Nordic Journal of Computing

Cost-based analysis of probabilistic programs mechanised in HOL

Orieta Celiku[*] and Annabelle McIver[†]

Abstract. We provide a HOL formalisation for analysing expected time bounds for probabilistic programs. Our formalisation is based on the quantitative program logic of Morgan et al. [21] and McIver's extension of it [17] to include performance-style operators. In addition we provide some novel results based on probabilistic data refinement which we use to improve the utility of the basic method.

1. Introduction

Randomisation is often used to improve overall efficiency in programs, or to solve programming problems where standard techniques fail — typical situations include symmetry breaking in networks, preventing potentially deadlocking behaviour. The underlying *probabilities* contributing to such programs' computations mean that many of their properties are *cost-based, viz.* quantified properties such as the probability that some goal is established, or the "expected" or "average" time it takes until it is. *Mechanising* formal techniques for analysing the latter property forms the principal topic of this paper.

Mechanising a theory refers to the creation of a machine readable logical formalisation, and there are two main reasons for doing so. First, if a mathematical theory is formalised within a consistent logic by making definitions and then deriving their consequences (contrast axiom assertion), then it has a strong assurance of consistency. The HOL theorem prover [10] provides tool support for this "definitional approach" to mechanisation, and as a result our probabilistic theories for analysing expected running times are as consistent as the base higher-order logic (which in turn is as consistent as ZF set theory [10]).

Second, once mechanised, we can use the theories to implement "automated proof tools" to support reasoning about probabilistic programs. For example, verifying the cost-based properties of probabilistic programs typically involves much numerical calculation, and this can be formally carried out by a straightforward proof tool that works by rewriting with relevant theorems about real numbers. Since HOL is a theorem prover in the LCF family, it provides a full programming language (ML) for the user to write

[*]Turku Centre for Computer Science and Åbo Akademi University, 20520 Finland: *oceliku@abo.fi*

[†]Department of Computer Science and Mathematics, Macquarie University, 2109 Australia: *anabel@ics.mq.edu.au*

Received June 7, 2004.

automatic proof tools such as these [9]. 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.

This paper makes three main contributions to both programming theory and practice.

- We create a HOL mechanisation of a theory for the probabilistic "weakest precondition semantics" for a small programming language pGCL, which includes probabilistic choice and nondeterminism; the latter is retained from standard programming theories and is crucial for abstraction and program refinement. The weakest precondition semantics is based on the *quantitative program logic* of Morgan et al. [21] which allows quantitative properties to be expressed via real- rather than Boolean-valued terms, and has a well-developed calculus which is suited to calculation of numeric quantities. Our mechanisation is based on Hurd's earlier formalisation of pGCL [13] which differs only in its treatment of nondeterminism — a difference which is necessary for our application to expected running times.
- We formalise a theory for a derived semantic operator Δ [17], which expresses the expected time for an iterative program (or system) to terminate. Based on the probabilistic weakest precondition theory we implemented an automatic proof tool which takes as input a program, and a "probabilistic invariant", and generates verification conditions to prove upper bounds on expected running times. The tool proves as many of these verification conditions as it can, simplifies the remainder and then returns them to the user as subgoals in an interactive proof.
- We introduce two novel proof rules based on the refinement of probabilistic programs, instances of which are suitable for implementation in a mechanised proof tool. We use these rules to propose a general method for analysing expected running times: in short the essential probabilistic properties underlying program termination are abstracted away obtaining a simpler "abstract" program which is more amenable to direct Δ analysis. The relation between the running times of the abstract and concrete systems is guaranteed by program refinement.

In addition we illustrate our mechanisation with several examples, including Herman's self-stabilisation protocol [11].

The paper is organised as follows. Sec. 2 comprises a summary of the basic theory for the quantitative program logic for pGCL, and a description of its formalisation in HOL, including a short overview of Hurd's work on which ours depends. Sec. 3 introduces the formalisation of the semantic operator Δ and derives some of its properties necessary to implement our automated tool, and in Sec. 4 we illustrate the ideas with some examples. In Sec. 5 we consider how to improve the efficiency of the mechanised analysis for more complicated programs, and finally in Sec. 6 we use the technique to analyse

Herman's self-stabilisation ring protocol. All our theorems except for Lem. 3 have been proved in HOL using the definitional theories described in Sec. 2 and Sec. 3, and so we do not supply the detailed proofs. The proof of Lem. 3 is set out in full in the appendix.

We use S for an underlying (countable) state space. $\overline{S}$ denotes the set of probability distributions over S: a probability distribution is a normalised function from S into the real interval $[0, 1]$. We use "." for function application, so that if $f : A \to B$, and a has type A then $f.a$ has type B.

2. Probabilistic semantics

Given a program *Prog* and predicates *pre* and *post*, the standard interpretation of the judgement $\mathit{pre} \Rrightarrow \mathsf{wp}.\mathit{Prog}.\mathit{post}$ using Dikstra's weakest precondition semantics is, if program *Prog* is executed from any initial state satisfying *pre*, then it is guaranteed to terminate in a state satisfying *post*. In this view programs are identified with *predicate transformers*, mapping postconditions to preconditions. Moreover, the wp semantics has considerable calculational appeal — rather than reasoning operationally about detailed execution paths, precise *static* properties of programs are articulated by wp judgements; syntactic rules eliminate "wp", reducing proofs of correctness to proofs in predicate calculus. In addition, *nondeterminism*, the mathematical notion underlying abstraction and refinement, is naturally accommodated. As we explain next, the probabilistic generalisation of wp enjoys both these qualities.

Unlike standard programs, probabilistic programs do not produce definite final states — although any *single* execution of such a program will result in the production of some specific state, which one in particular might well be impossible to predict (if its computation is governed by some random event). However over many executions the relative *frequencies* with which final states occur will be correlated with the program's known underlying random behaviour. Thus operationally we may model programs as functions which output *probability distributions* over final states.

Nondeterminism is still relevant even in the probabilistic context to give some leeway in precision when specifying and reasoning about those final distributions: it is unlikely that a specific probabilistic branching can be implemented to great accuracy, and often it is possible to prove a particular (probabilistic) property by citing only incomplete information. For example, we could specify a biased coin by stipulating merely that heads (denoted head) should appear "with probability *at least* 2/3" after each flip. Of course many coins satisfy this criterion and the "abstract" or "specification program" should be able to represent them all — nondeterminism makes this possible.[1]

[1] Here we take the view, proposed by for example Morgan [18] and Back [2] that there is no essential difference between specifications and implementations except in their degree of abstraction: they are all programs.

Rather than formalising the operational model based on functions from initial states to distributions, we instead formalise the associated *quantitative program logic* [21] which provides the means to specify and prove properties of those final distributions. Terms in the logic are interpreted as real- rather than Boolean-valued functions which enables the numeric frequencies of probabilistic events to be expressed. To specify the above bias in coins — that the probability of terminating in the state head is *at least* 2/3 — we are able to use a *probabilistic* "wp" statement for a suitably generalised definition (which we call dwp) and that of implication ($\Rrightarrow$, given below):

$$\underline{2/3} \ \Rrightarrow\ \mathsf{dwp}.\mathrm{coin}.[\mathsf{head}]\ . \tag{1}$$

The term now associated with the precondition is a real-valued function returning 2/3 when evaluated at any state, and $[\mathsf{head}]$ represents a *characteristic* function that returns 1 if the state is head, and zero otherwise. Thus we have weakened "guaranteed to terminate" to "guaranteed to terminate with some probability". More generally we allow arbitrary functions for the pre- and postconditions — we call them *expectations*, and we identify probabilistic programs with *expectation transformers*.

To illustrate this idea, consider an implementation of the above biased coin:[2]

$$\begin{array}{lll} \mathrm{coin}' \ \hat{=} & \mathsf{if}\ (c = \mathsf{head})\ \mathsf{then} & c := \mathsf{head}\ {}_{2/3}\!\oplus\ c := \mathsf{tail} \\ & \mathsf{else} & c := \mathsf{head}\ {}_{3/4}\!\oplus\ c := \mathsf{tail} \end{array}$$

The program coin' outputs two different distributions depending on c's initial state (though both satisfy the specified bias in (1)) — if c is initially showing a head then it will do so again with probability 2/3, but more favourably, with probability 3/4 if it is initially showing a tail. Now let Reward be the expectation

$$\mathrm{Reward} \quad \hat{=} \quad \lambda c.\ \mathsf{if}\ (c = \mathsf{head})\ \ \mathsf{then}\ 2\ \mathsf{else}\ 1\ ,$$

which is the "random variable" rewarding 2 for final states in which c comes up head and otherwise rewards 1. With these definitions we can compute the *expected final value* of Reward — though it does depend on the initial state. If c is initially head then it is the weighted average of Reward with respect to the distribution $(\mathsf{head}, \mathsf{tail}) = (2/3, 1/3)$, which is $2 \times 2/3 + 1 \times 1/3 = 5/3$. On the other hand if c is initially tail then it is the weighted average of Reward with respect to the distribution $(\mathsf{head}, \mathsf{tail}) = (3/4, 1/4)$, which is $2 \times 3/4 + 1 \times 1/4 = 7/4$. Thus we can say that the *pre-expectation* of coin' with respect to the *post-expectation* Reward is the function

$$\lambda c.\ \mathsf{if}\ (c = \mathsf{head})\ \mathsf{then}\ 5/3\ \mathsf{else}\ 7/4\ ,$$

[2] We are allowing ourselves the luxury of absolute precision for this example.

and coin', regarded as an expectation transformer has thus transformed Reward — i.e. the actual reward on termination — into the *expected* reward, which is all that can be reliably predicted prior to program execution.

