *P*) *s*)
⟨*proof*⟩

lemma *wp-loop-nguard*:
⟦ *healthy* (*wp body*); *sound P*; ¬ *G s* ⟧ ⟹ *wp do G* ⟶ *body od P s* = *P s*
⟨*proof*⟩

lemma *wp-loop-guard*:
⟦ *healthy* (*wp body*); *sound P*; *G s* ⟧ ⟹
wp do G ⟶ *body od P s* = *wp* (*body* ;; *do G* ⟶ *body od*) *P s*
⟨*proof*⟩

end

4.9 The Algebra of pGCL

theory *Algebra* **imports** *WellDefined* **begin**

Programs in pGCL have a rich algebraic structure, largely mirroring that for GCL. We show that programs form a lattice under refinement, with $a \sqcap b$ and $a \sqcup b$ 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* ⇒ *'s prog* ⇒ *bool* (**infix** ‹⊑› *70*)
where
prog ⊑ *prog'* ≡ ∀ *P*. *sound P* ⟶ *wp prog P* ⊨ *wp prog' P*

lemma *refinesI*[*intro*]:
⟦ ⋀*P*. *sound P* ⟹ *wp prog P* ⊨ *wp prog' P* ⟧ ⟹ *prog* ⊑ *prog'*
⟨*proof*⟩

lemma *refinesD*[*dest*]:
⟦ *prog* ⊑ *prog'*; *sound P* ⟧ ⟹ *wp prog P* ⊨ *wp prog' P*
⟨*proof*⟩

The equivalence relation below will turn out to be that induced by refinement. It is also the application of *equiv-trans* to the weakest precondition.

definition
pequiv :: *'s prog* ⇒ *'s prog* ⇒ *bool* (**infix** ‹≃› *70*)
where

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

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*]:
$a \sqcap b = b \sqcap a$
⟨*proof*⟩

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

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

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

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

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

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

lemma *PC-idem*:
$a \,{}_p\!\oplus\, a = a$
$\langle proof \rangle$

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

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

Laws relating demonic choice and refinement

lemma *left-refines-DC*:
$(a \sqcap b) \sqsubseteq a$
$\langle proof \rangle$

lemma *right-refines-DC*:
$(a \sqcap b) \sqsubseteq b$
$\langle proof \rangle$

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

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

Laws relating angelic choice and refinement

lemma *left-refines-AC*:
$a \sqsubseteq (a \sqcup b)$
$\langle proof \rangle$

lemma *right-refines-AC*:
$b \sqsubseteq (a \sqcup b)$
$\langle proof \rangle$

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

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

Laws depending on the arithmetic of $a \,{}_p\!\oplus\, b$ and $a \bigsqcap b$ together

lemma *DC-refines-PC*:
assumes *unit*: *unitary p*
shows $(a \bigsqcap b) \sqsubseteq (a \,{}_p\!\oplus\, b)$
⟨*proof*⟩

Laws depending on the arithmetic of $a \,{}_p\!\oplus\, b$ and $a \bigsqcup b$ together

lemma *PC-refines-AC*:
assumes *unit*: *unitary p*
shows $(a \,{}_p\!\oplus\, b) \sqsubseteq (a \bigsqcup b)$
⟨*proof*⟩

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

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

Laws Involving Refinement and Equivalence

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

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

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

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

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

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

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

Refinement induces equivalence by antisymmetry:

lemma *pequiv-antisym*:
$[\![a \sqsubseteq b; b \sqsubseteq a]\!] \Longrightarrow a \simeq b$
$\langle proof \rangle$

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

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

4.9.3 Deterministic Programs are Maximal

Any sub-additive refinement of a deterministic program is in fact an equivalence. Deterministic programs are thus maximal (under the refinement order) among sub-additive programs.

lemma *refines-determ*:
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.

$\langle proof \rangle$

4.9.4 The Algebraic Structure of Refinement

Well-defined programs form a half-bounded semilattice under refinement, where *Abort* is bottom, and $a \sqcap b$ is *inf*. There is no unique top element, but all fully-deterministic programs are maximal.

