Verifying Probabilistic Correctness in Isabelle with pGCL

David Cock
NICTA
Sydney, Australia*
School of Computer Science and Engineering
University of New South Wales
David.Cock@nicta.com.au

This paper presents a formalisation of pGCL in Isabelle/HOL. Using a shallow embedding, we demonstrate close integration with existing automation support. We demonstrate the facility with which the model can be extended to incorporate existing results, including those of the L4.verified project. We motivate the applicability of the formalism to the mechanical verification of probabilistic security properties, including the effectiveness of side-channel countermeasures in real systems.

1 Introduction

We present a new formalisation of the pGCL programming logic within Isabelle/HOL [14]. The motivation for this work is the desire for formal, mechanised verification of probabilistic security properties (in particular, bounds on side-channel vulnerability) for real systems. We build on existing theoretical work on pGCL [11], and present a complementary approach to the existing formalisation in HOL4 [8].

Our contribution is a shallow embedding of pGCL within Isabelle/HOL, using the unmodified real-number type for probabilities, leading to excellent support for proof automation and easy extensibility to tackle novel problems, and to integrate with existing results (including those of L4.verified [9]). We demonstrate these by example.

The structure of the paper is as follows: We first motivate the need for probabilistic reasoning in systems (Section 2), and in particular the need for probabilistic (Section 2.2) and refinement-sound (Section 2.3) security properties. We then present our pGCL theory package (Section 3), giving a cursory introduction to its syntax and underlying model (Section 3.1), and demonstrate the use of the package (Section 3.2), giving examples of 3 styles of proof. We next provide more detail on the underlying implementation (Section 4), touching on the specifics of the shallow embedding (Section 4.1), extending it into novel contexts (Section 4.2), the lattice structure of the semantic models that underlie recursion (Section 4.3), and finally the implementation of the VCG (Section 4.4). Surveying related work (Section 5), we conclude by describing the ongoing efforts in applying the tool (Section 6).

2 The Case for Probabilistic Correctness

Recent work [9] has demonstrated the practicality of verifying the functional correctness of real systems. Functional correctness, however, covers only properties that are *guaranteed* to hold. This excludes classes of properties relevant in practice, including certain execution-time and security guarantees. By

*NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

F. Cassez, R. Huuck, G. Klein and B. Schlich (eds.):
Systems Software Verification Conference 2012 (SSV 2012)
EPTCS 102, 2012, pp. 167–178, doi:10.4204/EPTCS.102.15

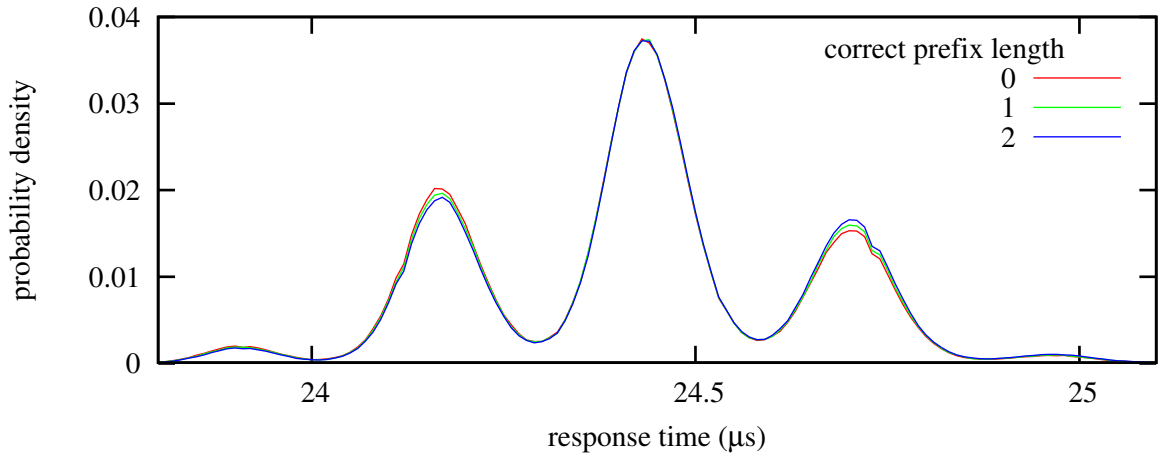

Figure 1: String compare (`strcmp`) execution time distribution

allowing security properties that only hold with some probability, we can give a more nuanced classification of systems than that arising from a functional property such as noninterference [6].

2.1 Probabilistic Behaviour in Systems

In formal specification, it is convenient to treat a system as fully predictable, and make concrete truth claims about its behaviour. Once implemented, while the system may be in principle predictable, the size and complexity (and often under-specification) of the state space makes a full description impractical.

The usual approach in this case is to retreat to a probabilistic model of the system, informed by benchmarking. That this is true is apparent on reading the evaluation section of a systems paper: While the precise performance of a system is theoretically predictable and could thus be calculated without empiricism, what we see are benchmarks and histograms. The tools of the evaluator are experiments and statistics! This is a testament to the immense difficulty of precisely predicting performance. Once real world phenomena intrude, as in networking, exact prediction becomes impossible, even in theory.