In general when nondeterministic choice is present as well, there may be a *range* of final distributions satisfying a specification. In (1) above, for example, all output distributions of the form $(\mathsf{head}, \mathsf{tail}) = (p, 1{-}p)$, with $p \geq 2/3$ would need to be considered.[3] We account for this in the transformer semantics by quantifying the expected value of all relevant final distributions. Morgan et al.'s original quantification took the *minimum*, but for our application we will also need to use the *maximum*. We give details below, but for the moment we summarise the ideas introduced so far with the following definition in which we introduce two mappings dwp and awp which identify probabilistic programs as expectation transformers. The two maps differ only in their interpretation of nondeterminism — in the former case the transformer gives the minimum expected reward, whereas in the latter it gives the maximum.

Definition 1. (Expectation transformer semantics) *Given a* probabilistic program *Prog which maps initial states in S to final distributions in $\underline{S}$, and an expectation A, a non-negative real-valued function of S, the greatest* guaranteed *expected value of A after execution of Prog from initial state s with respect to the final distributions of Prog.s is given by* dwp.*Prog.A.s. The greatest* achievable *expected value of A after execution of Prog from initial state s is given by* awp.*Prog.A.s.*

To illustrate, using the specification (1) of the biased coin, the expected value of Reward with respect to the outcomes of any such 2/3-biased coin must exceed 5/3 (in any initial state), since $p{\times}2{+}(1{-}p){\times}1 \geq 5/3$ whenever $p \geq 2/3$. Thus dwp.coin.Reward $= \underline{5/3}$. For similar reasons, the expected value cannot exceed 2, thus awp.coin.Reward $= \underline{2}$.

It was shown by Morgan et al. [21] that an operational semantics of probabilistic programs in terms of functions delivering probability distributions over final states is mathematically equivalent to the above transformer semantics. However as the transformer semantics offers appealing simplicity and clarity for analysis, it is what we formalise in HOL.

2.1 Expectations, pGCL and its transformer semantics

In the rest of this section we describe the HOL formalisation of dwp and awp with respect to a small programming language called the probabilistic guarded commands — pGCL — an extension of Dijkstra's original guarded commands, with the addition of *probabilistic choice*, denoted ${}_p\oplus$ as above. The semantic mapping dwp has already been formalised by Hurd [13], and we draw extensively on his theories. Our contribution is to formalise awp in a similar manner.

[3] We are also assuming that the program coin terminates.

We begin with Hurd's formalisation of expectations which is based on the *positive reals* [13]. The positive reals are obtained by completing the non-negative reals with an introduced term infinity — denoted by ∞ — which dominates all other finite values. A consequence is that the positive reals become a complete partial order[4] so that all increasing sequences now have a limit (even though it might be ∞).

Within HOL the positive reals are created as a new higher-order logic type $\mathsf{posreal}$, and include the new term ∞. Arithmetic is also extended to the completed domain and the usual arithmetic operations from the reals ($+$, $\times$, etc.) are extended to preserve the continuity of the operators, so for example

$$1/\infty = 0 \quad \text{and} \quad \forall x \bullet x \neq \infty \Rightarrow \infty - x = \infty \ .$$

Numerical calculations on positive reals can be carried out automatically using the HOL simplifier.

Next the expectations theory [13] is built from the positive reals — expectations are functions from the state space to positive reals. The state space is modelled as a type variable α (which can be instantiated to any higher-order logic type), thus the expectations themselves are of type:

$$\alpha\ \mathsf{expect} \mathrel{\hat{=}} \alpha \to \mathsf{posreal} \ .$$

The expectation theory defines several useful constants and operations on expectations, which are pointwise liftings of the corresponding ones on positive reals. Some of these are listed in Fig. 1. As with the positive reals, the expectations form a complete partial order.

implication	$A \Rrightarrow A'$	$\hat{=}$	$\forall s \bullet A.s \leq A'.s$
maximum	$A \sqcup A'$	$\hat{=}$	$\lambda s \bullet A.s\ \mathsf{max}\ A'.s$
minimum	$A \sqcap A'$	$\hat{=}$	$\lambda s \bullet A.s\ \mathsf{min}\ A'.s$
addition	$A + A'$	$\hat{=}$	$\lambda s \bullet A.s + A'.s$
subtraction	$A - A'$	$\hat{=}$	$\lambda s \bullet A.s - A'.s$
multiplication	$A \times A'$	$\hat{=}$	$\lambda s \bullet A.s \times A'.s$
lifting standard predicates	$[P]$	$\hat{=}$	$\lambda s \bullet \mathsf{if} P.s\ \mathsf{then}\ 1\ \mathsf{else}\ 0$
lifting scalars	$\underline{c}$	$\hat{=}$	$\lambda s \bullet c$
scaling	cA	$\hat{=}$	$\lambda s \bullet c \times A.s$

where $A, A' : \alpha\ \mathsf{expect}$, $P : \alpha \to \mathbb{B}$, and $c : \mathsf{posreal}$.

Fig. 1: Operations on expectations.

Next we summarise Hurd's formalisation of pGCL in HOL. States are modelled by the higher-order logic type

$$\mathsf{state} \mathrel{\hat{=}} \mathsf{string} \to \mathbb{Z} \ ,$$

[4] A complete partial order is a partially-ordered set in which all ordered sequences of elements have a least upper bound in the set.

representing a map from variable names to integer values.[5] The following definition creates a new state from an old state by making a variable assignment:

$$\mathsf{assign}\ v\ f\ s \mathrel{\hat{=}} \lambda w \bullet \mathsf{if}\ w = v\ \mathsf{then}\ f.s\ \mathsf{else}\ s.w\ ,$$

where v is a variable name and f an integer-valued state function.

State expectations themselves are now of type $\mathsf{state\ expect}$. We will suppress lambdas and states when writing state expectations; for example, $x+\underline{3}$ will stand for $\lambda s \bullet \&(s.\text{“}x\text{”} + 3)$, where “&” converts an expression over integers into a $\mathsf{posreal}$ expression.[6] In some examples, we will appear to be using program variables of types other than $\mathbb{Z}$; however, those types (for example $\mathbb{B}$) have direct translations into the integers.

The language pGCL is defined as a new higher-order datatype which includes assignment, sequential composition, probabilistic choice and nondeterministic choice (see Fig. 2). Nondeterministic choice ($[\!]$) is a control construct which allows an “unpredictable agent” to decide which of the two argument commands to execute — it can be used to specify ranges of output distributions for example. On the other hand, the probabilistic choice (${}_p\oplus$) chooses randomly between its two arguments according to the specified weighting. Since the probability argument p is modelled as a function $\mathsf{state} \to \mathsf{posreal}$ [7], the choice of probability is explicitly allowed to depend on the state, and thus ordinary conditional choice is a special case.

The semantics for pGCL is given in terms of expectation transformers, which are functions from expectations to expectations; their HOL type is

$$\alpha\ \mathsf{transformer}\ \mathrel{\hat{=}} \alpha\ \mathsf{expect} \to \alpha\ \mathsf{expect}\ .$$

Next we define on the pGCL datatype a semantic map awp which produces a state expectation transformer for each language construct: standard assignments induce substitution of variables in the expectations, whereas probabilistic choice essentially averages over the possible final results. The interpretation of nondeterminism implies that the agent is an “angel”, seeking to maximise the expected reward. We set out the exact definitions for awp with respect to pGCL in Fig. 2, though we use “syntactic sugared” versions for each construct, distinguishing for example conditional and probabilistic choice. Hurd’s dwp-semantic mapping gives a similar semantics for pGCL — indeed it is essentially the same, differing from awp only in its “demonic” treatment of nondeterministic choice as a “minimising agent”, a distinction we note in Fig. 2.

The difference between the angelic and demonic interpretations is illustrated by considering $\mathit{Prog} \mathrel{\hat{=}} x :=\ 0 \mathrel{[\!]} x :=\ 1$. For example, $\mathsf{dwp}.\mathit{Prog}.[x = 0]$

[5] It would be possible to treat the state space more generally, for example as a tuple (see [5]), to allow for program variables of different types; however, this would add much complexity to the calculations.

[6] Conversions into positive reals often involve proving side conditions such as “the denominator is non-zero”, and in general showing that the expression is non-negative.

[7] The semantics of the language guarantees that this function is bounded by 1 (see [13]).

$$\begin{array}{rcl}
\mathsf{awp.skip}.A & \hat{=} & A \\
\mathsf{awp}.(x := E).A & \hat{=} & A[x := E] \\
\mathsf{awp}.(r; r').A & \hat{=} & \mathsf{awp}.r.(\mathsf{awp}.r'.A) \\
\mathsf{awp}.(r\ {}_p\!\oplus r').A & \hat{=} & p \times \mathsf{awp}.r.A + (\underline{1}{-}p) \times \mathsf{awp}.r'.A \\
\mathsf{awp}.(r_0@p_0 \mid \dots \mid r_n@p_n).A & \hat{=} & p_0 \times \mathsf{awp}.r_0.A + \cdots + p_n \times \mathsf{awp}.r_n.A \\
\mathsf{awp}.(r \mathbin{[\!]} r').A & \hat{=} & \mathsf{awp}.r.A \ \sqcup\ \mathsf{awp}.r'.A \\
\mathsf{awp}.(\mathsf{if}\ B\ \mathsf{then}\ r\ \mathsf{else}\ r').A & \hat{=} & [B] \times \mathsf{awp}.r.A + [\neg B] \times \mathsf{awp}.r'.A \\
\mathsf{awp}.(\mathsf{do}\ B \to r\ \mathsf{od}).A & \hat{=} & (\mu X \bullet [B] \times \mathsf{awp}.r.X + [\neg B] \times A)
\end{array}$$

E is an integer-valued state function, and the update is carried out using assign.
$p_0, \dots, p_n$ in the definition of generalised probabilistic choice sum to 1.
The term $(\mu X \dots)$ in the definition of (do ... od) refers to the least fixed point with respect to $\Rrightarrow$; it is guaranteed to exist since the expectations form a complete partial order.