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

quotient-type *'s program* =
's prog / *partial* : $\lambda a\ b.\ a \simeq b \wedge$ *well-def a* $\wedge$ *well-def b*

⟨*proof*⟩

instantiation *program* :: (*type*) *semilattice-inf* **begin**
lift-definition
less-eq-program :: *'a program* ⇒ *'a program* ⇒ *bool* **is** *refines*
⟨*proof*⟩

lift-definition
less-program :: *'a program* ⇒ *'a program* ⇒ *bool*
is λ*a b*. *a* ⊑ *b* ∧ ¬ *b* ⊑ *a*
⟨*proof*⟩

lift-definition
inf-program :: *'a program* ⇒ *'a program* ⇒ *'a program* **is** *DC*
⟨*proof*⟩

instance
⟨*proof*⟩
end

instantiation *program* :: (*type*) *bot* **begin**
lift-definition
bot-program :: *'a program* **is** *Abort*
⟨*proof*⟩

instance ⟨*proof*⟩
end

lemma *eq-det*: ⋀*a b*::*'s prog*. ⟦ *a* ≃ *b*; *determ* (*wp a*) ⟧ ⟹ *determ* (*wp b*)
⟨*proof*⟩

lift-definition
pdeterm :: *'s program* ⇒ *bool*
is λ*a*. *determ* (*wp a*)
⟨*proof*⟩

lemma *determ-maximal*:
⟦ *pdeterm a*; *a* ≤ *x* ⟧ ⟹ *a* = *x*
⟨*proof*⟩

4.9.5 Data Refinement

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

definition
drefines :: (*'b* ⇒ *'a*) ⇒ (*'b* ⇒ *bool*) ⇒ *'a prog* ⇒ *'b prog* ⇒ *bool*
where

$$\mathit{drefines}\ \varphi\ G\ A\ B \equiv \forall\, P\ Q.\ (\mathit{unitary}\ P \wedge \mathit{unitary}\ Q \wedge (P \vDash \mathit{wp}\ A\ Q)) \longrightarrow (\langle\!\langle G \rangle\!\rangle\ \&\&\ (P\ o\ \varphi) \vDash \mathit{wp}\ B\ (Q\ o\ \varphi))$$

lemma *drefinesD*[*dest*]:
$[\![\ \mathit{drefines}\ \varphi\ G\ A\ B; \mathit{unitary}\ P; \mathit{unitary}\ Q; P \vDash \mathit{wp}\ A\ Q\ ]\!] \Longrightarrow$
$\langle\!\langle G \rangle\!\rangle\ \&\&\ (P\ o\ \varphi) \vDash \mathit{wp}\ B\ (Q\ o\ \varphi)$
$\langle proof \rangle$

We can alternatively use G as an assumption:

lemma *drefinesD2*:
assumes *dr*: $\mathit{drefines}\ \varphi\ G\ A\ B$
and *uP*: $\mathit{unitary}\ P$
and *uQ*: $\mathit{unitary}\ Q$
and *wpA*: $P \vDash \mathit{wp}\ A\ Q$
and *G*: $G\ s$
shows $(P\ o\ \varphi)\ s \leq \mathit{wp}\ B\ (Q\ o\ \varphi)\ s$
$\langle proof \rangle$

This additional form is sometimes useful:

lemma *drefinesD3*:
assumes *dr*: $\mathit{drefines}\ \varphi\ G\ a\ b$
and *G*: $G\ s$
and *uQ*: $\mathit{unitary}\ Q$
and *wa*: $\mathit{well\text{-}def}\ a$
shows $\mathit{wp}\ a\ Q\ (\varphi\ s) \leq \mathit{wp}\ b\ (Q\ o\ \varphi)\ s$
$\langle proof \rangle$

lemma *drefinesI*[*intro*]:
$[\![\ \bigwedge P\ Q.\ [\![\ \mathit{unitary}\ P; \mathit{unitary}\ Q; P \vDash \mathit{wp}\ A\ Q\ ]\!] \Longrightarrow$
$\langle\!\langle G \rangle\!\rangle\ \&\&\ (P\ o\ \varphi) \vDash \mathit{wp}\ B\ (Q\ o\ \varphi)\ ]\!] \Longrightarrow$
$\mathit{drefines}\ \varphi\ G\ A\ B$
$\langle proof \rangle$

