Available online at www.sciencedirect.com
SCIENCE DIRECT®

Theoretical Computer Science 346 (2005) 96–112

Theoretical Computer Science

www.elsevier.com/locate/tcs

Probabilistic guarded commands mechanized in *HOL*

Joe Hurd[a,*], Annabelle McIver[b], Carroll Morgan[c]

[a] *Computing Laboratory, Oxford University, Oxford, UK*
[b] *Department of Computing, Macquarie University, Australia*
[c] *School of Computer Science, University of New South Wales, Australia*

Abstract

The probabilistic guarded-command language (*pGCL*) contains both demonic and probabilistic non-determinism, which makes it suitable for reasoning about distributed random algorithms. Proofs are based on weakest precondition semantics, using an underlying logic of real- (rather than Boolean-)valued functions.

We present a mechanization of the quantitative logic for *pGCL* using the *HOL* theorem prover, including a proof that all *pGCL* commands satisfy the new condition *sublinearity*, the quantitative generalization of *conjunctivity* for standard *GCL*.

The mechanized theory also supports the creation of an automatic proof tool which takes as input an annotated *pGCL* program and its partial correctness specification, and derives from that a sufficient set of verification conditions. This is employed to verify the partial correctness of the probabilistic voting stage in Rabin's *mutual-exclusion* algorithm.

Keywords: pGCL; Formal verification; Probabilistic programs

1. Introduction

The probabilistic guarded-command language (*pGCL*) extends Dijkstra's original guarded-command language (*GCL*) [1] to include *probabilistic choice* [15]. The extension allows the specification of *quantitative* properties of programs, such as "the chance that the

* Corresponding author.
E-mail address: joe.hurd@comlab.ox.ac.uk (J. Hurd).

0304-3975/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2005.08.005

program delivers the correct output is at least 0.95." *Demonic non-determinism*, identified by Dijkstra as the key notion underlying *abstraction* and *refinement*, is retained. Within *pGCL* the combination of probability and non-determinism allows the realistic treatment of imprecise behaviour, avoiding the problem that exact probabilities cannot be implemented. For instance a program that behaves correctly (indicated by an ok result) with probability *at least* 0.95 can be described in *pGCL* as

$$\mathsf{ok}\ {}_{0.95}\oplus\ (\neg\mathsf{ok} \sqcap \mathsf{ok}).$$

Here, ${}_{0.95}\oplus$ represents a *probabilistic* choice of $(0.95, 0.05)$ between its left, right arguments, respectively; the $\sqcap$ on the other hand represents *demonic* choice, thought of as a selection made arbitrarily. This combination of probabilistic and demonic choices means that programs can exhibit a *range* of behaviours, rather than exactly one: above, the "demon" can affect the outcome only 5% of the time, and then might behave correctly in any case. The most that can be said is that the probability that the output will be ok lies in the interval between 95% and 100%. [1]

We describe the quantitative properties of probabilistic programs using *pGCL*'s *quantitative program logic* [16]. Programs are interpreted as *real*- rather than Boolean-valued functions of the state, and it is this generality which admits sound judgements concerning probabilistic and demonic choices, as above.

In this paper, we present the following significant novelties:

- A mechanization of *pGCL* programs (with weakest-precondition semantics) in higher-order logic, using the *HOL*4 theorem prover [3]:
- An automatic proof tool that takes as input annotated *pGCL* programs, and calculates verification conditions sufficient for their partial correctness; and
- The application of this proof tool to the formal verification of the probabilistic voting scheme in Rabin's *mutual-exclusion* algorithm [10].

A *mechanized* theory is one with a machine-readable logical formalization; and there are two main benefits to having a mechanized theory for *pGCL*. The first is the existence of a logical formalization at all: if the theory is formalized in a consistent logic by making definitions and then deriving consequences of them (instead of simply asserting axioms), then the theory has a strong assurance of consistency. The *HOL*4 theorem prover provides tool support for this "definitional approach," and as a result our *pGCL* theories are as consistent as the base higher-order logic.

The second benefit of mechanization is machine-readability: we can use the mechanized *pGCL* theories to support the creation of automatic proof tools that use weakest-precondition semantics for reasoning. For example, verifying *pGCL* programs typically involves much numerical calculation, and this can be formally carried out by rewriting with relevant theorems about real numbers. Since *HOL*4 is a theorem prover in the *LCF* family, it provides a full programming language (*ML*) for the user to write such tools [4]. Consistency is enforced by the *logical kernel*, a small module that is solely empowered to create objects of type `theorem`, which it does by applying the inference rules of higher-order logic.

[1] Another approach to the semantics of probabilistic programs [8] leaves out demonic non-determinism and instead takes these probability intervals as primitive.

We created many small tools to speed up mechanization and program verification, including the rewriting described above for real numbers. We also implemented a tool which takes as input an annotated program C, precondition P and postcondition Q, and generates verification conditions that are sufficient for partial correctness (the Hoare triple $\{P\}C\{Q\}$). It proves as many of these verification conditions as it can, simplifies the remainder and then returns them to the user as subgoals to be proved interactively.

Finally, we apply the theory and proof tools to the formal verification of the probabilistic module of Rabin's *mutual-exclusion* algorithm. This uses probability as a symmetry-breaking mechanism to elect a leader, and it is specified as having at least a $\frac{2}{3}$ chance of electing a unique leader, independent of the number of processors. We formally verify a sequential version of the algorithm, that is a data refinement of the original, establishing that if the algorithm terminates then the $\frac{2}{3}$ lower bound holds.

In Section 2, we present the formalization of *pGCL* in higher-order logic, illustrated with a simple worked example: the *Monty Hall* game. In Section 3, we describe the proof tool for generating verification conditions; and in Section 4, we apply the theory and tools to the verification of the probabilistic voting scheme in Rabin's *mutual-exclusion* algorithm.

1.1. Notation

Higher-order logic types include the Booleans $\mathbb{B}$, reals $\mathbb{R}$, and integers $\mathbb{Z}$. The notation $t : \tau$ means that the term t has type τ. Applying the function f to an argument x is expressed by juxtaposition $f\ x$, and multiplication uses an explicit operator $\times$ instead of juxtaposition. We use the notation $x \equiv t$ to mean x is defined to be t. Finally, we use the variable e to range over real-valued expressions denoting random variables over the state, t to range over transformers, s to range over states and c to range over commands.