Note that dwp, the usual *demonic* interpretation (formalised by Hurd) differs only for nondeterministic choice — it *minimises* rather than *maximises*, thus $\mathsf{dwp}.(r \mathbin{[\!]} r').A \mathrel{\hat{=}} \mathsf{dwp}.r.A \sqcap \mathsf{dwp}.r'.A$. All other definitions for dwp can be rendered from the above list, syntactically replacing "awp" by "dwp".

Fig. 2: Probabilistic awp semantics.

$= \underline{0}$, whereas $\mathsf{awp}.Prog.[x = 0] = \underline{1}$. Thus a minimising demon presiding over the nondeterministic choice aims to avoid the postcondition (by selecting the branch "$x = 1$"). A maximising angel, on the other hand, would instead helpfully choose "$x = 0$". Within a context of such angelic behaviour a given postcondition will only fail to be established if *all* possible selections would fail to establish it. In more general contexts the angel seeks to maximise the expected result as far as it can. Observe however that the angel and demon's choices are complementary, a situation formalised by the next theorem.

Theorem 1. (Bounded duality of awp and dwp)

$$\forall A, c \bullet c \neq \infty \wedge A \Rrightarrow \underline{c} \;\Rightarrow\; \mathsf{awp}.r.A + \mathsf{dwp}.r.(\underline{c} - A) = \underline{c}\ ,$$

where r is any loop-free program (i.e. contains no occurrence of do ... od*).*

Thm. 1 says that, at least for loop-free programs, dwp and awp determine each other, with the "$-A$" in the dwp expression essentially turning all instances of minimisation into maximisation. Despite this duality, dwp-interpretations often make proofs easier to do, which is one reason we retain it. Another is *program refinement*. Refinement defines a partial order on the space of programs such that one program *Prog′* is more refined than another *Prog* — denoted $Prog \sqsubseteq Prog'$ — if *Prog′* is more concretely specified than *Prog*, equivalently if it exhibits a *smaller* range of nondeterminism. A program is *deterministic* if it cannot be refined any further. It turns out that within the dwp-semantics of programs this idea is exactly characterised [21], and so we use dwp (and not awp) to formalise refinement.

Definition 2. *We say program Prog is* refined by *program Prog′, or Prog* $\sqsubseteq$ *Prog′, if and only if,*

$$\forall A \bullet \mathsf{dwp}.Prog.A \Rrightarrow \mathsf{dwp}.Prog'.A \ .$$

Def. 2 states that the more refined a program is, the more dwp properties it satisfies. For example, the following refinement holds

$$x := 0 \ [\!]\ x := 1 \ \sqsubseteq \ x := 0 \ {}_{1/2}\!\oplus x := 1 \ ,$$

since the LHS merely guarantees that the final state satisfies $(x = 0) \vee (x = 1)$, whereas the RHS in addition guarantees that $(x = 0)$ is satisfied with probability 1/2. Another way to think about this refinement is in terms of implementing a specification: a valid implementation of a nondeterministic choice is to use the outcome of a random coin flip. Probabilistic choice however cannot similarly be "refined away" — it is clear, for example, that the program $x := 0\ {}_{1/2}\!\oplus x := 1$ has no proper refinements, because if it did we would be claiming counter intuitively that a biased coin is more refined than a fair one. Using Thm. 1 we can show that refinement corresponds to decreasing awp judgements, for bounded expectations.

Corollary 1. (Refining dually)

$$\forall A, c \bullet c \neq \infty \wedge A \Rrightarrow \underline{c} \Rightarrow (r \sqsubseteq r' \Rightarrow \mathsf{awp}.r.A \Lleftarrow \mathsf{awp}.r'.A) \ ,$$

for loop-free programs r and r'.

2.2 Healthiness conditions

monotonic t	$\hat{=}$	$\forall A_1, A_2 \bullet A_1 \Rrightarrow A_2 \Rightarrow t.A_1 \Rrightarrow t.A_2$
feasible t	$\hat{=}$	$t.\underline{0} = \underline{0}$
sup-additive t	$\hat{=}$	$\forall A_1, A_2 \bullet t.(A_1 + A_2) \Rrightarrow t.A_1 + t.A_2$
subtraction t	$\hat{=}$	$\forall c, A \bullet c \neq \infty \Rightarrow t.(A - \underline{c}) \Rrightarrow \ t.A - \underline{c}$
scaling t	$\hat{=}$	$\forall c, A \bullet t.(cA) \ = \ c(t.A)$
up-continuous t	$\hat{=}$	$\forall \mathcal{A} \bullet t.(\sqcup \mathcal{A}) \ = \ \sqcup(\lambda A : \mathcal{A} \bullet t.A)$
sup-linear t	$\hat{=}$	sup-additive $t \wedge$ subtraction $t \wedge$ scaling t
ahealthy t	$\hat{=}$	feasible $t \wedge$ sup-linear $t \wedge$ up-continuous t

t is a transformer, and $\mathcal{A}$ a directed set.

Fig. 3: Properties characterising healthy transformers.

In his original presentation of the predicate transformers [7] Dijkstra stated a number of axioms satisfied by programs-as-transformers — these so-called "healthiness conditions" are used to prove properties about programs. Similarly, there is a set of healthiness conditions for probabilistic programs first discovered by Morgan et al. [21]. We reproduce some of them here in Fig. 3, adapted to the awp-interpretation, and in fact they can be seen as generalisations of the standard properties. For example, *sup-additivity* (a variation of additivity of expectation operators from probability theory) generalises *disjunctivity* in standard contexts. In general, healthiness conditions are invaluable for proving properties about programs — monotonicity, for example, allows us to refine a program by refining individual sub-programs [2, 18], and it will guarantee the existence of the least fixed points to come.

We end this section by proving that the awp-semantics produces healthy transformers.

Theorem 2. (awp gives rise to healthy transformers)

$$\forall r \bullet \mathsf{ahealthy}\ \mathsf{awp}.r\ .$$

Proof*:* *Structural induction over pGCL.*

Hurd proved a similar set of healthiness conditions for the dwp-transformers, whose principal difference was that dwp-transformers are *sub-additive*[8], rather than sup-additive.

In this section we have formalised the basic infrastructure for proving probabilistic properties about pGCL programs. Our next task is to formalise a derived operator for analysing expected running times, to which we now turn.

3. Upper bounds on expected running times

In this section we introduce our verification method for analysing expected running times of (nondeterministic) iterative programs. We begin with some intuition. Given a program *step* in pGCL, and a predicate G, we write

$$\mathsf{do}\ \neg G \rightarrow \mathit{step}\ \mathsf{od} \qquad (2)$$

to represent the looping program which executes *step* repeatedly until G is established (or continues to loop indefinitely if G cannot be established). Our goal is to estimate the *expected* number of complete iterations of *step* required until that happens (with an infinite result in cases when it never does). Given the formal infrastructure now available to us, an obvious solution might be to introduce a fresh variable n into (2), whose job is to count the iterations: initially it would be set to 0, and then would be incremented after each complete execution of *step*. Given a post expectation n [9] from

[8] t is sub-additive if for any expectations A_1 and A_2, $t.A_1 + t.A_2 \Rrightarrow t.(A_1 + A_2)$.

[9] Recall the idiom that n used as an expectation returns the value of n at state s when evaluated there.

Def. 1 we can attempt to express the expected number of iterations as the pre-expectation:

$$\mathsf{awp}.(n := 0; \mathsf{do}\ \neg G \rightarrow \mathit{step}; n := n+1\ \mathsf{od}).n\ . \tag{3}$$

Here the angelic interpretation makes sure that any nondeterministic choices in *step* are resolved to keep the loop iterating for as long as possible ensuring that n is as large as the worst case on termination, with a result of ∞ if that is arbitrarily large. This is in line with our desire to analyse tight upper bounds.[10] Unfortunately, there are some technical issues to do with well-definedness, since we would potentially be taking the expected value of an unbounded function; moreover, the semantics for loops as least fixed points is not sound in cases of nontermination.[11]

However, the idea motivates an approach which avoids altogether the introduction of an explicit fresh variable n and the related problems of undefinedness and nontermination — that is the approach we adopt. Following McIver [17] we assume a "system" composed of two parts:

$$\neg G \rightarrow\ \mathit{step}\ , \tag{4}$$

where as above *step* is a loop-free program of pGCL, and G is a predicate — again it carries the same information as (2), but we use it only in the restricted context of expected running times.[12] We now define the *worst* expected number of complete executions of *step* required to establish G.

Definition 3. (Δ) *The (worst) expected running time for a system defined by step to achieve satisfaction of predicate G is defined as the least fixed point of a function that accumulates:*

$$\Delta(G, \mathit{step}) \ \hat{=}\ (\mu X \bullet [\neg G] \times \mathsf{awp}.\mathit{step}.(\underline{1} + X))\ .$$

Since the expectations form a complete partial order and, by Thm. 2 $\mathsf{awp}.\mathit{step}$ is monotone within that order, $\Delta(G, \mathit{step})$ always exists (even though it might be ∞).