Use G as an assumption, when showing refinement:

lemma *drefinesI2*:
fixes $A :: 'a\ \mathit{prog}$
and $B :: 'b\ \mathit{prog}$
and $\varphi :: 'b \Rightarrow 'a$
and $G :: 'b \Rightarrow \mathit{bool}$
assumes *wB*: $\mathit{well\text{-}def}\ B$
and *withAs*:
$\bigwedge P\ Q\ s.\ [\![\ \mathit{unitary}\ P; \mathit{unitary}\ Q;$
$G\ s; P \vDash \mathit{wp}\ A\ Q\ ]\!] \Longrightarrow (P\ o\ \varphi)\ s \leq \mathit{wp}\ B\ (Q\ o\ \varphi)\ s$
shows $\mathit{drefines}\ \varphi\ G\ A\ B$
$\langle proof \rangle$

lemma *dr-strengthen-guard*:
fixes $a :: 's\ \mathit{prog}$ **and** $b :: 't\ \mathit{prog}$
assumes *fg*: $\bigwedge s.\ F\ s \Longrightarrow G\ s$

and *drab*: *drefines* φ *G a b*
shows *drefines* φ *F a b*
⟨*proof*⟩

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

definition
pcorres :: (*'b* ⇒ *'a*) ⇒ (*'b* ⇒ *bool*) ⇒ *'a prog* ⇒ *'b prog* ⇒ *bool*
where
pcorres φ *G A B* ⟷
(∀ *Q*. *unitary Q* ⟶ «*G*» && (*wp A Q o* φ) = «*G*» && *wp B* (*Q o* φ))

lemma *pcorresI*:
⟦ ⋀*Q*. *unitary Q* ⟹ «*G*» && (*wp A Q o* φ) = «*G*» && *wp B* (*Q o* φ) ⟧ ⟹
pcorres φ *G A B*
⟨*proof*⟩

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

lemma *pcorresI2*[*intro*]:
fixes *A*::*'a prog* **and** *B*::*'b prog*
assumes *withG*: ⋀*Q s*. ⟦ *unitary Q*; *G s* ⟧ ⟹ *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*
⟨*proof*⟩

lemma *pcorresD*:
⟦ *pcorres* φ *G A B*; *unitary Q* ⟧ ⟹ «*G*» && (*wp A Q o* φ) = «*G*» && *wp B* (*Q o* φ)
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⊑ *b* **and** *wb*: *well-def b*
shows *drefines* (λ*s*. *s*) *G a b*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*: ⋀*Q*. *unitary Q* ⟹ «*I*» && *wp B Q* = *wp B* («*H*» && *Q*)
and *p1p3*: ⋀*s*. *G s* ⟹ *I s*
shows *pcorres φ G* (*A*;;*C*) (*B*;;*D*)
⟨*proof*⟩

4.9.8 Structural Rules for Data Refinement

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

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

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

lemma *dr-repeat*:
fixes φ :: $'a \Rightarrow 'b$
assumes *dr-ab*: *drefines* φ *G a b*
and *Gpr*: «*G*» ⊨ *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*)
⟨*proof*⟩

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*) ⊨ *wp Abort Q*
⟨*proof*⟩

lemma *wlp-Abort*:
(λ*s*. *1*) ⊨ *wlp Abort Q*
⟨*proof*⟩

lemma *wp-Skip*:
P ⊨ *wp Skip P*
⟨*proof*⟩

lemma *wlp-Skip*:
P ⊨ *wlp Skip P*
⟨*proof*⟩

lemma *wp-Apply*:
Q o f ⊨ *wp* (*Apply f*) *Q*
⟨*proof*⟩

lemma *wlp-Apply*:
Q o f ⊨ *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*: $0 \leq p$ **and** *bp*: $p \leq 1$
shows $(\lambda s.\ p * P\ s + (1 - p) * Q\ s) \Vdash (a\ {}_{(\lambda s.\ p)}\!\oplus\ b)\ ab\ R$
⟨*proof*⟩

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

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

lemma *wp-DC*:

$(\lambda s.\ min\ (wp\ a\ Q\ s)\ (wp\ b\ Q\ s)) \vDash wp\ (a \sqcap b)\ Q$
⟨*proof*⟩

lemma *wlp-DC*:
$(\lambda s.\ min\ (wlp\ a\ Q\ s)\ (wlp\ b\ Q\ s)) \vDash wlp\ (a \sqcap b)\ Q$
⟨*proof*⟩