2. Formalized *pGCL*

Fix a (possibly infinite) state space α and let $\overline{\alpha}$ be the probability *subdistributions* over α, that is functions $f : \alpha \to [0, 1]$ such that $\sum_{x \in \alpha} f\ x \leqslant 1$.

We can then view a probabilistic command c as a relation $\alpha \times \overline{\alpha} \to \mathbb{B}$ between initial states and probability subdistributions over final states. This is a relational (or operational) semantics: a program evolves from a definite initial state yet produces not a definite final state, but rather a probability distribution over final states that reflects the probabilistic branching in its execution. Demonic branching is indicated by relating the initial state to more than one final distribution. The following example shows both why we need relations instead of functions, and probability *sub*-distributions.

Example 1. Consider the following probabilistic program

$$\mathsf{Ex1} \equiv (n := n{+}1 \sqcap n := n{+}2) \ {}_{1/2}{\oplus}\ \mathsf{Abort},$$

where $\sqcap$ denotes demonic choice, ${}_{1/2}{\oplus}$ denotes symmetric probabilistic choice and Abort means "go into an infinite loop" (see Section 2.2 for precise definitions). The state space of $\mathsf{Ex1}$ is $\mathbb{Z}$ (the possible values of the program variable n); and applying the above semantics to Ex1 gives a relation that relates initial state $n = 0$ to these two subdistributions over final states:

$$(\cdots,\ -1 \mapsto 0.0,\ 0 \mapsto 0.0,\ 1 \mapsto 0.5,\ 2 \mapsto 0.0,\ 3 \mapsto 0.0,\ 4 \mapsto 0.0,\ \cdots)$$
$$(\cdots,\ -1 \mapsto 0.0,\ 0 \mapsto 0.0,\ 1 \mapsto 0.0,\ 2 \mapsto 0.5,\ 3 \mapsto 0.0,\ 4 \mapsto 0.0,\ \cdots)$$

The logic for *pGCL* has this relational semantics as a model: it is a quantitative weakest-precondition formulation originally due to Kozen [9], but with demonic choice added [16]. A program's final distributions are described by giving their expected values with respect to arbitrary random variables which we think of as "reward functions" that quantify the benefit of successful termination. The effect of this approach is to simplify the resulting proof system, without conceding expressivity [14].

Given a probabilistic command c, fix a reward function $Q : \alpha \to \mathbb{R}^+$ from final states to non-negative real numbers. Given an initial state x we can compute the average reward from executing c repeatedly by taking the *expected value* of random variable Q with respect to c's output distribution. If c is also demonic, we average over all distributions separately and take the least result (because adversaries act to minimize expected rewards). Lastly, if c does not terminate the convention is to reward with zero.

Using this procedure, we can calculate the expected reward for each initial state x, and thus end up with a reward function $P : \alpha \to \mathbb{R}^+$ from initial states to non-negative real numbers: the weakest precondition of Q.

Example 2. Consider again the probabilistic program Ex1, and suppose the reward function Q on final states is defined as

$$Q\ n \equiv \text{"2 if } n \text{ is odd, and 3 if } n \text{ is even."}$$

What is the expected reward function P on an initial state x? Half the time the program will loop and the reward will be zero. The remaining half of the time the least expected value over the demon's choice will be due to whichever assignment delivers an odd result, because the reward is only 2 for this, as opposed to 3 for the even outcome. Thus, the expected reward is

$$P\ x \equiv 1/2 \times 0 + 1/2 \times 2,$$

that is *one*, for every initial state x.

Expected-reward functions such as P and reward functions such as Q are simply called *expectations*. In *pGCL*, we view a probabilistic command c as an expectation *transformer*, mapping expectations on final states to expectations on the initial states. It is an elementary fact of probability theory that if the post-expectation is derived from a predicate—a characteristic function that rewards one for states satisfying the predicate and zero otherwise—then the pre-expectation gives the greatest guaranteed probability that the program terminates in a state satisfying the predicate.

We spend the remainder of this section presenting a formalization of this weakest precondition-style semantics of probabilistic programs.

2.1. Formalizing expectation transformers

In *pGCL*, expectations are functions from a state space α to the extended positive real numbers $\mathbb{R}^+ \equiv [0, +\infty]$. The real numbers have previously been mechanized in several different theorem provers (for an example in Ergo see [18]), so we have a solid basis on which to construct extended positive real numbers. Accordingly, we first created a new higher-order logic-type $\mathsf{posreal}$ to capture this domain, and lifted the usual arithmetic operations to it. Naturally, we had to make some choices about how the lifted arithmetic operations should behave on ∞, and the following identities summarize our decisions:

$$\begin{array}{rcl} 1/0 = \infty & 1/\infty = 0 & \forall x.\, \infty + x = \infty \\ \forall x.\, x \neq \infty \Rightarrow \infty - x = \infty & & \forall x.\, x \neq \infty \Rightarrow x - \infty = 0 \\ \forall x.\, 0 \times x = 0 & & \forall x.\, x \neq 0 \Rightarrow \infty \times x = \infty. \end{array}$$

Both addition and multiplication are defined to be commutative, so the above rules tell us that $\forall x.\ x \times 0 = 0$, for example. Also, division is defined in terms of multiplication and reciprocal, so from the above we can infer that $\infty/\infty = 0$. In fact, the only operation not covered by the above rules is $\infty - \infty$, which we deliberately leave unspecified. [2]

To support our later development we define min and max operations on $\mathsf{posreal}$, and a useful shorthand to enforce one-boundedness: $[x]^{\leqslant 1} \equiv \mathsf{min}\ x\ 1$.

We also prove a collection of theorems that can be used as rewrites to perform numerical calculations on elements of $\mathsf{posreal}$, reducing the burden on the user in interactive proof.

Example 3. The $\mathsf{posreal}$ calculations

$$\vdash (1/3 - 1/5) \times 6 = 4/5$$

and

$$\vdash \infty - 53 = \infty$$

can be automatically carried out by the *HOL*4 simplifier.

Now, we have defined the type of positive real numbers, we focus our attention on the type

$$(\alpha)\mathsf{expect} \equiv \alpha \to \mathsf{posreal},$$

of expectations on the state space α. Note that α is a type variable, able to be instantiated to any higher-order logic type, and therefore the theorems that we prove about expectations do not assume any properties of the state space. [3] We define several operations on expectations,

[2] In higher-order logic every function must be total, so $\infty - \infty$ must be some element x of $\mathsf{posreal}$, but there is no theorem that gives any information about x.