For example, when G is "x is 1" and *step* is $x := 0\ {}_{1/2}\!\oplus\ x := 1$, then $\Delta(G, \mathit{step})$ is 2 when evaluated at $x = 0$, which agrees with our intuition that *step* must be executed on average twice for x to be set to 1; on the other hand $\Delta(\mathit{false}, \mathit{step})$ is always infinite, as *false* can never be satisfied.

[10] An ∞ result can be expected if the loop does not terminate with probability 1, or (as in the symmetric walk on the real line in Sec. 4) if the expected number of steps is arbitrarily large.

[11] The transformer semantics for loops is a *least* fixed point, which gives results that are too low in the case of nontermination. An alternative definition as a greatest fixed point would give results that are too high in general. See [15] for details.

[12] We avoid the use of "$\mathsf{do} \ldots \mathsf{od}$" here as we wish to develop some tailored transformers for performance; more generally, this notation fits in with a broader theoretical theory including probabilistic temporal logic which is useful for relating expected running times to other temporal notions.

McIver [17] showed how this definition does indeed formalise the expected number of *step* iterations, by considering probability distributions over "computation paths". Moreover, it can also be shown that if G can never be satisfied by executions of *step* then $\Delta(G, \mathit{step})$ is ∞, and conversely if $\Delta(G, \mathit{step})$ is finite then the *probability* of satisfying G by executing *step* is 1.[13]

We end this section by setting out some some properties of Δ. Our first property says that the more we want to achieve with *step*, the more steps it is going to take. (Recall from Fig. 1 that $[G] \Rrightarrow [G']$ is equivalent to saying that whenever G holds, so does G'.)

Lemma 1. (Δ is anti-monotonic on the first argument)

$$\forall G, G', \mathit{step} \bullet [G] \Rrightarrow [G'] \;\Rightarrow\; \Delta(G', \mathit{step}) \Rrightarrow \Delta(G, \mathit{step}) \ .$$

Next, we know that if G is already satisfied then no more steps are required.

Lemma 2. ($\Delta(G, \mathit{step})$ is zero at G)

$$\forall G, \mathit{step} \bullet [G] \times \Delta(G, \mathit{step}) \;\equiv\; \underline{0} \ .$$

Our final property follows directly from Def. 3 and the least fixed point property of monotone functions: to verify that X is an upper bound for $\Delta(G, \mathit{step})$, it is only necessary to verify that X satisfies the least fixed point equation for $\Delta(G, \mathit{step})$ — this makes it possible to reduce reasoning about upper bounds on expected times to reasoning about expectations. We call such X's *weak invariants.*

Theorem 3. (Invariance)

$$\forall G, \mathit{step}, X \bullet [\neg G] \times \mathsf{awp}.\mathit{step}.(\underline{1} + X) \Rrightarrow X \;\Rightarrow\; \Delta(G, \mathit{step}) \Rrightarrow X \ .$$

3.1 Automating the Δ analysis

To facilitate the mechanised proofs of running times, we implemented in HOL the *delta verification-condition generator* which uses Thm. 3, the awp semantics in Fig. 2, and a set of derived rules based on the healthiness conditions to generate the sufficient conditions for the correctness of goals of this form: $\Delta(G, \mathit{step}) \Rrightarrow X$. The tool uses a Prolog interpreter which applies the rules automatically to remove awp in goals of the form $\mathsf{awp}.r.A \Rrightarrow A'$, thus proofs are reduced to proving arithmetic inequalities between expectations. The generated conditions are then simplified as much as possible using the general HOL simplifier and tools for numerical calculations on positive reals [13]; any goal that cannot be proved automatically is returned to the user to be proved interactively.

[13] Equivalently we say that if the program $\mathsf{do}\ \neg G \rightarrow \mathit{step}\ \mathsf{od}$ does not terminate with probability 1 then $\Delta(G, \mathit{step})$ is ∞; conversely if $\Delta(G, \mathit{step}) < \infty$ then the loop terminates with probability 1.

4. Examples: Bounded and unbounded random walks

In this section we consider the expected performance of three different random walks. In each case we supply a weak invariant to verify the expected time to achieve the stated goal. We show later how random walks such as these often underlie the expected performance of much more complicated protocols.

Example 1: Finite bounded walk with absorbing barriers

Perhaps the simplest random walk is the symmetric bounded random walk with absorbing barriers. A particle can move on a bounded interval of the real line, moving one step to the left or right, with probability 1/2. We wish to analyse the expected time until the particle can escape its bounds (equivalently be "absorbed" by either one of its barriers).

We model this in pGCL using an integer-valued variable n which indicates the position of the particle on the real line. We take as its left-hand bound the position $n = 0$ and its right hand bound $n = N$ for some non-negative integer N, giving us the system

$$(0 < n < N) \;\rightarrow\; n := n{-}1 \;{}_{1/2}{\oplus}\; n := n{+}1 \;, \tag{5}$$

in respect of which we seek an upper bound for $\Delta(G_1, \mathit{step}_1)$. Here we write step_1 for $n := n-1\,{}_{1/2}{\oplus}\, n := n+1$, and $\neg G_1$ for the predicate $(0 < n < N)$.

According to Thm. 3, to verify an upper bound we must supply a weak invariant for the function [14]

$$(\lambda X \bullet [0 < n < N] {\times} (\underline{1} {+} \mathsf{awp}.\mathit{step}_1.X)) \;. \tag{6}$$

As with standard program invariants, this often requires an insight. Writing k (also) for the state in which n takes value k, we observe the following properties of $\Delta(G_1, \mathit{step}_1)$.

(1) If G_1 is already satisfied then no more steps are necessary (Lem. 2): $\Delta(G_1, \mathit{step}_1).0 \;=\; \Delta(G_1, \mathit{step}_1).N = \underline{0}$;

(2) The walk is symmetrical about $N/2$, thus so must $\Delta(G_1, \mathit{step}_1)$ be: $\Delta(G_1, \mathit{step}_1).k \;=\; \Delta(G, \mathit{step}_1).(N{-}k)$ for any $0 \leq k \leq N$.

One function that satisfies these conditions is the quadratic function [15]

$$[0 < n < N] \times n \times (\underline{N} - n) \;,$$

thus we conjecture that this is a weak invariant for (6). To verify it we reason as follows.

[14] If r is a loop-free pGCL program then $\mathsf{awp}.r.(\underline{1} + X) \;\equiv\; \underline{1} + \mathsf{awp}.r.X$.

[15] Note that a lesser symmetric linear function does not satisfy weak invariance.

$$[0<n<N]\times(\underline{1}+\mathsf{awp}.\mathit{step}_1.([0<n<N]\times n\times(\underline{N}-n)))$$

$$\equiv \quad [0<n<N]\times (\underline{1}+(n+\underline{1})\times(\underline{N}-n-\underline{1})/2+(n-\underline{1})\times(\underline{N}-n+\underline{1})/2) \qquad \text{Apply “}\mathsf{awp}\text{”}$$

$$\equiv \quad [0<n<N]\times n\times(\underline{N}-n)\ , \qquad \text{Arithmetic}$$

as required. Now appealing to Thm. 3 gives us that indeed $\Delta(G_1, \mathit{step}_1) \Rrightarrow [0<n<N]\times n\times(\underline{N}-n)$. Thus if $N=30$, and the particle is initially at position 15, then it must take $15(30-15)=225$ steps on average to escape the bounded interval.

A simple variation of this random walk is the "stumble" — this is when a particle can move up or down with probability p, or remain in position with probability $1-2p$, *viz.* it is represented by the system

$$(0<n<N)\ \rightarrow\ n:=\ n-1\ @p\mid \mathsf{skip}\ @(1-2p)\mid n:=\ n+1\ @p\ .$$

(Here we must have that $0<p\le 1/2$.) To compute the expected time, we observe that the *paths* taken by the stumbler are the same as for the walker above, it just takes longer to step off the current position — on average $1/(1-2p)$ longer. This means it must also take the stumbler $1/(1-2p)$ times longer to get anywhere at all, thus we must scale the above upper bound for the simple random walk by this factor. With that insight we can verify as above that the stumbler takes $n\times(\underline{N}-n)/(1-2p)$ on average to escape the bounded part of the line.

Example 2: An unbounded random walk

Consider again a particle moving between two boundaries as in Example 1. We generalise this system by allowing the boundaries to move as well. As before let n record the position of the particle on the real line; but this time let l and r respectively be variables recording the current positions of the left and right boundaries. Each of n, l and r can, with probability 1/2, remain in position, or with probability 1/2 move up one position, although each one elects to do so independently of the others. The aim is for the particle to "collide" with one of the two boundaries. The situation is summed up by the following system:

$$\begin{array}{lll}(l<n<r) & \rightarrow & \mathsf{skip}\ {}_{1/2}\!\oplus n:=\ n+1;\\ & & \mathsf{skip}\ {}_{1/2}\!\oplus l:=\ l+1;\\ & & \mathsf{skip}\ {}_{1/2}\!\oplus r:=\ r+1\end{array}$$

where we write step_2 for the program fragment to the right of the guard, and $\neg G_2$ for the guard. Note that since the particle's distance between its right and left boundaries can become arbitrarily large (though with vanishing probability!) this is a type of infinite random walk.