Moreover, there exist properties which can only be treated with statistical methods. If such properties are critical for correctness, then a proof will, of necessity, involve probabilistic reasoning. Execution time is an example. Figure 1 is representative, giving the response time distribution for the C `strcmp` routine for a fixed secret and a varying guess[1]. Note that within the measurement precision ($10ns$) the distribution is essentially continuous, and depends on a sensitive datum: the longest common prefix between the secret and the guess. If we are concerned with security, this is a correctness-critical property that is unavoidably probabilistic. This is a simple side-channel, vulnerable to a guessing attack.

2.2 Security Properties

As stated, our motivation is the verification of probabilistic security properties, given a concrete attack model. Building on the above example, imagine that an adversary is supplying guesses in decreasing order of probability, and updating this order based on observed response time. A possible security property then, is that the adversary does not guess the true secret in fewer than, say, n attempts.

[1] This figure first appeared in a previous work [3], where it is discussed in more depth.

For a functional security property, we would now calculate the set of initial states that guarantee that our property will hold in the final state: its *weakest precondition* (wp). Security is assured by showing that the initial state lies in this set.

In a probabilistic system, however, it could easily be the case that from *any* initial state, there is a nonzero probability that the final state is insecure, even if the overall chance of this occurring is small. This would be the case if the attacker guessed completely at random. The best we could then hope for is to find the *probability* that the final state is secure.

What we need is a probabilistic analogue of the weakest precondition. Consider the functional weakest precondition as a $\{0,1\}$-valued function (identifying it with its selector). We then ask whether we can define a $[0,1]$-valued function, that instead of answering "secure" or "not secure" for a state, answers "secure with probability p". We could then show that the system remains secure with probability $\geq p$ by showing that the initial state lies in the set for which wp $\geq p$. In the probabilistic programming logic pGCL of McIver & Morgan [11] we find just such an analogue.

2.3 Refinement and Security

The standard problem in applying refinement to security is in treating nondeterminism. Writing $a\,;;\,b$ for the sequential composition of programs a and b, and $c \sqcap d$ for a demonically nondeterministic choice between c and d, consider the following fragment:

$$h := \text{secret}\,;;\,(l := 0 \sqcap l := 1) \quad \text{where} \quad \text{secret} \in \{0,1\}\ .$$

Let h (high) be hidden, and l (low) be globally visible. We wish to formalise the intuitive security property: 'the value in l doesn't reveal the value in h'. Writing $a \sqsubseteq b$ for 'b refines a', taken to mean that every trace of b is also a trace of a, the following is a valid refinement:

$$h := \text{secret}\,;;\,l := h\ ,$$

which clearly violates the security property. Any attempt to verify a property such as $\forall s_{\text{final}}.\ l \neq h$ for this program fails due to such insecure refinements.[2]

For a property Q to survive refinement, writing $\vdash$ for predicate entailment, we need that

$$\frac{a \sqsubseteq b}{\text{wp}\ a\ Q \vdash \text{wp}\ b\ Q} \qquad \text{whence by transitivity,} \qquad \frac{a \sqsubseteq b \quad P \vdash \text{wp}\ a\ Q}{P \vdash \text{wp}\ b\ Q}\ .$$

Under an appropriate model (e.g. wp $(a \sqcap b)\ P\ s = \text{wp}\ a\ P\ s \wedge \text{wp}\ b\ P\ s$), these relations hold for the above program and security predicate, but show the hopelessness of the case as:

$$\forall s.\ \neg\,\text{wp}\ (h := \text{secret}\,;;\,(l := 0 \sqcap l := 1))\ (l \neq h)\ s\ .$$

In other words the weakest precondition of $l \neq h$, considered as a set, is empty, and thus the original specification cannot satisfy the security property. Pessimistically, on any trace the program might set $l = h$. It is critical to establish that a desired security property is preserved by refinement, in order that such insecure specifications are exposed.

[2] With a state space of two, of course, such an anti-correlation would be just as insecure as $l = h$. In a large space however, knowing $l \neq h$ leaks very little information. We are not establishing that l and h are uncorrelated, rather that in a given trace they did not happen to coincide. When we consider guessing attacks, this will be precisely the formulation we need: Whether or not the attacker simply got lucky is irrelevant, as the system is still compromised.

Our formalism of choice is pGCL, where we have a novel notion of entailment. We write[3] $P \Vdash Q$ for comparison defined pointwise i.e. $\forall s.\ P\ s \leq Q\ s$. If $P \Vdash Q$, then P has a lower value in every state than does Q. Write[4] «P» for the boolean predicate P as real-valued function (an *expectation*):

$$«P»\ s = \text{if } P\ s \text{ then } 1 \text{ else } 0 \quad \text{noting that} \quad P \vdash Q \leftrightarrow «P» \Vdash «Q»$$

Considering again the example of Section 2.2, we model a guessing attack on h as

$$(h := 0 \sqcap h := 1)\ ;;\ (l := 0\ {}_{1/2}\!\oplus\ l := 1)\ , \tag{1}$$

where the secret (h) is chosen nondeterministically, and the guess (l) randomly ($a\ {}_p\!\oplus\ b$ denotes probabilistic choice between programs a and b, with probability p for a).