[3] In particular, the state space might be infinite.

which are just pointwise liftings of the corresponding operations on positive reals:

$$\begin{array}{rlrl} \mathsf{Zero} \equiv \lambda s.\, 0 & & \mathsf{Min}\; e_1\; e_2 \equiv \lambda s.\, \mathsf{min}\; (e_1\; s)\; (e_2\; s) \\ \mathsf{Infty} \equiv \lambda s.\, \infty & & \mathsf{Max}\; e_1\; e_2 \equiv \lambda s.\, \mathsf{max}\; (e_1\; s)\; (e_2\; s) \\ e_1 \sqsubseteq e_2 \equiv \forall s.\, e_1\; s \leqslant e_2\; s & & \mathsf{Cond}\; b\; e_1\; e_2 \equiv \lambda s.\, \mathsf{if}\; b\; s\; \mathsf{then}\; e_1\; s\; \mathsf{else}\; e_2\; s \end{array}$$
$$\mathsf{Lin}\; p\; e_1\; e_2 \equiv \lambda s.\, \mathsf{let}\; x \leftarrow [p\; s]^{\leqslant 1}\; \mathsf{in}\; x \times e_1\; s + (1-x) \times e_2\; s.$$

The type (α)expect forms a complete lattice, with Min and Max being the meet and join operators, and Zero and Infty being the bottom and top elements. Whereas the Zero expectation assigns every state a value of zero, the Infty expectation assigns every state a value of ∞.

Finally, the Lin operation constructs the linear interpolation between two expectations, and Cond switches between two expectations according to a predicate on the state space.

In *pGCL*, the semantics of a probabilistic program is an expectation transformer mapping postconditions on final states to weakest preconditions on initial states. Expectation transformers thus have higher-order logic type

$$(\alpha)\mathsf{transformer} \equiv (\alpha)\mathsf{expect} \to (\alpha)\mathsf{expect}.$$

To reason about expectation transformers, we borrow a few standard concepts from lattice theory, in particular the existence of least and greatest fixed points of monotonic transformers, which we refer to, respectively, as expect_lfp and expect_gfp.

Formalizing what it means to be a least or greatest fixed point of a expectation transformer is an easy matter:

$$\begin{array}{rl} \mathsf{lfp}\; t\; e & \equiv (t\; e = e) \wedge \forall e'.\, t\; e' \sqsubseteq e' \Rightarrow e \sqsubseteq e', \\ \mathsf{gfp}\; t\; e & \equiv (t\; e = e) \wedge \forall e'.\, e' \sqsubseteq t\; e' \Rightarrow e' \sqsubseteq e. \end{array}$$

The definitions of expect_lfp and expect_gfp use Hilbert's ε-operator[4] to pick any expectation that is a fixed point:

$$\mathsf{expect\_lfp}\; t \equiv \varepsilon e.\, \mathsf{lfp}\; t\; e, \qquad \mathsf{expect\_gfp}\; t \equiv \varepsilon e.\, \mathsf{gfp}\; t\; e.$$

Of course, such a definition is only useful if we can prove that there exist fixed points for a particular expectation transformer. That is why we also formalize the Knaster–Tarski theorem for lattices, which guarantees the existence of least and greatest fixed points for monotonic, up-continuous expectation transformers. Since these lattice theory concepts are referred to later in the definition of healthy transformers, for completeness we list here the formalized definitions:

$$\begin{array}{rl} \mathsf{monotonic}\; t & \equiv \forall e_1, e_2.\, e_1 \sqsubseteq e_2 \Rightarrow t\; e_1 \sqsubseteq t\; e_2, \\ \mathsf{lub}\; S\; e & \equiv (\forall e' \in S.\, e' \sqsubseteq e) \wedge \forall e_1.\, (\forall e' \in S.\, e' \sqsubseteq e_1) \Rightarrow e \sqsubseteq e_1, \\ \mathsf{chain}\; C & \equiv \forall e_1, e_2 \in C.\, e_1 \sqsubseteq e_2 \vee e_2 \sqsubseteq e_1, \\ \mathsf{up\_continuous}\; t & \equiv \forall C, e.\, \mathsf{chain}\; C \wedge \mathsf{lub}\; C\; e \Rightarrow \mathsf{lub}\; \{y \mid \exists z \in C.\, y = t\; z\}\; (t\; e). \end{array}$$

[4] Hilbert's ε-operator is a form of the axiom of choice: the term $\varepsilon x.\, \phi(x)$ is equal to some element that satisfies ϕ, or some element of the type if nothing satisfies ϕ.

2.2. Formalizing the weakest-precondition semantics

Next, we define the *pGCL* semantics of a simple programming language. For concreteness, we begin by defining a state space, $\mathsf{state} \equiv \mathsf{string} \to \mathbb{Z}$, representing a map from variable names to integer values. The following definition creates a new state from an old state by making a variable assignment of $f\ s$ to v:

$$\mathsf{assign}\ v\ f\ s \equiv \lambda w.\ \mathsf{if}\ w = v\ \mathsf{then}\ f\ s\ \mathsf{else}\ s\ w$$

Next, we define a new higher-order datatype for *pGCL* commands:

$$\begin{aligned}\mathsf{command} \equiv\ & \mathsf{Abort}\\ \mid\ & \mathsf{Skip}\\ \mid\ & \mathsf{Assign\ of\ string} \times (\mathsf{state} \to \mathbb{Z})\\ \mid\ & \mathsf{Seq\ of\ command} \times \mathsf{command}\\ \mid\ & \mathsf{Demon\ of\ command} \times \mathsf{command}\\ \mid\ & \mathsf{Prob\ of}\ (\mathsf{state} \to \mathsf{posreal}) \times \mathsf{command} \times \mathsf{command}\\ \mid\ & \mathsf{While\ of}\ (\mathsf{state} \to \mathbb{B}) \times \mathsf{command}.\end{aligned}$$

The Abort command represents non-termination of the program; in a technical sense it is “the worst possible program.” The next three command are completely standard: the Skip command does nothing; Assign $v\ f$ evaluates f on the current state and assigns the result to variable v; and the Seq $c_1\ c_2$ command is sequential composition, executing first c_1 and then c_2.

The Demon command uses demonic choice to decide which of the two argument commands to execute, and the Prob command uses probabilistic choice. Since the probability argument of Prob is a function $\mathsf{state} \to \mathsf{posreal}$, the choice probability is explicitly allowed to depend on the state.

Finally, the While $c\ b$ is a loop command that tests whether the state satisfies condition c: if so, the body b is executed and the loop is repeated, otherwise the command does nothing.