To solve this problem, we again need an insight. First we notice that *relative* to either boundary the particle is exhibiting a “stumbling” walk mentioned as a variation to Example 1. To see that, observe that the distance between the particle and its left boundary is $(n{-}l)$, and that on each step this distance can either increase, decrease or remain the same with probabilities respectively 1/4, 1/4 or 1/2. We observe the same behaviour with respect to the particle's distance from its right boundary. Next we note that the formula verified for Example 1 is in fact the product of the particle's distances from its two boundaries. Thus we guess optimistically that the product of distances will work here too, though possibly scaled by some constant M to account for the stumbling behaviour. To calculate an appropriate value for M, we compute

$$\mathsf{awp}.\mathit{step}_2.(M(n{-}l){\times}(r{-}n)) \;\equiv\; M(n{-}l)\times(r{-}n){-}\underline{M}/4\ .$$

The condition for weak invariance of $M(n{-}l){\times}(r{-}n)$ with respect to the least fixed point equation for $\Delta(G_2, \mathit{step}_2)$ is that RHS+$\underline{1} \Rrightarrow M(n{-}l){\times}(r{-}n)$ whenever $(l < n < r)$ holds. Solving this inequation for M gives us that $M = 4$, and thus we have that $4(n{-}l){\times}(r{-}n)$ is a suitable weak invariant, which is easily verified:

$$\underline{1} + \mathsf{awp}.\mathit{step}_2.([l < n < r] \times 4(n{-}l)\times(r{-}n)) \;\Rrightarrow\; 4(n{-}l)\times(r{-}n)\ .$$

We can now deduce from Thm. 3 that $\Delta(G_2, \mathit{step}_2) \Rrightarrow [l < n < r]\times 4(n{-}l)\times (r{-}n)$. Thus if initially $l, n, r := 0, 3, 5$ then it takes on average $4{\times}3{\times}2 = 24$ steps for the particle to collide with one of its boundaries.

A by-product of this analysis is that the probability that the collision occurs at all is 1. (Recall the comment after Def. 3, and that this system operates over an infinite state space.)

Example 3: A bounded random walk in two dimensions

For our final example we take the problem whose solution first appeared in Mathematical Monthly [12], where it was described as follows.

> Three men play the old coin-matching game “odd man wins”. Play proceeds in a series of rounds: in a round, each player flips one of his (fair) coins. The player whose outcome differs from both the others wins that round, and he collects a coin from each of the two losers. As usual, in the case of a tie, they just flip again. If the stakes of the players consist of x, y and z coins, what is the average (or expected) number of rounds before one of them goes broke?

We model their game in pGCL as the following system.

$$\begin{array}{lcr}
(x > 0 \wedge y > 0 \wedge z > 0) & \rightarrow & \mathsf{skip} \quad @1/4 \\
 & & (x := x+2; y := y-1; z := z-1) \quad @1/4 \\
 & & (y := y+2; z := z-1; x := x-1) \quad @1/4 \\
 & & (z := x+2; x := x-1; y := y-1) \quad @1/4
\end{array}$$

The first outcome is the result of a draw, and the other three correspond to how the distribution of coins changes depending on which of the three players wins.

Our aim is to find an upper bound to $\Delta(G_3, step_3)$, where as before $step_3$ represents the program to the right of the guard and $\neg G_3$ the guard itself. We begin by noting some properties of this game.

(1) Money is neither created nor destroyed, thus the total $x+y+z$ remains constant throughout the game;

(2) The relative motion of players' stakes is coupled, just as the particle's relative motion with respect to its two boundaries is in Example 2;

(3) Because of the possibilities of a tie, there is some "stumbling" behaviour similar to that of the stumbler described in Example 1.

From the second observation, we guess that a weak invariant will be proportional to the product of the players' stakes; from the third observation we also guess that the weak invariant we seek will be multiplied by some constant to account for the stumbling behaviour. Thus the weak invariant we seek is of the form $M(x \times y \times z)$ for some constant value M. To calculate M, we compute:

$$\mathsf{awp}.step_3.M(x \times y \times z) \;\equiv\; M(x \times y \times z) - M(x + y + z)3/4\ .$$

The condition for weak invariance is, as above, that $RHS{+}\underline{1} \Rrightarrow M(x{\times}y{\times}z)$, an inequation which we can solve for M, giving $M = 4/(3(x + y + z))$. However, M was assumed to be *constant* in this calculation, and luckily the first observation implies that it is. Thus we deduce that $4/3(x \times y \times z)/(x + y + z)$ (for non-negative x, y and z) is the upper bound we seek, a fact we can verify directly as above.

In this section we saw how the basic properties of Δ can be applied to verify upper bounds on various kinds of random walks. In the next section we show how this approach is also effective more generally.

5. Data abstraction and refinement

In distributed protocols, randomisation is often employed as a "symmetry breaker" for avoiding potential deadlocks. This introduced randomisation in turn induces patterns of random state changes which, if looked at in the right way, are often consistent with the patterns observed in some varieties of random walks. It is thus those underlying random walks which determine the expected running time of the original protocol. In this section we provide the formal theory and some associated proof rules for revealing such underlying behaviour; the methods we use are based on the well known technique of *data refinement*, a variant of program refinement that allows programs to be compared even when they operate over different state spaces.

In summary the method we are proposing can be stated as follows. Given a randomised protocol, we first show that it is *anti-data-refined* by an "abstract" program which performs a random walk. We then analyse the abstract walking program directly, using the techniques of Sec. 3, finally appealing to an extended version of Cor. 1 to show that the upper bound for the abstract walk dominates the expected running time of the original protocol.

There are several benefits to this staged approach. The first is that the anti-refined walking program is much simpler algorithmically than the original protocol, which makes the Δ-analysis much easier to do. Secondly, one of the effects of refinement is to "collapse" probabilistic branches when different probabilistic choices correspond to the same "movement" in the abstract random walk. This collapsing stage requires appeal to a number of arithmetic (in)equalities, whose incorporation in a mechanised proof can be quite challenging. By localising the site where the arithmetic applies — at a refinement step instead of within the Δ-analysis — much of the work can be carried out automatically, improving the overall efficiency of the mechanised proof. Perhaps most important of all is that the Δ-analysis can be reused for different protocols: as the same variety of random walk underlies many different protocols, one imagines that a library of pre-analysed random walks would be available to the verifier, who then would need only to select the appropriate walk with which to establish the refinement, looking-up the Δ-analysis already provided.

Our task for this section is to address the data-refinement stage — we consider how to demonstrate rigorously that two systems operating over different state spaces correspond so that their respective running times can be compared. The idea is very simple: borrowing from the standard theory of data refinement [3, 20], we use an "abstraction invariant" *rep* (itself a program in pGCL), which converts variables used by one system into those used by another in such a way that the respective (probabilistic) transitions correspond. Formally, we say that the two systems $\neg G \rightarrow step$ and $\neg G' \rightarrow step'$ are related by abstraction invariant *rep* if

$$\neg G \quad \Rrightarrow \quad \mathsf{awp}.rep.\neg G' \text{ , and} \tag{7}$$

$$rep; step' \quad \sqsubseteq \quad step; rep \text{ .} \tag{8}$$

Condition (7) says that if the unprimed system requires another step (to establish G), then so does the primed system (to establish G'). Condition (8) says that the transitions in *step* correspond to those of *step*$'$, though *step*$'$ might be more nondeterministic. Overall we can prove that if both conditions hold together then $\Delta(G', step')$ dominates $\Delta(G, step)$.

In assessing the utility of this observation, we recall that the definition of program refinement Def. 2 includes a quantification over all expectations (and there are infinitely many of them), thus using Def. 2 directly to verify condition (8) is clearly infeasible.[16] We propose two solutions to this problem, dealt with separately below. The first (given at Thm. 4) is generally applicable, whereas the second (given at Thm. 5) can only be used for a restricted class of systems.

For the first solution we use the fact that (8) is too strong for our application — it is not necessary that the transitions correspond in all contexts, but only that they correspond within the context of the Δ-analysis. This means that it is sufficient to check that the inequality $\mathsf{dwp}.(rep; step').X \Rrightarrow \mathsf{dwp}.(step; rep).X$ holds only when X is (an upper bound of) $\Delta(G', step')$. In Thm. 4 condition (8) has thus been weakened to the condition labelled by (†), where we see it expressed in the equivalent awp-form.

THEOREM 4. (Δ DATA REFINEMENT) *Let $\neg G \rightarrow step$ and $\neg G' \rightarrow step'$ be two systems. Let rep be a pGCL program. If*

$$\begin{array}{ll}
 & [\neg G]\times\mathsf{awp}.(step; rep).(\underline{1}+\Delta(G', step')) \Rrightarrow \qquad\qquad\qquad \dagger \\
 & [\neg G]\times\mathsf{awp}.(rep; step').(\underline{1}+\Delta(G', step'))\ , \\
\textit{and} & [\neg G] \Rrightarrow \mathsf{awp}.rep.[\neg G']\ , \\
\textit{and} & (\forall A, A' \bullet \mathsf{awp}.rep.(A \times A') = \mathsf{awp}.rep.A \times \mathsf{awp}.rep.A')\ , \\
 & \\
\textit{then} & \Delta(G, step) \Rrightarrow \mathsf{awp}.rep.\Delta(G', step')\ .
\end{array}$$

Proof*: The proof uses the assumptions to show that $\mathsf{awp}.rep.\Delta(G', step')$ is a weak invariant with respect to the least fixed point function for $\Delta(G, step)$; the result then follows on appeal to Thm. 3.*

Given a system $G \rightarrow step$, a verifier can use Thm. 4 if he supplies two things: an abstract system $\neg G' \rightarrow step'$, and a pGCL program *rep* which effectively computes the primed state from the unprimed state. The conditions can be established with the help of the Δ-condition generator and similar rewriting tools. However, for all but trivial examples, interactive proof is required in order to discharge the verification conditions — the general HOL simplifier is not powerful enough to handle the arithmetic needed to prove the conditions.