We have structural refinement rules, for example $a \sqcap b \sqsubseteq a$, and the following relations:

$$\frac{\text{WP\_REFINES}}{\dfrac{a \sqsubseteq b}{\text{wp}\ a\ Q \Vdash \text{wp}\ b\ Q}} \quad \text{whence} \quad \frac{a \sqsubseteq b \quad P \Vdash \text{wp}\ a\ Q}{P \Vdash \text{wp}\ b\ Q}\ ,$$

which are the probabilistic equivalents of our refinement conditions. We have, therefore, that $h := 0\ ;;\ (l := 0\ {}_{1/2}\!\oplus\ l := 1)$ is a refinement of Equation 1, but importantly, $(h := 0 \sqcap h := 1)\ ;;\ l := h$ is not. In contrast to demonic choice, probabilistic choice cannot be refined away.

A refinement in pGCL establishes a predicate with *higher* probability than does its specification. Thus if we arrange our predicate on final states as 'the system is secure', and we are content with establishing the minimum probability with which it is established, refinement *by definition* can only increase this probability. Our guessing-game model is thus refinement-sound in pGCL.

3 The pGCL Theory Package

To automate such reasoning, we present a shallow embedding of pGCL in Isabelle/HOL. With a few noted exceptions, we preserve the syntax of McIver & Morgan. The advantage of our approach is the ease with which we can apply the existing machinery of Isabelle/HOL to the underlying arithmetic. The disadvantage is that we cannot appeal to the axioms of a stricter type, in particular Isabelle's fixed-point lemmas, which reside in the class of complete lattices. This is dealt with in detail in Section 4.

The language models imperative computation, extended with both nondeterministic and probabilistic choice. The non-probabilistic component corresponds to Dijkstra's guarded command language, GCL [5]. Nondeterminism (by default) is demonic, with respect to the postcondition of a program: A demonic choice minimises the likelihood of establishing the postcondition.

Sequential composition (;;), demonic choice ($\sqcap$) and probabilistic choice ($_p\!\oplus$) were introduced in Section 2.3. The remainder consists of: Abort and Skip, representing failure and a null operation, respectively; Apply, which embeds a state transformer; and the recursive primitive, μ.

3.1 The Expectation-Transformer Model

The intuitive interpretation of a program, its operational semantics, is as a probabilistic automaton. From a given state, the program chooses the next randomly, with a probability depending on the state. The program is thus a *forward* transformer, taking a distribution on initial states to one on final states.

[3] We differ in syntax, as the symbol $\Rrightarrow$ is not readily available within Isabelle

[4] Again we differ, as the established syntax, $[\cdot]$, clashes with lists.

$$\begin{array}{rcl@{\qquad}rcl}
\text{wp Abort } R & = & \lambda s.\ 0 & \text{wp } (a\,;;b)\ R & = & \text{wp } a\ (\text{wp } b\ R) \\
\text{wp Skip } R & = & R & \text{wp } (a\ {}_p\!\oplus b)\ R & = & \lambda s.\ p\ s \times \text{wp } a\ R\ s + (1-p\ s) \times \text{wp } b\ R\ s \\
\text{wp (Apply } f)\ R & = & \lambda s.\ R\ (f\ s) & \text{wp } (a \sqcap b) R & = & \lambda s.\ \min\ (\text{wp } a\ R\ s)\ (\text{wp } b\ R\ s)
\end{array}$$

Figure 2: Structural *wp* rules.

For mechanisation, we are more interested in the axiomatic interpretation of a program: as a *reverse* transformer. Here, the program maps a function on the final state to one on the initial state: a 'real-valued predicate'. These generalised predicates are the *expectations* of Section 2.3, and are the bounded, nonnegative functions from the state space S, to $\mathbb{R}$:

$$\begin{array}{rcl@{\qquad}rcl}
\text{bounded_by } b\ P & \equiv & \forall s.\ P\ s \leq b & \text{bounded } P & \equiv & \exists b.\ \text{bounded_by } b\ P \\
\text{nneg } P & \equiv & \forall s.\ 0 \leq P\ s & \text{sound } P & \equiv & \text{bounded } P \wedge \text{nneg } P
\end{array}$$

The *strict* interpretation of non-recursive constructions is given in Figure 2. The *liberal* interpretation differs only for Abort and μ, and is explained in Section 4.1. That the forward and reverse interpretations are equivalent is established [11], although we have not mechanised the proof.

To see that this model produces the probabilistic weakest precondition demanded in Section 2, note that the expectation $\langle\!\langle R \rangle\!\rangle : S \to \mathbb{R}$, applied to final states, gives the probability that the predicate R holds. This interpretation is preserved under transformation: wp prog $\langle\!\langle R \rangle\!\rangle\ s_{\text{initial}}$ is the probability that R will hold in the final state, if prog executes from s_{initial}. Equivalently, it is the current expected value of the predicate on the final state: $\sum_{s_{\text{final}}} P(s_{\text{final}} | s_{\text{initial}}) \times \langle\!\langle R \rangle\!\rangle\ s_{\text{final}}$, hence the term expectation.

A program is represented as a function from expectation to expectation: $(S \to \mathbb{R}) \to S \to \mathbb{R}$. The function maps post-expectation to weakest pre-expectation. For a *standard* post-expectation, the embedding of a predicate (e.g. $\langle\!\langle P \rangle\!\rangle$), this is the greatest lower bound on the likelihood of it holding, finally.