When writing commands, we enhance the readability with the following syntactic sugar:

$$\begin{aligned}v := f &\equiv \mathsf{Assign}\ v\ f,\\ c_1; c_2 &\equiv \mathsf{Seq}\ c_1\ c_2,\\ c_1 \sqcap c_2 &\equiv \mathsf{Demon}\ c_1\ c_2,\\ c_1\ {}_p{\oplus}\ c_2 &\equiv \mathsf{Prob}\ (\lambda s.\ p)\ c_1\ c_2,\\ \mathsf{If}\ b\ c_1\ c_2 &\equiv \mathsf{Prob}\ (\lambda s.\ \mathsf{if}\ b\ s\ \mathsf{then}\ 1\ \mathsf{else}\ 0)\ c_1\ c_2,\\ v := \{e_1, \ldots, e_n\} &\equiv v := e_1 \sqcap \cdots \sqcap v := e_n,\\ v := \langle e_1, \ldots, e_n\rangle &\equiv v := e_1\ {}_{1/n}{\oplus}\ v := \langle e_2, \ldots, e_n\rangle,\end{aligned}$$

$$b_1 \to c_1 \mid \cdots \mid b_n \to c_n$$
$$\equiv \begin{cases}\mathsf{Abort} & \text{if none of the } b_i \text{ holds (on the current state)}\\ \sqcap_{i\in I}\ c_i & \text{where } I \equiv \{i \mid 1 \leqslant i \leqslant n\ \wedge\ b_i \text{ holds}\}.\end{cases}$$

In addition, we routinely suppress mention of the state in expressions and conditions, writing for example $v := n + 1$ instead of $v := \lambda s.\ s\ n + 1$.

We now define the weakest precondition semantic operator wp, which is a higher-order logic function of type command $\rightarrow$ (state)transformer and maps commands to their semantic meaning as expectation transformers:

$$
\begin{aligned}
\vdash\ \ & (\mathsf{wp\ Abort} = \lambda e.\ \mathsf{Zero}) \\
\wedge\ & (\mathsf{wp\ Skip} = \lambda e.\ e) \\
\wedge\ & (\mathsf{wp}\ (\mathsf{Assign}\ v\ f) = \lambda e, s.\ e\ (\mathsf{assign}\ v\ f\ s)) \\
\wedge\ & (\mathsf{wp}\ (\mathsf{Seq}\ c_1\ c_2) = \lambda e.\ \mathsf{wp}\ c_1\ (\mathsf{wp}\ c_2\ e)) \\
\wedge\ & (\mathsf{wp}\ (\mathsf{Demon}\ c_1\ c_2) = \lambda e.\ \mathsf{Min}\ (\mathsf{wp}\ c_1\ e)\ (\mathsf{wp}\ c_2\ e)) \\
\wedge\ & (\mathsf{wp}\ (\mathsf{Prob}\ p\ c_1\ c_2) = \lambda e.\ \mathsf{Lin}\ p\ (\mathsf{wp}\ c_1\ e)\ (\mathsf{wp}\ c_2\ e)) \\
\wedge\ & (\mathsf{wp}\ (\mathsf{While}\ b\ c) = \lambda e.\ \mathsf{expect\_lfp}\ (\lambda e'.\ \mathsf{Cond}\ b\ (\mathsf{wp}\ c\ e')\ e)).
\end{aligned}
$$

Example 4. In this example, the desired final state is one in which the variables *i* and *j* have the same value, and so we use the postcondition

$$post \equiv \mathsf{if}\ i = j\ \mathsf{then}\ 1\ \mathsf{else}\ 0.$$

First, consider the program

$$\mathsf{pd} \equiv i := \langle 0, 1 \rangle; \quad j := \{0, 1\}.$$

The intuitive reading of pd is that the variable *i* is first set to either 0 or 1 by tossing a fair coin, and then the demon sets variable *j* to either 0 or 1. With this interpretation, it is no surprise that we can never beat the demon, and indeed we can prove that in the weakest precondition every initial state is mapped to zero:

$$\vdash \mathsf{wp\ pd}\ post = \mathsf{Zero}.$$

Next, consider the program

$$\mathsf{dp} \equiv j := \{0, 1\}; i := \langle 0, 1 \rangle,$$

which does the assignments the other way around. First, the demon must set variable *j*, and then variable *i* is set using the fair coin. In this case, we can prove

$$\vdash \mathsf{wp\ dp}\ post = \lambda s.\ 1/2,$$

which corresponds to our intuition that the demon does not know the outcome of the fair coin before it is tossed, and therefore can be beaten half the time on average.

2.3. Healthiness conditions

For standard *GCL*, Dijkstra introduced several "healthiness conditions" that characterize exactly the predicate transformers that correspond formally to an equivalent operational (relational) semantics of programs [1]; the conditions are used to derive sound proof rules for verification. Likewise, there is a correspondence between the expectation-transformer semantics of probabilistic programs and the operational interpretation of probabilistic programs—in fact an expectation transformer is healthy if it is feasible, up_continuous and sublinear [16], where up_continuous is a property of lattice theory and

$$\begin{aligned}
\mathsf{feasible}\ t &\equiv t\ \mathsf{Zero} = \mathsf{Zero},\\
\mathsf{scaling}\ t &\equiv \forall\, e, x.\ t\ (\lambda s.\ x \times e\ s) = \lambda s.\ x \times t\ e\ s,\\
\mathsf{subadditive}\ t &\equiv \forall\, e_1, e_2.\ t\ (\lambda s.\ e_1\ s + e_2\ s) \sqsubseteq \lambda s.\ t\ e_1\ s + t\ e_2\ s,\\
\mathsf{subtractive}\ t &\equiv \forall\, e, x.\ c \neq \infty \Rightarrow \lambda s.\ t\ e\ s - x \sqsubseteq t\ (\lambda s.\ e\ s - x),\\
\mathsf{sublinear}\ t &\equiv \mathsf{scaling}\ t \wedge \mathsf{subadditive}\ t \wedge \mathsf{subtractive}\ t.
\end{aligned}$$

Feasibility is an intuitive property, corresponding to Dijkstra's *Law of the Excluded Miracle*: if the value of all final states is zero, then so must be the value of all the initial states. Sublinearity in *pGCL* is the generalization of the conjunctivity healthiness condition in standard *GCL*, and is in fact equivalent to the single formula

$$\begin{aligned}
&\mathsf{sublinear}\ t\\
\equiv\ &\forall\, e_1, e_2, x_1, x_2, x.\\
&\quad (\lambda s.\ x_1 \times t\ e_1\ s + x_2 \times t\ e_2\ s - x)\ \sqsubseteq\ t\ (\lambda s.\ x_1 \times e_1\ s + x_2 \times e_2\ s - x).
\end{aligned}$$

Our present formalization does not include the proofs that connect expectation transformers with the relational semantics (which was first demonstrated by Morgan et al. [16]). Instead, we simply define a predicate

$$\mathsf{healthy}\ t \equiv \mathsf{feasible}\ t \wedge \mathsf{up\_continuous}\ t \wedge \mathsf{sublinear}\ t$$

and restrict our attention to healthy transformers. The properties monotonic, scaling, linear, subtractive are all logical consequences of healthy, as we check in the theorem prover.