We illustrate Thm. 4 by the systems set out in Fig. 4: two coins are tossed repeatedly until they show different sides. Although there are four possible states corresponding to the pair of variables x and y, it is clear that the running time depends only on a single bit of data, namely whether the coins are showing the same or different sides. This simpler random walk is described by the 2-state-walker, whose performance is given by the

[16] We recall though that the definition of standard program refinement for standard programs also uses a quantification over all predicates, yet sound rules for verifying data refinement have been implemented in the B toolkit [1] for instance that use only first order predicate calculus. The reason is that *conjunctivity* — one of Dijkstra's original healthiness conditions — justifies elimination of the universal quantification over predicates. Unfortunately, expectation transformers are not conjunctive [21] and we must solve the problem in some other way.

expectation $2[n = 1]$ (recall the comment after Def. 3). By Thm. 4 the running time of 2-coins is just as simple, and is given by $2[x = y]$, where the *rep* we use as justification is $n := [x = y]$. The conditions for Thm. 4 are routinely discharged in this case.

2-coins $\hat{=}$	$(x = y) \rightarrow$ $x := \mathsf{head}\ {}_{1/2}\oplus\ x := \mathsf{tail};$ $y := \mathsf{head}\ {}_{1/2}\oplus\ y := \mathsf{tail}$	2-state-walker $\hat{=}$	$(n = 1) \rightarrow$ $n := n - 1\ {}_{1/2}\oplus\ \mathsf{skip}$

The abstraction invariant is $n := [x = y]$. The running time for the 2-state-walker is given by $2[n = 1]$, thus by Thm. 4 the running time for the 2-coins is dominated by $\mathsf{awp}.(n := [x = y]).2[n = 1] \;\equiv\; 2[x = y]$.

Fig. 4: Abstracting a two-coin-tossing program.

The trouble with Thm. 4 is that it still combines the verification condition involving both $\Delta(G', \mathit{step}')$ and *step*. This means that if *step* (and therefore *rep*) or $\Delta(G', \mathit{step}')$ are necessarily complicated then the verification of condition (†) in Thm. 4 can still be very challenging. Our aim for the remainder of this section is to reduce (†) into conditions which do not involve both $\Delta(G', \mathit{step}')$ and *step* together. This leads us again to consider the feasibility of verifying the full refinement at (8), and to our second solution of this section, to which we now turn.

To get a feel for the problem of deciding valid program refinements, we consider what we would need to do to prove that $x := 0\ {}_{2/3}\oplus\ x := 1 \sqsubseteq Q$ for some arbitrary program Q. If the refinement is valid then intuitively we would expect Q to execute the probabilistic assignment to x, so at the very least we would have to perform two checks, namely that overall Q sets x to 0 with probability 2/3 and to 1 with probability 1/3.[17] It turns out that for this particular refinement that is all we would need to check — we say that $x := 0\ {}_{2/3}\oplus\ x := 1$ is *determined by predicates*, whose full definition is set out next.

Definition 4. (Determined by predicates) *A program Prog is said to be* determined by predicates *if, given an arbitrary program Q we have*

$$\mathit{Prog} \sqsubseteq Q \quad \textit{iff} \quad \mathsf{dwp}.\mathit{Prog}.[\mathit{pred}_i^s].s \;\leq \mathsf{dwp}.Q.[\mathit{pred}_i^s].s\ ,\ \textit{for all } i \textit{ and } s\ ,$$

where for each initial state s, the collection pred_i^s, $1 \leq i \leq m_s$ forms a finite set of predicates (depending on s and Prog, but not on Q). We call them the determining predicates.

The importance of the class of programs characterised by Def. 4 is a consequence of the fact that the set of determining postconditions depends only

[17] Unfortunately, there seems to be no way to reduce this number of checks.

on *Prog* (and not on the arbitrary Q); if there is only a small number of such predicates then the task of proving program refinement is again first-order, and the universal quantification in Def. 2 is not necessary.

We note first that standard programs are determined by predicates (see for example the normal form construction for predicate transformers [4]). Next we can extend the class of programs that are determined by predicates by introducing probabilistic choice in a restricted way.

Lemma 3. (Probabilistic specification programs) *pGCL programs of the form*

$$P_1 \ @p_1 \mid P_2 \ @p_2 \mid \ldots \mid P_n \ @p_n \ ,$$

are determined by predicates, where $P_1, \ \ldots, P_n$ are standard (probabilistic-choice-free).[18]

Proof: *See the appendix.*

In general, the set of determining predicates might be quite difficult to find, but in practice for probabilistic choices over nondeterministic assignments they can be constructed quite easily. For instance, in the generalised abstract random-walk step

```
walk ≙ if (n < N) then
           n :∈ {n+1, ..., n+k}       @p
         | n :∈ {n−k', ..., n−1}      @q
         | skip                       @(1−p−q)
       else n := n−1 _q⊕ skip ,
```

which specifies that when n is strictly less than N, the walker might step up (with step size at least 1 and at most k) with probability p, or step down (with step size at least 1 and at most k') with probability q; the remaining possibility is to stumble. Here the determining predicates are just related to the size of the step, namely $(n_0 < n \leq n_0 + k)$, $(n_0 - k' \leq n < n_0)$ and $(n = n_0)$, where we use the convention that n_0 represents the initial state of n. Thus we can formulate rules for proving general program refinements of the form $walk \sqsubseteq Q$ using a small number of checks (in this case four [19]). The following lemma sets out a typical example.

Lemma 4. (Walk-tailored simplification for data refinement) *Let $walk'$ be the program above with $k, k' \mathrel{\hat=} 1, 1$. Furthermore let $\neg G \rightarrow step$ be some system, where step is an arbitrary (loop-free) pGCL program. Define the predicates $n^+ \mathrel{\hat=} [n = n_0 + 1]$, $n^- \mathrel{\hat=} [n = n_0 - 1]$ and $e \mathrel{\hat=} [n = n_0]$. If rep is a pGCL program satisfying the following conditions*

$$[\neg G] \times \mathsf{dwp}.(rep; walk').n^+ \Rrightarrow [\neg G] \times \mathsf{dwp}.(step; rep).n^+ \ ,$$
$$\textit{and } [\neg G] \times \mathsf{dwp}.(rep; walk').n^- \Rrightarrow [\neg G] \times \mathsf{dwp}.(step; rep).n^- \ ,$$
$$\textit{and } [\neg G] \times \mathsf{dwp}.(rep; walk').e \Rrightarrow [\neg G] \times \mathsf{dwp}.(step; rep).e \ ,$$

[18] Recall from Sec. 2 that standard programs include ordinary conditional choice.
[19] We need one condition for each determining predicate and one condition involving their union. The latter is related to the presence of nondeterminism in the walk.

then $\forall X \bullet [\neg G] \times \mathsf{dwp}.(\mathit{rep}; \mathit{walk}').X \;\Rrightarrow\; [\neg G] \times \mathsf{dwp}.(\mathit{step}; \mathit{rep}).X$.
Proof: *The proof uses a construction similar to that given in the appendix to express an arbitrary expectation in terms of the determining predicates. The general refinement follows from the healthiness conditions of the* dwp *semantics.*

Note that in rules such as Lem. 4 the expectations on the left of the inequality are simple functions of p and q — for instance $\mathsf{dwp}.(\mathit{rep}; \mathit{walk}).n^+$ evaluated at n_0 is $p \times n_0$. Moreover experience shows that the proofs of the conditions are very similar to each other, so any difficulty in using such rules is more apparent than real. Finally combining Lem. 4 and Thm. 4 we have the following general proof rule, representing the two stages, for analysing expected running times.

Theorem 5. (An underlying simple random walk) *Let* $\neg G \rightarrow \mathit{step}$ *and* $(0 < n \leq N) \rightarrow \mathit{walk}'$ *be systems where* walk' *is defined in Lem. 4, and step is loop-free. If rep satisfies the antecedents of Thm. 4 with the condition (†) replaced by the antecedents of Lem. 4, then* [20]

$$\Delta(G, \mathit{step}) \Rrightarrow \mathsf{awp}.\mathit{rep}.\Delta(n = 0, \mathit{walk}') \ .$$

To illustrate Thm. 5 we consider again the programs in Fig. 4. First we note that 2-state-walker is in fact a special case of program *walk* with $N = 1$ and $p = q = 1/2$. Now to apply the theorem we see that $n^+ \mathrel{\hat{=}} \underline{0}$, $n^- \mathrel{\hat{=}} [n = 1]$ and $e \mathrel{\hat{=}} [n = 0]$. Thus we must show that both $\mathsf{dwp}.(\text{2-coins}; n := (x = y)).[n = 0]$ and $\mathsf{dwp}.(\text{2-coins}; n := (x = y)).[n = 1]$ are at least $1/2$, facts which follow immediately. Again we can deduce that $\Delta(\text{2-coins})$ is bounded above by $2[x = y]$.

Overall theorems like Thm. 5 can be formulated for any appropriate random walk, although proving them mechanically can be quite tricky. However, once proved they "pay for themselves" for complicated protocols, since the conditions are easier to verify than those in Thm. 4. Our experiments have shown that using Thm. 5 can decrease the effort for the mechanical proof by about a factor of 2.[21]

Finally we note that since Lem. 3 represents a first-order formulation for probabilistic refinement, we expect it to be of general interest. We are not aware of any other such first-order refinement rule for probabilistic programs.