Combining annotations for both branches:

lemma *DC-split*:
fixes $a::'s\ prog$ **and** b
assumes *wpa*: $P \vDash a\ ab\ R$
and *wpb*: $Q \vDash b\ ab\ R$
shows $(\lambda s.\ min\ (P\ s)\ (Q\ s)) \vDash (a \sqcap b)\ ab\ R$
⟨*proof*⟩

lemma *wp-DC-split*:
$[\![\ P \vDash wp\ prog\ R;\ Q \vDash wp\ prog'\ R\ ]\!] \Longrightarrow$
$(\lambda s.\ min\ (P\ s)\ (Q\ s)) \vDash wp\ (prog \sqcap prog')\ R$
⟨*proof*⟩

lemma *wlp-DC-split*:
$[\![\ P \vDash wlp\ prog\ R;\ Q \vDash wlp\ prog'\ R\ ]\!] \Longrightarrow$
$(\lambda s.\ min\ (P\ s)\ (Q\ s)) \vDash wlp\ (prog \sqcap prog')\ R$
⟨*proof*⟩

lemma *wp-DC-split-same*:
$[\![\ P \vDash wp\ prog\ Q;\ P \vDash wp\ prog'\ Q\ ]\!] \Longrightarrow P \vDash wp\ (prog \sqcap prog')\ Q$
⟨*proof*⟩

lemma *wlp-DC-split-same*:
$[\![\ P \vDash wlp\ prog\ Q;\ P \vDash wlp\ prog'\ Q\ ]\!] \Longrightarrow P \vDash wlp\ (prog \sqcap prog')\ Q$
⟨*proof*⟩

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 $(\lambda s.\ \sum x \in supp\ (p\ s).\ p\ s\ x * P\ x\ s) \vDash SetPC\ f\ p\ ab\ Q$
⟨*proof*⟩

lemma *wp-SetPC-split*:
$[\![\ \bigwedge x\ s.\ x \in supp\ (p\ s) \Longrightarrow P\ x \vDash wp\ (f\ x)\ Q;\ \bigwedge s.\ nneg\ (p\ s)\ ]\!] \Longrightarrow$
$(\lambda s.\ \sum x \in supp\ (p\ s).\ p\ s\ x * P\ x\ s) \vDash wp\ (SetPC\ f\ p)\ Q$
⟨*proof*⟩

lemma *wlp-SetPC-split*:
$[\![\ \bigwedge x\ s.\ x \in supp\ (p\ s) \Longrightarrow P\ x \vDash wlp\ (f\ x)\ Q;\ \bigwedge s.\ nneg\ (p\ s)\ ]\!] \Longrightarrow$
$(\lambda s.\ \sum x \in supp\ (p\ s).\ p\ s\ x * P\ x\ s) \vDash wlp\ (SetPC\ f\ p)\ Q$

⟨*proof*⟩

lemma *wp-SetDC-split*:
⟦ ⋀*s x*. *x* ∈ *S s* ⟹ *P* ⊩ *wp* (*f x*) *Q*; ⋀*s*. *S s* ≠ {} ⟧ ⟹
P ⊩ *wp* (*SetDC f S*) *Q*
⟨*proof*⟩

lemma *wlp-SetDC-split*:
⟦ ⋀*s x*. *x* ∈ *S s* ⟹ *P* ⊩ *wlp* (*f x*) *Q*; ⋀*s*. *S s* ≠ {} ⟧ ⟹
P ⊩ *wlp* (*SetDC f S*) *Q*
⟨*proof*⟩

lemma *wp-SetDC*:
assumes *wp*: ⋀*s x*. *x* ∈ *S s* ⟹ *P x* ⊩ *wp* (*f x*) *Q*
and *ne*: ⋀*s*. *S s* ≠ {}
and *sP*: ⋀*x*. *sound* (*P x*)
shows (λ*s*. *Inf* ((λ*x*. *P x s*) ‘ *S s*)) ⊩ *wp* (*SetDC f S*) *Q*
⟨*proof*⟩

lemma *wlp-SetDC*:
assumes *wp*: ⋀*s x*. *x* ∈ *S s* ⟹ *P x* ⊩ *wlp* (*f x*) *Q*
and *ne*: ⋀*s*. *S s* ≠ {}
and *sP*: ⋀*x*. *sound* (*P x*)
shows (λ*s*. *Inf* ((λ*x*. *P x s*) ‘ *S s*)) ⊩ *wlp* (*SetDC f S*) *Q*
⟨*proof*⟩

lemma *wp-Embed*:
P ⊩ *t Q* ⟹ *P* ⊩ *wp* (*Embed t*) *Q*
⟨*proof*⟩

lemma *wlp-Embed*:
P ⊩ *t Q* ⟹ *P* ⊩ *wlp* (*Embed t*) *Q*
⟨*proof*⟩

lemma *wp-Bind*:
⟦ ⋀*s*. *P s* ≤ *wp* (*a* (*f s*)) *Q s* ⟧ ⟹ *P* ⊩ *wp* (*Bind f a*) *Q*
⟨*proof*⟩