As a point of interest, in finite state spaces the property up_continuous follows from feasible and sublinear, but in infinite state spaces this is no longer the case. By instantiating the state space to $\mathbb{Z}$ and using the transformer $\lambda e, s.\ \mathsf{inf}_n\{e\ n\}$ as a witness, it is possible to formally prove

$$\vdash \exists\, t.\ \mathsf{feasible}\ t \wedge \mathsf{sublinear}\ t \wedge \neg \mathsf{up\_continuous}\ t.$$

The main theorem of our formalization looks deceptively simple:

$$\vdash \forall c.\ \mathsf{healthy}\ (\mathsf{wp}\ c).$$

It states that applying the weakest precondition semantic operator wp to any command yields a healthy transformer.

Our direct proof is a structural induction on the command, and required 800 lines of *HOL*4 proof script for the main proof. (Dijkstra similarly used structural induction for the corresponding *GCL* proof.) The hardest part was proving sublinearity of while loops; for that we needed several lemmas, such as the monotonicity of expect_lfp and that subtraction subdistributes through healthy transformers.

However, the importance of healthiness conditions cannot be overstated: for instance, properties like these are what we use to deduce the simplifying rules for the verification calculator described below.

2.4. *The* Monty Hall *game*

An example is provided by the infamous *Monty Hall* game, where the role of the demon is played by the game show host.[5] There are three curtains and the contestant hopes to win a prize by guessing the curtain where it is hidden. The game begins with the demon choosing a prize curtain *pc* behind which to hide the prize. Next, the contestant chooses a curtain *cc* uniformly at random. The demon then chooses an alternative curtain *ac* that is not equal to either of *pc* and *cc*, and opens it. At this point, the contestant may either stick with his original choice of curtain, or switch to the remaining closed curtain. Should the contestant switch?

We code up the *Monty Hall* contestant with the following definition:

```
contestant $switch$ ≡
  $pc := \{1, 2, 3\}$;
  $cc := \langle 1, 2, 3 \rangle$;
    $pc \neq 1 \wedge cc \neq 1 \rightarrow ac := 1$
  | $pc \neq 2 \wedge cc \neq 2 \rightarrow ac := 2$
  | $pc \neq 3 \wedge cc \neq 3 \rightarrow ac := 3$;
  if $\neg switch$ then Skip else
    $cc :=$ (if $cc \neq 1 \wedge ac \neq 1$ then 1 else if $cc \neq 2 \wedge ac \neq 2$ then 2 else 3)
```

The left-hand side of the definition includes *switch* as a parameter of the contestant; this is used in the program on the right-hand side to determine whether to switch curtain in the last step. The postcondition is the desired goal of the contestant, i.e.,

$$\mathsf{win} \equiv \mathsf{if}\ cc = pc\ \mathsf{then}\ 1\ \mathsf{else}\ 0.$$

This example is small enough that we can verify it directly in *HOL*4 simply by rewriting away all the syntactic sugar, expanding the definition of wp and carrying out the numerical calculations. This has the effect of pushing the postcondition back to the start of the program, something that is not trivial to do by hand because the formulae become quite large. After 22 s and 250,536 primitive inferences in the logical kernel, the verification succeeds with the following theorem:

$$\vdash \mathsf{wp}\ (\mathsf{contestant}\ switch)\ \mathsf{win} = \lambda s.\ \mathsf{if}\ switch\ \mathsf{then}\ 2/3\ \mathsf{else}\ 1/3.$$

In other words, by switching the contestant is twice as likely to win the prize.

3. A verification-condition generator

In general, programs are shown to have desirable properties by proving *lower bounds*—for example, a program *Prog* can be shown to behave correctly with probability at least 0.95

[5] *Monty Hall* was host of the game show *Let's Make a Deal* from 1963 to 1976; ironically this game show was notable for requiring absolutely no skill or intelligence from its contestants.

by proving the inequality

$$\vdash (\lambda s.\, 0.95) \sqsubseteq \mathsf{wp}\ \mathit{Prog}\ (\mathsf{if\ ok\ then}\ 1\ \mathsf{else}\ 0),$$

where the post-expectation encodes the characteristic function of the set of states in which some Boolean ok holds. Of course, if a stronger guarantee is required (a 0.99 level of confidence, for example) then a stronger theorem would be required to establish it. In this section, we show how to mechanize the proof of such lower bounds; in fact, we concentrate on a generalization of the *weakest liberal precondition* semantics, a useful weakening of weakest precondition semantics. [6]

3.1. Weakest-liberal-precondition semantics

The weakest-liberal-precondition operator wlp is the partial correctness analogue of wp. Focussing on wlp and partial correctness greatly simplifies formal verification of looping programs, since the wp least fixed-point semantics are "the wrong way around" for proving lower bounds on preconditions.

In fact, the usual technique for proving total correctness for loops in *pGCL* is first to prove partial correctness, and then to show that wp and wlp agree on the while loop—this amounts to proving that the loop terminates with probability 1. This is the *pGCL* analogue of the well-known rule

$$\text{total correctness} = \text{partial correctness} + \text{proof of termination}$$

and has been proved elsewhere for *pGCL* [12]. Moreover, simple techniques based on program variants have also been derived. However, for the remainder of this paper we will be solely interested in partial correctness, and so questions of termination will not concern us.