For our inference rules to apply, transformers must satisfy several *healthiness* properties, which are slightly weaker than the standard versions [10]. We combine the treatment of strict transformers (weakest precondition, giving total correctness) and liberal transformers (weakest *liberal* precondition, giving partial correctness) by working in the union of their domains. This basic healthiness is defined as the combination of *feasibility*, *monotonicity* and *weak scaling*:

$$\begin{array}{c}
\text{feasible } t \equiv \forall P\, b.\ \text{bounded_by } b\ P \wedge \text{nneg } P \to \text{bounded_by } b\ (t\ P) \wedge \text{nneg } (t\ P) \\
\text{mono_trans } t \equiv \forall P\, Q.\ (\text{sound } P \wedge \text{sound } Q \wedge P \Vdash Q) \longrightarrow (t\ P) \Vdash (t\ Q) \\
\text{scaling } t \equiv \forall P\, c\, s.\ (\text{sound } P \wedge 0 < c) \longrightarrow c \times t\ P\ s = t\ (\lambda s.\ c \times P\ s)\ s
\end{array}$$

Stronger results are established on-the-fly by appealing to one of several supplied rule-sets.

A well-defined program has healthy strict and liberal interpretations, related appropriately:

$$\text{well\_def } a \equiv \text{healthy } (\text{wp } a) \wedge \text{healthy } (\text{wlp } a) \wedge (\forall P.\ \text{sound } P \longrightarrow \text{wp } a\ P \Vdash \text{wlp } a\ P)$$

3.2 Reasoning with pGCL

The rules in Figure 2 evaluate the weakest pre-expectation of non-recursive program fragments structurally. If the exponential growth of the term is not problematic, the simplifier can calculate it exactly. We also show two other approaches supported by the package: Modular reasoning by structural decomposition, and the verification condition generator (VCG). Examples are given in Isabelle proof script.

$$\text{hide} \equiv \text{Apply}\ (\lambda s.\ s(\!|\text{P} := 1|\!)) \sqcap \text{Apply}\ (\lambda s.\ s(\!|\text{P} := 2|\!)) \sqcap \text{Apply}\ (\lambda s.\ s(\!|\text{P} := 3|\!))$$

$$\text{guess} \equiv \text{Apply}\ (\lambda s.\ s(\!|\text{G} := 1|\!))\ {}_{\lambda s.\ 1/3}\oplus \Big(\text{Apply}\ (\lambda s.\ s(\!|\text{G} := 2|\!))\ {}_{\lambda s.\ 1/2}\oplus \text{Apply}\ (\lambda s.\ s(\!|\text{G} := 3|\!))\Big)$$

$$\text{reveal} \equiv \bigsqcap d \in (\lambda s.\ \{1,2,3\} - \{\text{P}\ s, \text{G}\ s\}).\ \text{Apply}\ (\lambda s.\ s(\!|\text{C} := d|\!))$$

$$\text{switch} \equiv \bigsqcap d \in (\lambda s.\ \{1,2,3\} - \{\text{C}\ s, \text{G}\ s\}).\ \text{Apply}\ (\lambda s.\ s(\!|\text{G} := d|\!))$$

$$\text{monty}\ s \equiv \text{hide};; \text{guess};; \text{reveal};; \text{if}\ s\ \text{then switch else Skip}$$

Figure 3: The Monty Hall game in pGCL

Consider Figure 3, a model of the Monty Hall problem [8, 15]. Briefly, the context is a game show: A prize is hidden behind one of three doors (hide), of which the contestant then guesses one (guess). The host then opens a door other than the one the contestant has chosen (reveal), showing that it does not hide the prize. The contestant now has a choice: to switch to the unopened door (switch), or to stick to the original (Skip). Is the contestant better off switching?

Figure 3 introduces a new primitive: demonic choice over a set:

$$\bigsqcap x \in \{a, b, \ldots\}.\ p\ x \equiv p\ a \sqcap p\ b \sqcap \ldots \qquad \text{wp}\ \Big(\bigsqcap x \in S.\ p\ x\Big)\ R\ s = \inf\{\text{wp}\ (p\ x)\ R\ s \mid x \in S\ s\}$$

The ability to extend the language in this way is a benefit of the shallow embedding. We will return to this point in Section 4.2.

We first define the victory condition for the game, whence the least probability of the contestant winning, over all choices by the (demonic) host, and from a given starting state s, is then given by the weakest precondition:

$$\text{win}\ g \equiv (\text{G}\ g = \text{P}\ g) \qquad P_{\min}(\text{win}) = \text{wp}\ (\text{monty}\ \mathit{switch})\ \text{«win»}\ s$$

Proof by unfolding When *switch* is false, we solve by explicitly unfolding the rules in Figure 2. This approach was demonstrated by Hurd et. al. [8] As expected, the contestant has a $1/3$ chance of success:

```
lemma wp_monty_noswitch: "λs. 1/3 ⊨ wp (monty false) «wins»"
  unfolding monty_def hide_def guess_def reveal_def switch_def
  by(simp add:wp_eval insert_Diff_if)
```