lemma *wlp-Bind*:
⟦ ⋀*s*. *P s* ≤ *wlp* (*a* (*f s*)) *Q s* ⟧ ⟹ *P* ⊩ *wlp* (*Bind f a*) *Q*
⟨*proof*⟩

lemma *wp-repeat*:
⟦ *P* ⊩ *wp a Q*; *Q* ⊩ *wp* (*repeat n a*) *R*;
well-def a; *sound Q*; *sound R* ⟧ ⟹ *P* ⊩ *wp* (*repeat* (*Suc n*) *a*) *R*
⟨*proof*⟩

lemma *wlp-repeat*:
⟦ *P* ⊩ *wlp a Q*; *Q* ⊩ *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*: (λ*s*. *1*) ⊨ *wp s* (λ*s*. *1*)
shows (λ*s*. *1*) ⊨ *wp* (*r* ;; *s*) (λ*s*. *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* (λ*s*. *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 $\lambda s.\ 1 \Vdash wp\ do\ G \longrightarrow body\ od\ (\lambda s.\ 1)$
⟨*proof*⟩

This version allows progress to depend on an invariant. Termination is then determined by the invariant's value in the initial state.

lemma *loop-term-nat-measure*:
fixes $m :: 's \Rightarrow nat$ **and** $body :: 's\ prog$
assumes *wb*: *well-def body*
and *guard*: $\bigwedge s.\ m\ s = 0 \longrightarrow \neg\ G\ s$
and *variant*: $\bigwedge n.\ «\lambda s.\ m\ s = Suc\ n» \mathbin{\&\&} «I» \Vdash wp\ body\ «\lambda s.\ m\ s = n»$
and *inv*: *wp-inv G body «I»*
shows $«I» \Vdash wp\ do\ G \longrightarrow body\ od\ (\lambda s.\ 1)$
⟨*proof*⟩

4.11.3 Probabilistic Termination

Any loop that has a non-zero chance of terminating after each step terminates with probability 1.

lemma *termination-0-1*:
fixes $body :: 's\ prog$
assumes *wb*: *well-def body*
— The loop terminates in one step with nonzero probability
and *onestep*: $(\lambda s.\ p) \Vdash wp\ body\ «\mathcal{N}\ G»$
and *nzp*: $0 < p$
— The body is maximal i.e. it terminates absolutely.
and *mb*: *maximal* (*wp body*)
shows $\lambda s.\ 1 \Vdash wp\ do\ G \longrightarrow body\ od\ (\lambda s.\ 1)$
⟨*proof*⟩

end

4.12 Automated Reasoning

theory *Automation* **imports** *StructuredReasoning*
begin

This theory serves as a container for automated reasoning tactics for pGCL, implemented in ML. At present, there is a basic verification condition generator (VCG).

named-theorems *wd*
theorems to automatically establish well−definedness
named-theorems *pwp-core*
core probabilistic wp rules, for evaluating primitive terms

named-theorems *pwp*
user−supplied probabilistic wp rules
named-theorems *pwlp*
user−supplied probabilistic wlp rules

$\langle M\,L \rangle$

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*: $\bigwedge x.\ x \notin S \Longrightarrow f\ x = 0$
shows *sum f S* = *sum f UNIV*
⟨*proof*⟩

lemma *cInf-mono*:
fixes *A*::*'a*::*conditionally-complete-lattice set*
assumes *lower*: $\bigwedge b.\ b \in B \Longrightarrow \exists\, a{\in}A.\ a \leq b$
and *bounded*: $\bigwedge a.\ a \in A \Longrightarrow c \leq a$
and *ne*: $B \neq \{\}$
shows *Inf A* ≤ *Inf B*
⟨*proof*⟩

lemma *max-distrib*:
fixes *c*::*real*
assumes *nn*: $0 \leq c$
shows $c * max\ a\ b = max\ (c * a)\ (c * b)$
⟨*proof*⟩

lemma *mult-div-mono-left*:
fixes *c*::*real*
assumes *nnc*: $0 \leq c$ **and** *nzc*: $c \neq 0$
and *inv*: $a \leq inverse\ c * b$
shows $c * a \leq b$
⟨*proof*⟩