For partial correctness, if a program does not terminate then it satisfies every postcondition. Since the only places where a program may diverge are the Abort and While commands, the weakest-liberal-precondition semantic operator wlp differs from wp *only* on those two commands: they have semantics, respectively

$$\mathsf{wlp\ Abort} \equiv \lambda e.\, \mathsf{Infty},$$

and

$$\mathsf{wlp}\ (\mathsf{While}\ b\ c) \equiv \lambda e.\, \mathsf{expect\_gfp}\ (\lambda e'.\, \mathsf{Cond}\ b\ (\mathsf{wlp}\ c\ e')\ e).$$

The full *HOL* formalization is based on the partial correctness theory for *pGCL* [12]. [7]

We cannot expect wlp to produce healthy transformers like wp, since the fact that wlp Abort Zero = Infty trivially breaks feasibility, but wlp transformers are at least monotonic:

$$\vdash \quad \forall c, e_1, e_2.\, e_1 \sqsubseteq e_2 \quad \Rightarrow \quad \mathsf{wlp}\ c\ e_1 \sqsubseteq \mathsf{wlp}\ c\ e_2.$$

[6] In fact, for terminating programs there is no weakening.

[7] In fact, only the wlp *verification conditions* (proved in Section 3.2) are important here, and the most crucial of these—monotonicity—is satisfied by both our formalization of wlp and McIver and Morgan's [12].

This is a useful sanity check, and means that (because of the lattice theory) the greatest fixed point in the wlp semantics of the While command is always well-defined.

Example 5. We illustrate the difference between wp and wlp semantics on the simplest infinite loop: $\mathsf{loop} \equiv \mathsf{While}\ (\lambda s.\ \top)\ \mathsf{Skip}$.

For any postcondition *post*: $\vdash \mathsf{wp}\ \mathsf{loop}\ post = \mathsf{Zero}$ and $\vdash \mathsf{wlp}\ \mathsf{loop}\ post = \mathsf{Infty}$.

These correspond to the Hoare triples $[\bot]\ \mathsf{loop}\ [post]$ and $\{\top\}\ \mathsf{loop}\ \{post\}$, just what we would expect from an infinite loop.

3.2. wlp *verification conditions*

In this section, we assume that we have a *pGCL* command c and a postcondition q, and we wish to derive a lower bound on the weakest-liberal precondition. If we think of this as the first-order query $P \sqsubseteq \mathsf{wlp}\ c\ q$, then we can use the following theorems together with a Prolog interpreter to solve for the variable P.

$$\begin{array}{lrcl}
\vdash & \mathsf{Infty} & \sqsubseteq & \mathsf{wlp}\ \mathsf{Abort}\ Q \\
\vdash & Q & \sqsubseteq & \mathsf{wlp}\ \mathsf{Skip}\ Q \\
\vdash & (Q \circ \mathsf{assign}\ V\ F) & \sqsubseteq & \mathsf{wlp}\ (\mathsf{Assign}\ V\ F)\ Q \\
\vdash & R \sqsubseteq \mathsf{wlp}\ C_2\ Q\ \wedge\ P \sqsubseteq \mathsf{wlp}\ C_1\ R & \Rightarrow & P \sqsubseteq \mathsf{wlp}\ (\mathsf{Seq}\ C_1\ C_2)\ Q \\
\vdash & P_1 \sqsubseteq \mathsf{wlp}\ C_1\ Q\ \wedge\ P_2 \sqsubseteq \mathsf{wlp}\ C_2\ Q & \Rightarrow & \mathsf{Min}\ P_1\ P_2 \sqsubseteq \mathsf{wlp}\ (\mathsf{Demon}\ C_1\ C_2)\ Q \\
\vdash & P_1 \sqsubseteq \mathsf{wlp}\ C_1\ Q\ \wedge\ P_2 \sqsubseteq \mathsf{wlp}\ C_2\ Q & \Rightarrow & \mathsf{Lin}\ P\ P_1\ P_2 \sqsubseteq \mathsf{wlp}\ (\mathsf{Prob}\ P\ C_1\ C_2)\ Q \\
\vdash & P_1 \sqsubseteq \mathsf{wlp}\ C_1\ Q\ \wedge\ P_2 \sqsubseteq \mathsf{wlp}\ C_2\ Q & \Rightarrow & \mathsf{Cond}\ B\ P_1\ P_2 \sqsubseteq \mathsf{wlp}\ (\mathsf{If}\ B\ C_1\ C_2)\ Q
\end{array}$$

The advantage of propagating conditions backward (implemented here with a Prolog interpreter) is that unnecessary annotations can be avoided. For example, consider the sequence $\mathsf{wlp}\ (\mathsf{Seq}\ c_1\ c_2)\ q$. There is no need for an annotation between the two commands, because the Prolog interpreter uses the rules to solve for a lower-bound r on $\mathsf{wlp}\ c_2\ q$, then solves for a lower-bound p on $\mathsf{wlp}\ c_1\ r$, and then returns p as a lower bound on the whole command $\mathsf{wlp}\ (\mathsf{Seq}\ c_1\ c_2)\ q$.

However, annotations are required to deploy the following theorem about while loops:

$$\vdash \forall P, Q, b, c.\ P \sqsubseteq \mathsf{Cond}\ b\ (\mathsf{wlp}\ c\ P)\ Q \ \Rightarrow\ P \sqsubseteq \mathsf{wlp}\ (\mathsf{While}\ b\ c)\ Q.$$

To insert annotations, we define an assertion command that simply ignores the formula given as its first argument: thus $\mathsf{Assert}\ p\ c \equiv c$. This is the precise rule we give to the Prolog interpreter:

$$\vdash R \sqsubseteq \mathsf{wlp}\ c\ P \wedge P \sqsubseteq \mathsf{Cond}\ b\ R\ Q \ \Rightarrow\ P \sqsubseteq \mathsf{wlp}\ (\mathsf{Assert}\ P\ (\mathsf{While}\ b\ c))\ Q.$$

It is therefore left to the user to provide a useful loop invariant P in the Assert around the while loop. Note that the Prolog tactic will succeed on the first subgoal, deriving a lower bound for the body of the while loop, but the second subgoal will fail because there are no applicable rules. In our tactic failed subgoals do not initiate backtracking, but are instead turned into verification conditions. Therefore, in this way each while loop in the program will generate one verification condition, in this case that the supplied P is in fact a correct invariant for establishing Q. Nested while loops work in exactly the same way: the invariant for the outer loop will be propagated backwards through the body, and when it meets the inner while loop a verification condition will be generated. It is usually impossible to calculate the precise loop invariant, but the fact that the ability to provide a weaker loop invariant that still satisfies the specification turns out to be a effective strategy.