Proof by decomposition If *switch* is true the state space grows dramatically, and such a straightforward proof becomes computationally expensive, although it is still just possible in this case. To scale to still larger examples, we must employ modular reasoning. Fortunately, weakest preconditions in pGCL admit

composition rules very similar to their counterparts for GCL:

$$\textsc{wp\_strengthen\_post}\quad \frac{P \Vdash \text{wp}\ p\ Q \quad Q \Vdash R \quad \text{healthy}\ (\text{wp}\, p) \quad \text{sound}\ Q \quad \text{sound}\ R}{P \Vdash \text{wp}\ p\ R}$$

$$\textsc{wp\_Seq}\quad \frac{Q \Vdash \text{wp}\ b\ R \quad P \Vdash \text{wp}\ a\ Q \quad \text{healthy}\ (\text{wp}\ a) \quad \text{healthy}\ (\text{wp}\ b) \quad \text{sound}\ Q \quad \text{sound}\ R}{P \Vdash \text{wp}\ (a;;b)\ R}$$

The healthiness and soundness obligations result from the shallow embedding. The rules are otherwise identical. To integrate with Isabelle's calculational reasoner [1], we define probabilistic Hoare triples:

$$\textsc{wp\_validI}\quad \frac{P \Vdash \text{wp}\ a\ Q}{\{P\}\ a\ \{Q\}} \qquad \textsc{wp\_validD}\quad \frac{\{P\}\ a\ \{Q\}}{P \Vdash \text{wp}\ a\ Q}$$

$$\textsc{valid\_Seq}\quad \frac{\{P\}\ a\ \{Q\} \quad \{Q\}\ b\ \{R\} \quad \text{healthy}\ (\text{wp}\ a) \quad \text{healthy}\ (\text{wp}\ b) \quad \text{sound}\ Q \quad \text{sound}\ R}{\{P\}\ a;;b\ \{R\}}$$

Note that VALID_SEQ is simply the composition of WP_COMPOSE with WP_VALIDI and WP_VALIDD. We need one final rule, peculiar to pGCL, which illustrates the real-valued nature of expectations:

$$\textsc{wp\_scale}\quad \frac{P \Vdash \text{wp}\ a\ Q \quad \text{healthy}\ (\text{wp}\ a) \quad \text{sound}\ Q \quad 0 < c}{(\lambda s.\ c \times P\ s) \Vdash \text{wp}\ a\ (\lambda s.\ c \times Q\ s)}$$

This follows from the healthiness of the transformer, and allows us to scale the pre- and post-expectations such that the latter 'fits under' some target. To illustrate, consider the 'obvious' specification of hide:

$$\lambda s.\ 1 \Vdash \text{wp hide}\ «\lambda s.\ P\ s \in \{1,2,3\}» \tag{2}$$

This states that with probability 1, the prize ends up behind door 1, 2 or 3. In evaluating our preconditions stepwise, however, we find that the weakest precondition of the remainder of the program is in fact:

$$\lambda s.\ \ 2/3 \times «\lambda s.\ P\ s \in \{1,2,3\}»\ s\ .$$

Applying rule WP_SCALE to Equation 2 we derive a scaled rule:

$$\lambda s.\ 2/3 \Vdash \text{wp hide}\ (\lambda s.\ 2/3 \times «\lambda s.\ P\ s \in \{1,2,3\}»\ s)\ .$$

Using this, we see that the probability of success if the contestant switches is at least[5] 2/3:

```
declare valid_Seq[trans]
lemma wp_monty_switch_modular: "λs. 2/3 ⊩ wp (monty true) «wins»"
  proof(rule wp_validD)
    note wp_validI[OF wp_scale, OF wp_hide, simplified]
    also note wp_validI[OF wp_guess]
    also note wp_validI[OF wp_reveal]
    also note wp_validI[OF wp_switch]
    finally show "λs. 2/3 ⊩ wp (monty true) «wins»"
      unfolding monty_def by(simp add:healthy_intros sound_intros monty_healthy)
  qed
```

[5] In fact it is exactly 2/3, but our object is to demonstrate an entailment proof. In more complicated situations, calculating the exact pre-expectation is impractical.

$$a;;b \equiv \lambda ab.\ (a\ ab) \circ (b\ ab) \qquad \text{Abort} \equiv \lambda ab\,P.\ \text{if } ab \text{ then } \lambda s.\ 0 \text{ else } \lambda s.\ \text{bound\_of } P$$

$$\text{Embed } f \equiv \lambda ab.\ f \qquad \mu x.\ \text{prog } x \equiv \lambda ab.\ \text{if } ab \text{ then lfp\_trans } (\lambda t.\ \text{prog } (\text{Embed } t)\ ab) \text{ else gfp\_trans } (\lambda t.\ \text{prog } (\text{Embed } t)\ ab)$$

$$\text{wp } a \equiv a\ \text{True} \qquad \text{wlp } a \equiv a\ \text{False}$$

Figure 4: The underlying definitions of selected primitives

As mentioned, we take advantage of the calculational reasoning facility of Isabelle. The intermediate Hoare triples are automatically derived, by applying VALID_SEQ to the previous relation and the supplied specification. Finally, we again discharge all side-conditions using the simplifier.