lemma *mult-div-mono-right*:
fixes *c*::*real*
assumes *nnc*: $0 \leq c$ **and** *nzc*: $c \neq 0$
and *inv*: $inverse\ c * a \leq b$
shows $a \leq c * b$
⟨*proof*⟩

lemma *min-distrib*:
fixes *c*::*real*
assumes *nnc*: $0 \leq c$
shows $c * min\ a\ b = min\ (c * a)\ (c * b)$
⟨*proof*⟩

lemma *finite-set-least*:
fixes *S*::'*a*::*linorder set*
assumes *finite*: *finite S*
and *ne*: $S \neq \{\}$
shows $\exists x{\in}S.\ \forall y{\in}S.\ x \leq y$
⟨*proof*⟩

lemma *cSup-add*:
fixes *c*::*real*
assumes *ne*: $S \neq \{\}$
and *bS*: $\bigwedge x.\ x{\in}S \Longrightarrow x \leq b$
shows $Sup\ S + c = Sup\ \{x + c\ |x.\ x \in S\}$
⟨*proof*⟩

lemma *cSup-mult*:
fixes *c*::*real*
assumes *ne*: $S \neq \{\}$
and *bS*: $\bigwedge x.\ x{\in}S \Longrightarrow x \leq b$
and *nnc*: $0 \leq c$
shows $c * Sup\ S = Sup\ \{c * x\ |x.\ x \in S\}$
⟨*proof*⟩

lemma *closure-contains-Sup*:
fixes *S* :: *real set*
assumes *neS*: $S \neq \{\}$ **and** *bS*: $\forall x{\in}S.\ x \leq B$
shows $Sup\ S \in closure\ S$
⟨*proof*⟩

lemma *tendsto-min*:
fixes *x y*::*real*
assumes *ta*: $a \longrightarrow x$
and *tb*: $b \longrightarrow y$
shows $(\lambda i.\ min\ (a\ i)\ (b\ i)) \longrightarrow min\ x\ y$
⟨*proof*⟩

definition *supp* :: $('s \Rightarrow real) \Rightarrow 's\ set$
where $supp\ f = \{x.\ f\ x \neq 0\}$

definition *dist-remove* :: $('s \Rightarrow real) \Rightarrow 's \Rightarrow 's \Rightarrow real$
where *dist-remove p x* = $(\lambda y.$ *if y=x then 0 else p y / (1 − p x*$))$

lemma *supp-dist-remove*:

$p\ x \neq 0 \Longrightarrow p\ x \neq 1 \Longrightarrow supp\ (dist\text{-}remove\ p\ x) = supp\ p - \{x\}$
⟨*proof*⟩

lemma *supp-empty*:
$supp\ f = \{\} \Longrightarrow f\ x = 0$
⟨*proof*⟩

lemma *nsupp-zero*:
$x \notin supp\ f \Longrightarrow f\ x = 0$
⟨*proof*⟩

lemma *sum-supp*:
fixes $f :: 'a :: finite \Rightarrow real$
shows $sum\ f\ (supp\ f) = sum\ f\ UNIV$
⟨*proof*⟩

4.13.1 Truncated Subtraction

definition
$tminus :: real \Rightarrow real \Rightarrow real$ (**infixl** ‹⊖› 60)
where
$x \ominus y = max\ (x - y)\ 0$

lemma *minus-le-tminus*[*intro*!,*simp*]:
$a - b \leq a \ominus b$
⟨*proof*⟩

lemma *tminus-cancel-1*:
$0 \leq a \Longrightarrow a + 1 \ominus 1 = a$
⟨*proof*⟩

lemma *tminus-zero-imp-le*:
$x \ominus y \leq 0 \Longrightarrow x \leq y$
⟨*proof*⟩

lemma *tminus-zero*[*simp*]:
$0 \leq x \Longrightarrow x \ominus 0 = x$
⟨*proof*⟩

lemma *tminus-left-mono*:
$a \leq b \Longrightarrow a \ominus c \leq b \ominus c$
⟨*proof*⟩

lemma *tminus-less*:
$[\![\ 0 \leq a; 0 \leq b\ ]\!] \Longrightarrow a \ominus b \leq a$
⟨*proof*⟩

lemma *tminus-left-distrib*:
assumes *nna*: $0 \leq a$

shows $a * (b \ominus c) = a * b \ominus a * c$
⟨*proof*⟩

lemma *tminus-le*[*simp*]:
$b \leq a \Longrightarrow a \ominus b = a - b$
⟨*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.