[20] When $p = q$, the expected time $\Delta(n = 0, \mathit{walk}')$ is exactly $N(N + 1)/4q^2$ for $n = N$ initially.

[21] We analysed a variation on Herman's ring described in [19] using both approaches. Using a theorem similar to Thm. 5 significantly reduced the amount of arithmetic needed in the proofs.

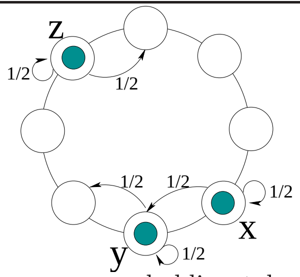
6. Example: Herman's ring

In this example we analyse Herman's probabilistic self-stabilisation [11], a distributed algorithm that can be used for leadership election in a ring of synchronous processors.

A single token is to circulate at all times in a ring of N identical processors; if more than one token is circulating in the ring, the following algorithm is applied to return the state of the ring to that of a single circulating token. On each step, each token-holding processor, synchronously with the others, flips a coin to decide whether to keep its token or pass it clockwise to the next processor. If two tokens collide — a token-holding processor receives a token from the next-anticlockwise processor — the tokens are annihilated. The process continues until only one processor holds a token. For the protocol to succeed, the number of processors initially holding tokens must be odd since the tokens are removed from the ring two at a time.

In our analysis we will assume that the initial configuration of the ring is such that only three tokens are being passed around in the ring (see Fig. 5). Eventually two of the tokens collide, that is one of the distances xy, yz, or

```
Herman-ring $\hat{=}$ $x \neq y \neq z \rightarrow$
        skip ${}_{1/2}\oplus$ $x := (x+1)$ mod $N$;
        skip ${}_{1/2}\oplus$ $y := (y+1)$ mod $N$;
        skip ${}_{1/2}\oplus$ $z := (z+1)$ mod $N$
```



The ring has N processors. x, y, and z represent the processors holding tokens. The algorithm terminates when two of the tokens collide.

Fig. 5: A program for Herman's ring with three tokens.

zx becomes 0. In fact, the behaviour of these distances is all that matters when calculating the running times; hence, we proceed by analysing the random walk that the distances perform themselves. We then prove that this random walk is a correct abstraction of the ring program with respect to running times.

As the processes decide whether to keep or pass the tokens, the distances xy, yz, and zx increase, decrease, or remain unchanged with some probability. For example, xy increases with probability 1/4 if x keeps the token and y passes it; decreases with probability 1/4 if x passes the token and y keeps it; and remains unchanged with probability 1/2 if both x and y keep the tokens or both pass them. The behaviour of yz, and zx is analysed in a similar fashion. The resulting random walk is described in Fig. 6.

$$
\begin{array}{lll}
\text{Herman-walk} \mathrel{\hat{=}} 0 < xy \wedge 0 < yz \wedge 0 < zx \rightarrow & & \\
& \mathsf{skip} & @1/4 \\
& \mid xy := xy - 1; yz := yz + 1 & @1/8 \\
& \mid xy := xy + 1; yz := yz - 1 & @1/8 \\
& \mid yz := yz - 1; zx := zx + 1 & @1/8 \\
& \mid yz := yz + 1; zx := zx - 1 & @1/8 \\
& \mid zx := zx - 1; xy := xy + 1 & @1/8 \\
& \mid zx := zx + 1; xy := xy - 1 & @1/8
\end{array}
$$

Fig. 6: An abstracted random walk for Herman's ring with three tokens, containing distance variables.

We prove that $4(xy \times yz \times zx)/(xy + yz + zx)$ is a weak invariant for this walk.

Next we provide an abstraction invariant for the data refinement:

$$
\begin{array}{ll}
\text{Herman-rep} \mathrel{\hat{=}} & xy := \mathsf{distance}(x, y, N); \\
& yz := \mathsf{distance}(y, z, N); \\
& zx := \mathsf{distance}(z, x, N)
\end{array}
$$

where distance is defined to take into consideration the ring structure; for example, one can prove that the three distances sum to N. We apply Thm. 4 and Thm. 3, and discharge the verification conditions interactively to finally get the bound for the ring program:

$$\mathsf{awp}.\text{Herman-rep}.(4(xy \times yz \times zx)/N)\ . \tag{9}$$

This formula expresses the upper bound on the ring's average running time as a function of the initial distances between the processors holding tokens and the number of processors. It can be easily seen that the more evenly-distributed the tokens are initially, the more time will be needed for the leader to be elected.

Finally, we compare our results on the worst average stabilising time with those verified using the PRISM model checker, which has also been used to analyse this protocol [14, 23]. Tab. I summarises these results. The PRISM verification has been carried out for rings with up to 17 processors. The HOL results are calculated by instantiating the bound we proved (9) so that the tokens are distributed as evenly as possible.

Perhaps the most interesting observation is that although the PRISM results in Tab. I are given as a *worst case* analysis (that is, over all possible initial configurations of odd coins), the expected running time is no larger

N	PRISM result	HOL result
3	1.333333	1.333333
5	3.199998	3.2
7	6.857138	6.857142
9	11.999993	12
11	17.454534	17.454545
13	24.615369	24.615384
15	33.333312	33.333333
17	42.352913	42.352941
19	-	53.052632
$\vdots$	-	$\frac{4(\lfloor N/3 \rfloor \times \lfloor (N+1)/3 \rfloor \times \lfloor (N+2)/3 \rfloor)}{N}$

$\lfloor a \rfloor$ denotes the floor of a number a.

TABLE I: PRISM and HOL bounds on the maximum expected time for ring stabilisation.

than the maximum expected running time for the case when only three processors start with tokens. Thus the case of three initial tokens seems to deliver the worst possible result for $3 \leq N \leq 17$. It is unknown whether the three-token case is the worst possible for all N [16].

7. Related work and conclusions

Segala et al. [24] have considered techniques for estimating expected running times within I/O automata, and de Alfaro [6] has formulated "expected rewards" in systems where the reward varies depending on the state from which a computation is made. Our technique based on probabilistic data refinement however addresses the problem of improving the efficiency of the analysis; moreover, data refinement means that previously analysed random walks may be reused.

Removing the loop-free condition in *step* remains a topic for further investigation, and indeed our current definition of Δ is not general enough to treat this. To account for nested loops, a general operator for expected termination would need to allow the possibility of varying rewards, so that applied to a loop of the form

$$\mathsf{do}\ G \rightarrow (\mathsf{do}\ G' \rightarrow \mathit{step}'\ \mathsf{od}); \mathit{step}\ \mathsf{od}$$

each iteration of the outer loop would account for the expected termination time for the inner loop.

Another interesting line of research would be to investigate how to find weak invariants for the Δ-analysis; work along these lines has been done by Ernst et al. [8] for generating standard program invariants, and it would be interesting to see whether similar techniques would apply to quantitative invariants.

Acknowledgements

We thank Carroll Morgan who suggested the formula in Sec. 6 for analysing Herman's ring, and for commenting on the manuscript. We also thank Joe Hurd who made many helpful suggestions in respect of our HOL formalisation.

This work was started while Orieta Celiku was visiting Macquarie University's Department of Computing, whose help in facilitating the visit is appreciated.

References

[1] J.-R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] R.-J. Back. *On the Correctness of Refinement Steps in Program Development.* PhD thesis, Department of Computer Science, University of Helsinki, Finland, 1978.

[3] R.-J. Back. Data refinement in the refinement calculus. In *Proceedings of the 22nd Hawaii International Conference of System Sciences*, Jan. 1989.

[4] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.

[5] M. J. Butler, J. Grundy, T. Långbacka, R. Rukšenas, and J. von Wright. The refinement calculator: Proof support for program refinement. In L. Groves and S. Reeves, editors, *Formal Methods Pacific '97*, Discrete Mathematics and Theoretical Computer Science. Springer-Verlag, July 1997.

[6] L. de Alfaro. Temporal logics for the specification of performance and reliability. In *STACS 1997, 14th Annual Symposium on Theoretical Aspects of Computer*, volume 1200 of *Lecture Notes in Computer Science*. Springer, 1997.

[7] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[8] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *International Conference on Software Engineering.* ACM, 2000.

[9] M. J. C. Gordon. Mechanizing programming logics in higher-order logic. In G. M. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, pages 387–439. Springer-Verlag, 1988.

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

[11] T. Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, June 1990.

[12] R. Honsberger. A problem in coin-tossing. *Mathematical Diamonds.* The Mathematical Association of America, 2003.

[13] J. Hurd, A. McIver, and C. Morgan. Probabilistic guarded commands mechanized in HOL. In *Proc. of QAPL 2004*, Mar. 2004.

[14] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In *Proceedings of TOOLS 2002*, volume 2324 of *Lecture Notes in Computer Science*, pages 200–204. Springer, Apr. 2002.

[15] A. McIver and C. Morgan. *Abstraction, refinement and proof for probabilistic systems.* Springer, 2004. To appear.

[16] A. McIver and C. Morgan. An elementary proof that Herman's Ring is $\theta(N^2)$. Available at *http://web.comlab.ox.ac.uk/oucl/research/areas/probs/bibliography.html*, 2004.

[17] A. K. McIver. Quantitative program logic and expected time bounds in probabilistic distributed algorithms. *Theoretical Computer Science*, 282:191–219, 2002.

[18] C. C. Morgan. *Programming from Specifications.* Prentice-Hall, 1990.