Proof by VCG Finally, we can pass the component specifications to our verification condition generator (VCG), which follows a similar strategy to the above, automatically matching the appropriate rule to the goal. The VCG leaves an inequality between the target pre-expectation and that calculated internally (generally not the weakest). In this case, the final goal is trivial enough to be discharged internally.

```
lemmas scaled_hide = wp_scale[OF wp_hide, simplified]
declare scaled_hide[wp] wp_guess[wp] wp_reveal[wp] wp_guess[wp]
declare healthy_wp_hide[health] healthy_wp_guess[health]
        healthy_wp_reveal[health] healthy_wp_switch[health]
lemma wp_monty_switch_vcg: "λs. 2/3 ⊩ wp (monty true) «wins»"
    unfolding monty_def by(simp,pvcg)
```

3.3 Loops and Recursion

Recursive programs cannot be evaluated by unfolding. The treatment of recursion in pGCL is well developed, and we have incorporated some of this work, specifically regarding loops. Our treatment is at an early stage of development, but we already provide several useful rules, including this, which is a specialisation of lemma 7.3.1 of McIver & Morgan [11], and gives the correctness condition for standard post-expectations on loops which terminate[6] with probability 1:

$$\text{WP\_LOOP}\quad \frac{\text{well\_def } (\text{wp } body) \quad \text{sub\_distrib } (\textbf{do } G \to body) \quad (\lambda s.\ \text{«}G\text{»}\ s \times \text{«}I\text{»}\ s) \Vdash \text{wp } body\ \text{«}I\text{»}}{\text{«}I\text{»} \,\&\&\, \text{wp } (\textbf{do } G \to body)\ (\lambda s.1) \Vdash \text{wp } (\textbf{do } G \to body)\ (\lambda s.\ \text{«}\overline{G}\text{»} \times \text{«}I\text{»})} \tag{3}$$

4 Implementation and Extensions

We now expand on some details of the implementation, and its extensions.

4.1 Implementing wp and wlp

Figure 4 details the implementation of several primitives, together with the definitions of wp and wlp. Programs are represented as their associated transformer, parameterised by the treatment of Abort (the

[6] A strictly weaker condition than terminating along all paths: Non-terminating traces with probability 0 are acceptable. Consider flipping a coin until it shows heads: The only non-terminating path (flipping tails forever) has probability 0.

type_synonym (σ,α) nondet_monad $= \sigma \Rightarrow (\alpha \times \sigma)$ set $\times$ bool
$\{P\}f\{Q\} \equiv \forall s.\ P\ s \longrightarrow (\forall (r,s') \in \text{fst}\ (f\ s).\ Q\ r\ s')$
no_fail $P\ m \equiv \forall s.\ P\ s \longrightarrow \neg(\text{snd}\ (m\ s))$

Exec :: (σ,α) nondet_monad $\Rightarrow \sigma$ prog
Exec $M \equiv \lambda ab\ R\ s.$
  let $(SA, f) = M\ s$ in — *Run the monad*
    if f then Abort $ab\ R\ s$ — *Fail is Abort*
      else if $SA = \{\}$ then (bound_of R) — *Stuck is Success*
        else let $S = \text{snd`}\ SA$ in — *Ignore result*
          glb $(R\text{`}\ S)$ — *Infimum over states*

Figure 5: The L4.verified non-deterministic monad in Isabelle

parameter *ab*), giving either strict or liberal semantics. Only Abort and μ change their behaviour between wp and wlp: The former gives either failure ($\lambda P\,s.\ 0$) or success ($\lambda P\,s.$ bound_of P), whereas the latter is the least or the greatest fixed point, respectively. All others, as for (;;), and simply pass *ab* inward.

4.2 Consequences of a Shallow Embedding

The very shallow embedding used has two important consequences, the first of which is negative. The healthiness of transformers, and soundness of expectations, must be explicitly carried as assumptions. A deeper embedding [8] might restrict to the type of healthy transformers, in which case these would be satisfied by the type axioms.

We avoid such an embedding to reuse as much of the mechanisation within Isabelle/HOL as possible. Reasoning within a fresh type requires lifting (and modifying) all necessary rules. The burden of discharging our side-conditions is, moreover, not high. For any primitively constructed program, healthiness follows by invoking the simplifier with the appropriate lemmas.

The positive consequence of an extremely shallow embedding is the ease with which it can be extended. We have already seen an example: the definition of demonic choice from a set (following a standard abbreviation [11]). To do so, one need only supply weakest-precondition (and weakest-liberal-precondition) rules, rules to infer healthiness and (optionally) rules for proof decomposition.

Interestingly, it is not necessary to show that the new primitive is sound, that is, produces a healthy transformer for all inputs. It is merely necessary that the supplied rules show healthiness for just those cases in which it is actually used. Set demonic choice is just such a partially sound primitive: Healthiness does not generally hold for infinite sets. Applied here to finite sets, the given rules establish healthiness.