Note that the rule for while loops is the only one where the presence of the $\sqsubseteq$ predicate is necessary. In each of the rules for the other commands, all occurrences of $\sqsubseteq$ could be replaced by $=$ and the result would still be a valid rule. The reason that the $\sqsubseteq$ is necessary in the rule for while loops is because of the user-provided loop invariant. If the loop invariant provided was known to be the strongest possible, then every occurrence of $\sqsubseteq$ could be replaced by $=$ and the tool would calculate the exact value of wlp. This is exactly the approach taken in model checking.

The full wlp tactic works as follows:

(1) Take as input a goal of the form $p \sqsubseteq \mathsf{wlp}\ c\ q$.
(2) Expand any syntactic sugar in c.
(3) Create the query $X \sqsubseteq \mathsf{wlp}\ c\ q$ and pass to the Prolog interpreter.
(4) The result will be a theorem

$$\vdash \bigwedge_{1 \leqslant i \leqslant n} V_i \;\Rightarrow\; r \sqsubseteq \mathsf{wlp}\ c\ q,$$

where the V_i are verification conditions.
(5) Apply transitivity of $\sqsubseteq$ to reduce the initial goal to the subgoals $p \sqsubseteq r$ and $r \sqsubseteq \mathsf{wlp}\ c\ q$.
(6) Use the theorem returned by Prolog to reduce the subgoal $r \sqsubseteq \mathsf{wlp}\ c\ q$ to the subgoals $V_1, \ldots, V_n$.
(7) Expand all the subgoals with the definitions of $\sqsubseteq$, Min, Lin and Cond.
(8) Try to prove all the subgoals by simplifying them and carrying out any numerical calculations.
(9) Return all unproved subgoals to the user, to prove interactively.

Returning to the example of the *Monty Hall* game, we can apply the wlp tactic to prove the following partial correctness theorem completely automatically:

$$\vdash (\lambda s.\ \mathsf{if}\ \mathit{switch}\ \mathsf{then}\ 2/3\ \mathsf{else}\ 1/3) \;\sqsubseteq\; \mathsf{wlp}\ (\mathsf{contestant}\ \mathit{switch})\ \mathsf{win}.$$

Since there are no while loops in the $\mathsf{contestant}$ program, there were no verification conditions, and the only non-trivial subgoal was the $p \sqsubseteq r$ generated in Step 5 of the tactic. However, this was proved automatically by the simplification and calculation in Step 8, and so no subgoals were returned to the user.

This automatic verification of the *Monty Hall* game is obviously much less effort than the interactive proof version described in Section 2.4 which took 18 lines of *HOL*4 proof script, but the automatic version of the theorem is weaker: it only shows partial correctness.

4. Example: Rabin's *mutual-exclusion* algorithm

Suppose N processors are concurrently executing, and from time to time some of them need to access a critical section of code. Rabin's *mutual-exclusion* algorithm uses a probabilistic voting scheme to elect a unique "leader processor" that is permitted to enter the critical section [10].

The idea behind the voting scheme is beautifully simple: each processor tosses a fair coin until the first head is shown, [8] and the processor that required the largest number of tosses wins the election.

Example 6. The following *pGCL* program sets the variable *n* according to the desired distribution:

$$n := 0; b := 0; \mathsf{While}\ (b = 0)\ (n := n + 1; b := \langle 0, 1\rangle).$$

In our verification, we do not model *i* processors concurrently executing the above voting scheme, but rather the equivalent formulation of that system used by Rabin [10]:

(1) Initialize *i* with the number of processors competing for exclusive access to the critical section.
(2) If $i = 1$ then we have a unique winner: return SUCCESS.
(3) If $i = 0$ then the election has failed: return FAILURE.
(4) Toss the coins: since each toss of a fair coin produces a head with probability $\frac{1}{2}$, each processor retires with that probability. We reduce *i* by eliminating all these processors, since certainly none of them won the election.
(5) Return to Step (2).

The following *pGCL* program implements this algorithm:

```
rabin ≡ While $(1 < i)$ (
            $n := i$;
            While $(0 < n)$
               $(d := \langle 0, 1\rangle; i := i - d; n := n - 1)$
        )
```

The desired postcondition, that there was a unique winner, is

$$post \equiv \mathsf{if}\ i = 1\ \mathsf{then}\ 1\ \mathsf{else}\ 0.$$

A surprising fact about this voting scheme is that the probability of its success is *independent* of the number of processors. To prove that, we need to be able to show

$$pre \sqsubseteq \mathsf{wlp\ rabin}\ post, \qquad (1)$$

where $pre \equiv (\mathsf{if}\ i = 1\ \mathsf{then}\ 1\ \mathsf{else\ if}\ 1 < i\ \mathsf{then}\ 2/3\ \mathsf{else}\ 0)$, in which the $\frac{2}{3}$ does not depend on *i*.

Recall the interpretation of a precondition with respect to a given postcondition. The expression on the right at (1), evaluated at an initial state *s*, gives the probability that the postcondition will be established (namely, that there is a unique winner). This must be at least the expression on the left, which is *at least* $\frac{2}{3}$ for all initial states except $i = 0$ (when the satisfaction of the postcondition would be impossible in any case).

[8] In other words, each processor picks an integer from a Geometric($\frac{1}{2}$) distribution.

As rabin contains two While loops the invariant rule must be used twice. Thus, two loop invariants are needed, one for the inner, and one for the outer loop, and the most challenging part of the verification turned out to be finding them (of course). The correct invariant for the outer loop is simply *pre* above, but for the inner loop we used

$$\text{if } 0 \leqslant n \leqslant i \ \text{ then } (2/3) \times \text{invar1 } i\ n + \text{invar2 } i\ n \ \text{ else } 0,$$

where

$$\begin{aligned}&\text{invar1 } i\ n \equiv 1 - (\text{if } i = n \text{ then } (n+1)/2^n \text{ else if } i = n+1 \text{ then } 1/2^n \text{ else } 0),\\ &\text{invar2 } i\ n \equiv \text{if } i = n \text{ then } n/2^n \text{ else if } i = n+1 \text{ then } 1/2^n \text{ else } 0.\end{aligned}$$

Translating very roughly into English: invar1 corresponds to the probability that the inner loop terminates with $i > 1$; and invar2 to the probability that the inner loop terminates with $i = 1$. Therefore, the probability p that the *outer* loop will terminate with $i = 1$ satisfies $p = p \times \text{invar1} + \text{invar2}$, and we are proving that the voting algorithm works with $p = \frac{2}{3}$.