[19] C. C. Morgan. Proof rules for probabilistic loops. In *Proceedings of the BCS-FACS 7th Refinement Workshop*, Workshops in Computing. Springer-Verlag, 1996.

[20] C. C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990. Reprinted in [22].

[21] C. C. Morgan, A. K. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, May 1996.

[22] C. C. Morgan and T. N. Vickers, editors. *On the Refinement Calculus*, FACIT Series in Computer Science. Springer-Verlag, 1994.

[23] PRISM case studies: Randomised self-stabilising algorithms. Available at *http://www.cs.bham.ac.uk/~dxp/prism/casestudies/self-stabilisation.html*.

[24] R. Segala, N. Lynch, and I. Saias. Proving time bounds for randomized distributed algorithms. In *13th Annual Symposium on Principles of Distributed Algorithms*, pages 314–323, 1994.

Appendix A. Proof of Lem. 3

Programs in pGCL of the form

$$P \mathrel{\hat{=}} P_1 \ @p_1 \mid P_2 \ @p_2 \mid \ldots \mid P_n \ @p_n \ , \tag{10}$$

are determined by predicates, where $P_1, \ \ldots, P_n$ are standard (probabilistic-choice free), terminating programs.

To prove this result, we must show that given a pGCL program P,

$$P \sqsubseteq Q \quad \text{iff} \quad \mathsf{dwp}.P.[post_i^s].s \leq \mathsf{dwp}.Q.[post_i^s].s \ , \ 1 \leq i \leq m_s \ , \tag{11}$$

where for each s, $post_i^s$ $(1 \leq i \leq m_s)$ is a finite set of predicates (depending on s and P, but not Q).

To avoid clutter, we fix initial state s and drop the s super- and subscripts — thus for instance we write $\mathsf{dwp}.r.E = E'$ instead of $\mathsf{dwp}.r.(E^s).s = E'.s$ We also define

$$B \subseteq C \quad \hat{=} \quad (\forall s \bullet B.s \Rightarrow Q.s) \ .$$

for predicates B and C, and we identify a predicate with the subset of states for which it is true.

We prove (11) by constructing the determining predicates for P (and fixed initial state s). The detailed reasoning is set out in the steps below together with either a proof, or a citation in the cases where proofs occur elsewhere.

(1) For any standard (probabilistic-choice free), terminating program *Prog* there is an associated predicate *pred* such that $\mathsf{dwp}.Prog.[post] = [pred \subseteq post]$, for any predicate *post*. (Back and von Wright [4] for the selection of *pred*, and Morgan and McIver [15] for the embedding of predicate transformers into expectation transformers.)

(2) For standard program *Prog* and expectation E, $\mathsf{dwp}.Prog.E = e \times [pred]$, where *pred* is *Prog*'s associated determining predicate from (step 1) and e is the largest scalar such that $e \times [pred] \Rrightarrow E$. (McIver and Morgan [15].)

(3) To prove the refinement at equation (11), we may assume that the expectation E in Def. 2 has finite range, i.e. that the set $\{E.s \mid s : S\}$ is finite. This follows from continuity of $\mathsf{dwp}.P$. (McIver and Morgan [15].)

(4) Now let P be the program defined at equation (10). Let $\mathit{pred}_1,\ \mathit{pred}_2, \ldots, \mathit{pred}_k$ be the associated predicates given at (step 1), respectively for the programs $P_1,\ P_2, \ldots, P_k$. The following equality holds for any predicate B:

$$\mathsf{dwp}.P.[B] \ = \ \mathsf{dwp}.P.[\bigvee_{\mathit{pred}_i \subseteq B} \mathit{pred}_i] \ ,$$

where $\vee_{\mathit{pred}_i \subseteq B}\, \mathit{pred}_i$ denotes the disjunction over all $1 \leq i \leq n$ such that $\mathit{pred}_i \subseteq B$.

Proof: We reason as follows:

$$\begin{array}{lll}
 & \mathsf{dwp}.P.[B] & \\
= & \mathsf{dwp}.(P_1 \ @p_1 \mid P_2 \ @p_2 \mid \ldots \mid P_n \ @p_n).[B] & \\
= & \sum_{1\leq i\leq n} p_i \times \mathsf{dwp}.P_i.[B] & \text{Definition "}@p_i\text{", Fig. 2} \\
= & \sum_{1\leq i\leq n} p_i \times [\mathit{pred}_i \subseteq B] & \text{(step 1) above} \\
= & \sum_{1\leq i\leq n} p_i \times [\mathit{pred}_i \subseteq \vee_{\mathit{pred}_j \subseteq B}\, \mathit{pred}_j] & \text{Predicate calculus} \\
= & \mathsf{dwp}.P.[\vee_{\mathit{pred}_i \subseteq B}\, \mathit{pred}_i] \ . & \text{Definition "}@p_i\text{", Fig. 2}
\end{array}$$

We continue now to show that the set of determining predicates for P (at fixed initial state s) is $\{\vee_{i:\, I}\, \mathit{pred}_i \mid I \subseteq \{1, \ldots, n\}\}$.

(5) Given an expectation E, by (step 3) we may assume that it takes distinct values $e_1 < e_2 < \cdots < e_k$. We construct a set of predicates as follows. For each $1 \leq i \leq k$,

$$B_i \mathrel{\hat{=}} \{s : S \mid E.s \geq e_i\} \ .$$

We note that $B_1 \supseteq B_2 \supseteq \cdots \supseteq B_k$, and that $e_i \times [B_i] \Rrightarrow E$.

(6) Using the definitions from (step 5), we define

$$E' \mathrel{\hat{=}} e_1 \times [B_1] + (e_2 - e_1) \times [B_2] + \cdots + (e_k - e_{k-1}) \times [B_k] \ ,$$

and note that by construction $E' \equiv E$. We define non-negative scalars $e'_1 \mathrel{\hat{=}} e_1$, and $e'_i \mathrel{\hat{=}} (e_i - e_{i-1})$ for $2 \leq i \leq k$.

(7) For any standard program Prog, we have that

$$\mathsf{dwp}.\mathit{Prog}.E = \sum_{1\leq i\leq k} e'_i \times \mathsf{dwp}.\mathit{Prog}.[B_i] \ .$$

Proof: We show first that the RHS $\geq$ LHS — we assume that LHS $>$ 0, otherwise the result is trivial. Observe first from (step 2) that $\mathsf{dwp}.\mathit{Prog}.E = e_j$ for some j where $e_j \times [\mathit{pred}] \Rrightarrow E$, and pred is P's determining predicate. We see now from the construction in (step 5) that $\mathit{pred} \subseteq B_j$, from which it follows that

$$\sum_{1\leq i\leq k} e'_i \times \mathsf{dwp}.\mathit{Prog}.[B_i] = \sum_{1\leq i\leq k} e'_i \times [\mathit{pred} \subseteq B_i] \geq e_j \ = \mathsf{dwp}.\mathit{Prog}.E \ ,$$

where the first equality follows from (step 1) and the nesting of the B_i's from (step 5) and the definition of the e'_i.
To show that LHS $\geq$ RHS, we simply observe that $\sum e'_i \times \mathsf{dwp}.\mathit{Prog}.[B_i] \leq \mathsf{dwp}.\mathit{Prog}.(\sum e'_i \times [B_i]) = \mathsf{dwp}.\mathit{Prog}.E$, which follows from sub-additivity of $\mathsf{dwp}.\mathit{Prog}$ (Morgan et al. [21]) and then (step 6) above, concluding the proof.

(8) Now let Q be a pGCL program such that

$$\mathsf{dwp}.P.[(\bigvee_{i:\,I} \mathit{pred}_i)] \leq \mathsf{dwp}.Q.[(\bigvee_{i:\,I} \mathit{pred}_i)] \ ,$$

for all subsets I of $\{1, \ldots, n\}$. We show in that case that $P \sqsubseteq Q$.
Proof: We reason as follows. Let E be an expectation with finite range.

$$\begin{array}{lll}
 & \mathsf{dwp}.P.E & \\
= & \sum_{1\leq i\leq n} p_i \times \mathsf{dwp}.P_i.E & \text{Definition “@}p_i\text{”, Fig. 2} \\
= & \sum_{1\leq i\leq n} p_i \times (\sum_{1\leq j\leq k} e'_j \times \mathsf{dwp}.P_i.[B_j]) & \text{(step 7); } P_i \text{ is standard} \\
= & \sum_{1\leq j\leq k} e'_j \times (\sum_{1\leq i\leq n} p_i \times \mathsf{dwp}.P_i.[B_j]) & \text{Arithmetic} \\
= & \sum_{1\leq j\leq k} e'_j \times \mathsf{dwp}.P.[B_j] & \text{Definition “@}p_i\text{”, Fig. 2} \\
= & \sum_{1\leq j\leq k} e'_j \times (\mathsf{dwp}.P.[\vee_{\mathit{pred}_i \subseteq B_j} \mathit{pred}_i]) & \text{(step 4); } B_j \text{ is a pred.} \\
\leq & \sum_{1\leq j\leq k} e'_j \times (\mathsf{dwp}.Q.[\vee_{\mathit{pred}_i \subseteq B_j} \mathit{pred}_i]) & \text{Assumption} \\
\leq & \mathsf{dwp}.Q.(\sum_{1\leq j\leq k} e'_j \times [B_j]) & \mathsf{dwp}.Q \text{ is sub-additive, [21]} \\
= & \mathsf{dwp}.Q.E \ , & \text{(step 6)}
\end{array}$$

as required. Lem. 3 now follows from Def. 2 and (step 3), since E was chosen arbitrarily.