As a further example, we embed the non-deterministic monad at the heart of the L4.verified proof. The existing definition, Figure 5, is as a function from states (σ) to a set of result (α), state pairs. The extra result is the failure flag, used to explicitly signal failure. This was added to ensure that termination is preserved under refinement, as described elsewhere [4]. We embed as follows: Stuck (no successor states) is success, for compatibility with our infimum-over-alternatives interpretation; Explicit failure is

Abort, which in turn is either success or failure under wlp or wp, respectively. We lift results as follows:

$$\textsc{wp\_Exec}\ \frac{\{P\}\ prog\ \{\lambda r\ s.\ Q\ s\} \quad \text{no\_fail}\ P\ prog \quad \exists s.\ P\ s}{\text{«}P\text{»} \vdash \text{wp}\ prog\ \text{«}Q\text{»}} \qquad \textsc{wlp\_Exec}\ \frac{\{P\}\ prog\ \{\lambda r\ s.\ Q\ s\} \quad \exists s.\ P\ s}{\text{«}P\text{»} \vdash \text{wlp}\ prog\ \text{«}Q\text{»}}$$

4.3 Fixed Points and the Lattice Structure of Expectations and Transformers

Handling recursion means reasoning about fixed points. In this case, we need both least and greatest, on expectations and on transformers. Due to the shallow embedding, we cannot appeal to the existing fixed point results, which are phrased on a complete lattice. Neither the underlying type for expectations ($S \to \mathbb{R}$) or for transformers ($(S \to \mathbb{R}) \to S \to \mathbb{R}$) can be so instantiated, due to the lack of both top and bottom elements. The solution in each case is different.

Sound expectations have an obvious bottom element, $\lambda s.\ 0$, but there is no universal upper bound. We only require that there exists a bound for any given expectation. There need not exist any bound on an arbitrary set of sound expectations. For example, with $S = \mathbb{N}$, consider the set

$$\{(\lambda s.\ \text{if } s = n \text{ then } n \text{ else } 0) : n \in \mathbb{N}\}\ .$$

Each expectation is bounded (by n) and non-negative, yet the least upper bound, $\lambda s.\ s$, is unbounded.

We need a surrogate for the top element. To illustrate, take our definition for greatest fixed point:

$$\text{gfp\_in}\ f\ S \equiv \text{if } \exists x \in S.\ x \leq f\ x \text{ then lub } \{x \in S.\ x \leq f\ x\} \text{ else lowerbound } S$$

The lower bound is easy ($\lambda s.\ 0$), but in order to find the *least* upper bound, we first need *some* upper bound. In a complete lattice, this is the top element. Instead, we appeal to feasibility:

$$\text{bound\_of}\ ((\mu x.\ f\ x)\ P) \leq \text{bound\_of}\ P\ , \quad \text{and thus} \quad (\mu x.\ f\ x)\ P \leq \lambda s.\ \text{bound\_of}\ P\ .$$

It is therefore sufficient to consider fixed points that are *weakly bounded* by P:

$$\text{weakly\_bounded\_by}\ P \equiv \{Q.\ \text{sound}\ Q \wedge \text{bound\_of}\ Q \leq \text{bound\_of}\ P\}$$

Finally, we establish the standard fixed-point results parameterised by P. For example:

$$\textsc{gfp\_in\_unfold}\ \frac{\text{healthy}\ t}{\text{gfp\_in}\ t\ (\text{weakly\_bounded\_by}\ P) = t\ (\text{gfp\_in}\ t\ (\text{weakly\_bounded\_by}\ P))}$$

The case of transformers is simpler, again due to feasibility:

$$t\ P\ s \leq \text{bound\_of}\ P \quad \text{and thus} \quad t \leq \lambda P\ s.\ \text{bound\_of}\ P$$

Thus we have a top element, and establish a complete lattice by means of a quotient:

$$\text{le\_trans}\ t\ u \equiv \forall P.\ \text{sound}\ P \to t\ P \leq u\ P \qquad \text{equiv\_trans}\ t\ u \equiv \text{le\_trans}\ t\ u \wedge \text{le\_trans}\ u\ t$$

$$\text{htrans\_rel}\ t\ u \equiv \text{healthy}\ t \wedge \text{healthy}\ u \wedge \text{equiv\_trans}\ t\ u$$

$$\textbf{quotient\_type}\ \sigma\ \text{trans} = (\sigma \to \mathbb{R}) \to \sigma \to \mathbb{R}\ /\ \text{partial} : \text{htrans\_rel}$$

Using the induced homomorphism, we draw back the standard results:

$$\textsc{gfp\_trans\_unfold}\ \frac{\wedge t.\ \text{healthy}\ t \vdash \text{healthy}\ (T\ t) \quad \wedge t\ u.\ [\text{healthy}\ t; \text{healthy}\ u; \text{le\_trans}\ t\ u] \vdash \text{le\_trans}\ (T\ t)\ (T\ u)}{\text{equiv\_trans}\ (\text{gfp\_trans}\ T)\ (T\ (\text{gfp\_trans}\ T))}$$

4.4 The Verification Condition Generator

The VCG tactic, `pvcg`, is simple but nonetheless capable of handling Figure 3. It alternates applying an entailment rule, and attempting to discharge side-goals using internal and user-supplied rules.

The user supplies specifications as proved entailment rules, tagged with `[wp]`, and healthiness rules, tagged with `[health]`. Internal rules are tagged `[wp_core]`. The VCG selects rules as follows:

1. User-supplied rules, as written.
2. User-supplied rules, with a strengthened postcondition: `wp_strengthen_post[OF rule]`. This leaves a side-goal that the postcondition of the supplied rule entails the strengthened version.
3. Internal rules.

A user-supplied rule will override an internal rule, and may refer directly to a compound structure e.g. $P \Vdash \mathrm{wp}\ (a\,;;b)\ Q$. The given rule will be used rather than unfolding the composition. If no user rule is found, the VCG will proceed using its internal rules, calculating the exact weakest precondition by unfolding.