To deploy the wlp tactic, an equivalent annotated version of the program is required, constructed by using Assert to annotate rabin with the above invariants. Next, the wlp tactic is applied to the annotated program, and three subgoals are produced (one as usual, plus two verification conditions generated by the while loops). The wlp tactic proves one of these automatically, and simplifies the other two. We apply some custom simplifications, and are left with three non-trivial subgoals which depend on properties of exponentials. These are despatched by 58 lines of proof script, completing the verification of the specification (1) of the behaviour of rabin.

5. Conclusions and future work

We have shown how to formalize in higher-order logic the theory of *pGCL*, a language for reasoning about both demonic and probabilistic choice in a common framework; we have implemented a verification-condition generator to assist with formally proving the partial correctness of programs, and we have demonstrated it on some small examples.

In addition to mechanizing a direct proof that the weakest precondition semantics always give healthy transformers, we have formalized the notion of weakest liberal preconditions and implemented a verification condition generator to assist with formally proving the partial correctness of programs. Finally, we applied the theory and tools to the verification of the probabilistic voting scheme in Rabin's *mutual-exclusion* algorithm.

This work demonstrates the benefits of mechanizing a theory of program semantics using a theorem prover. In particular, the fact that the theorem prover was interactive fitted very nicely with the verification-condition generator: if subgoals appeared that could not be proved automatically, then instead of causing a failure they could be passed on to the user for manual proof. Moreover we took advantage of the *LCF* design of *HOL*4, which preserves the consistency of user-defined tactics: the verification-condition generator is highly complex, but nevertheless any theorems that it creates have a high assurance of soundness.

Future work will focus on formalizing the correspondence between wp and wlp semantics, with the aim of implementing a total-correctness verification generator. This will additionally require proofs of termination, and it will be interesting to provide tool support for probabilistic variants and other termination arguments.

6. Related work

The first author has mechanized a semantics of probabilistic programs in *HOL*4 [7], but this language did not support demonic choice. The third author has recently extended the B tool (a proof assistant for program refinement) with a probabilistic choice construct [6].

Probabilistic model checkers such as *PRISM* [11] effectively calculate weakest preconditions for finite-state machines incorporating both probabilistic and demonic choice, and can also deal with loops without needing helpful annotations. On the other hand, the limited expressivity of the logic means that sometimes it cannot model algorithms in their full generality, but instead must restrict to a fixed number of processors.

Harrison has previously mechanized Dijkstra’s weakest precondition semantics for standard *GCL* in the *HOL Light* theorem prover [5], and Nipkow has produced a comprehensive mechanization of Hoare logics in the Isabelle theorem prover [17]. Finally, there have been several verification condition generators for while languages created for use with the *HOL* theorem prover, beginning with Gordon’s in 1989 [2].

Acknowledgements

This work was completed while Hurd was on leave from the Computer Laboratory at Cambridge, visiting McIver with the support of a Fellowship at Macquarie University in Sydney. He is currently a Junior Research Fellow at Magdalen College in Oxford. Morgan holds an Australian Professorial Fellowship at the University of New South Wales, and is associated with *NICTA*.

References

[1] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[2] M.J.C. Gordon, Mechanizing programming logics in higher order logic, in: G. Birtwistle, P.A. Subrahmanyam (Eds.), Current Trends in Hardware Verification and Automated Theorem Proving, Springer, Berlin, 1989, pp. 387–439.

[3] M.J.C. Gordon, T.F. Melham, Introduction to HOL (A theorem-proving environment for higher order logic), Cambridge University Press, Cambridge, 1993.

[4] M. Gordon, R. Milner, C. Wadsworth, Edinburgh LCF, Lecture Notes in Computer Science, Vol. 78, Springer, Berlin, 1979.

[5] J. Harrison, Formalizing Dijkstra, in: J. Grundy, M. Newey (Eds.), Theorem Proving in Higher Order Logics, 11th Internat. Conf., TPHOLs ’98, Lecture Notes in Computer Science, Vol. 1497, Canberra, Australia, Springer, Berlin, September 1998, pp. 171–188.

[6] T.S. Hoang, Z. Jin, K. Robinsion, A.K. McIver, C.C. Morgan, Probabilistic invariants for probabilistic machines, in: Proc. third Internat. Conf. of *B* and *Z* Users 2003, Lecture Notes in Computer Science, Vol. 2651, Springer, Berlin, pp. 240–159.

[7] J. Hurd, Formal verification of probabilistic algorithms, Ph.D. Thesis, University of Cambridge, 2002.

[8] M. Huth, The interval domain: a matchmaker for aCTL and aPCTL, in: R. Cleaveland, M. Mislove, P. Mulry (Eds.), US–Brazil Joint Workshops on the Formal Foundations of Software Systems, Electronic Notes in Theoretical Computer Science, Vol. 14, Elsevier, Amsterdam, 2000.
[9] D. Kozen. A probabilistic PDL, Proc. 15th ACM Symp. on Theory of Computing, 1983.
[10] E. Kushilevitz, M.O. Rabin, Randomized mutual exclusion algorithms revisited, in: M. Herlihy (Ed.), Proc. 11th Ann. Symp. on Principles of Distributed Computing, Vancouver, BC, Canada, ACM Press, New york, August 1992, pp. 275–283.
[11] M. Kwiatkowska, G. Norman, D. Parker, Prism: probabilistic symbolic model checker, in: Proc. of PAPM/PROBMIV 2001 Tools Session, September 2001.
[12] A.K. McIver, C.C. Morgan, Partial correctness for probabilistic programs, Theoret. Comput. Sci. 266 (1–2) (2001) 513–541.
[14] C. Morgan, Proof rules for probabilistic loops, in: H. Jifeng, J. Cooke, P. Wallis (Eds.), Proc. BCS-FACS 7th Refinement Workshop, Workshops in Computing, Springer, Berlin, 1996.
[15] C. Morgan, A. McIver, pGCL: formal reasoning for random algorithms, South African Comput. J. 22 (1999) 14–27.
[16] C. Morgan, A. McIver, K. Seidel, Probabilistic predicate transformers, ACM Trans. Programming Languages Systems 18 (3) (1996) 325–353 See also [12].
[17] T. Nipkow, Hoare logics in Isabelle/HOL, in: H. Schwichtenberg, R. Steinbrüggen (Eds.), Proof and System-Reliability, Kluwer, Dordrecht, 2002, pp. 341–367.
[18] J. Shield, I.J. Hayes, D.A. Carrington, Using theory interpretation to mechanise the reals in a theorem prover, in: C. Fidge (Ed.), Electronic Notes in Theoretical Computer Science, Vol. 42, Elsevier, Amsterdam, 2001.