5 Related Work

The mechanisations of many existing programming logic formalisations have been presented, in Isabelle and in other theorem provers [4, 7, 12, 13]. Relative to these, the novelty of this work is in the treatment of probabilistic programs and properties, through pGCL.

A previous formalisation of pGCL, in the HOL4 theorem prover, was presented by Hurd et. al. [8], whence we have adapted our worked example (Monty Hall). Celiku & McIver [2], working from this same formalisation, developed an approach to analysing the execution time of probabilistic algorithms, presenting a mechanised proof of the self-stabilisation time for Herman's ring. We have presented a novel approach to formalising the same underlying logic. By prioritising the reuse of existing theories in Isabelle/HOL [14], in particular by defining expectations using the standard real type by means of our quotient and pullback technique, we achieve excellent automation and integration with existing work, notably L4.verified [9].

6 Applications & Ongoing Work

This formalisation was developed to assist in the verification of high-assurance component systems. Our principal application is the verification of probabilistic security properties, of the sort established in Section 2. Having established that a guessing-attack based measure is refinement-sound, we plan to formally establish bounds on vulnerability due to guessing attacks involving side-channel leakage.

The fact that we can embed the nondeterministic monad used in the L4.verified proof is significant, as it means that we can appeal directly to that project's top-level theorem in establishing a probabilistic result on seL4 (the microkernel in question). An attractive target is a randomised scheduler: In seL4, as in many kernels, the scheduler is a small component, called at the top level after all other state-modifying code has executed. We could thus appeal to the existing, unmodified, proof to establish the soundness of the transformation and then, lifting the result via the above construction, establish a probabilistic result on the system including the scheduler. As randomised scheduling is a common approach to side-channel mitigation, this is a promising approach to achieving our stated goal of formally establishing side-channel bounds in real systems.

References

[1] Gertrud Bauer & Markus Wenzel (2001): *Calculational Reasoning Revisited – An Isabelle/Isar experience*. In: *14th TPHOLs*, Springer-Verlag, pp. 75–90, doi:10.1007/3-540-44755-5_7.

[2] Orieta Celiku & Annabelle McIver (2004): *Cost-based analysis of probabilistic programs mechanised in HOL*. *Nordic J. of Computing* 11(2), pp. 102–128. Available at http://www.cse.unsw.edu.au/~carrollm/probs/Papers/Celiku-04.pdf.

[3] David Cock (2011): *Exploitation as an inference problem*. In: *4th AISEC*, Chicago, IL, USA, pp. 105–106, doi:10.1145/2046684.2046702.

[4] David Cock, Gerwin Klein & Thomas Sewell (2008): *Secure Microkernels, State Monads and Scalable Refinement*. In Otmane Ait Mohamed, César Muñoz & Sofiène Tahar, editors: *21st TPHOLs*, *LNCS* 5170, Springer-Verlag, Montreal, Canada, pp. 167–182, doi:10.1007/978-3-540-71067-7_16.

[5] Edsger W. Dijkstra (1975): *Guarded commands, nondeterminacy and formal derivation of programs*. *CACM* 18(8), pp. 453–457, doi:10.1145/360933.360975.

[6] Joseph Goguen & José Meseguer (1982): *Security Policies and Security Models*. In: *IEEE Symp. Security & Privacy*, Comp. Soc., Oakland, California, USA, pp. 11–20.

[7] William L. Harrison & Richard B. Kieburtz (2005): *The logic of demand in Haskell*. *J. Functional Progr.* 15(6), pp. 837–891, doi:10.1017/S0956796805005666.

[8] Joe Hurd, Annabelle McIver & Carroll Morgan (2005): *Probabilistic guarded commands mechanized in HOL*. *Theoretical Computer Science* 346(1), pp. 96 – 112, doi:10.1016/j.tcs.2005.08.005. Available at http://www.sciencedirect.com/science/article/pii/S0304397505004767.

[9] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch & Simon Winwood (2009): *seL4: Formal Verification of an OS Kernel*. In: *22nd SOSP*, ACM, Big Sky, MT, USA, pp. 207–220, doi:10.1145/1629575.1629596.

[10] Annabelle McIver & Carroll Morgan (2001): *Partial correctness for probabilistic demonic programs*. *Theoretical Comp. Sci.* 266(1–2), pp. 513 – 541, doi:10.1016/S0304-3975(00)00208-5. Available at http://www.sciencedirect.com/science/article/pii/S0304397500002085.

[11] Annabelle McIver & Carroll Morgan (2004): *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer.

[12] Till Mossakowski, Lutz Schröder & Sergey Goncharov (2010): *A generic complete dynamic logic for reasoning about purity and effects*. *Formal Aspects Comput.* 22(3-4), pp. 363–384, doi:10.1007/s00165-010-0153-4.

[13] Tobias Nipkow (2002): *Hoare Logics in Isabelle/HOL*. In H. Schwichtenberg & R. Steinbrüggen, editors: *Proof and System-Reliability*, Kluwer, pp. 341–367.

[14] Tobias Nipkow, Lawrence Paulson & Markus Wenzel (2002): *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. *LNCS* 2283, Springer-Verlag.

[15] Steve Selvin (1975): *A problem in probability (letter to the editor)*. *American Statistician* 29(1), p. 67.