Automatic Function Annotations for Hoare Logic

Daniel Matichuk
NICTA
Sydney, Australia
daniel.matichuk@nicta.com.au

In systems verification we are often concerned with multiple, inter-dependent properties that a program must satisfy. To prove that a program satisfies a given property, the correctness of intermediate states of the program must be characterized. However, this intermediate reasoning is not always phrased such that it can be easily re-used in the proofs of subsequent properties. We introduce a function annotation logic that extends Hoare logic in two important ways: (1) when proving that a function satisfies a Hoare triple, intermediate reasoning is automatically stored as function annotations, and (2) these function annotations can be exploited in future Hoare logic proofs. This reduces duplication of reasoning between the proofs of different properties, whilst serving as a drop-in replacement for traditional Hoare logic to avoid the costly process of proof refactoring. We explain how this was implemented in Isabelle/HOL and applied to an experimental branch of the seL4 microkernel to significantly reduce the size and complexity of existing proofs.

1 Introduction

It is not always apparent what properties need to be proven when formally verifying real software. Clearly a program should maintain the consistency of its data structures, but it may be less obvious to show that, for example, a scheduling policy is fair. As a result, a verified system will inevitably be reasoned about multiple times while considering different properties. This will involve characterizing the correctness of intermediate states of a program with respect to each of these properties. Likely these properties will have some inter-dependencies, e.g. we can only reason that a scheduling policy is fair if the scheduler queues are valid. In these cases, the proofs of these properties will depend on results previously shown. This inter-dependence of reasoning demonstrates a necessity to structure theories and lemmas in such a way that maximizes their re-use.

The seL4 microkernel is to our knowledge the most extensive code-level verification ever performed of a general-purpose operating system kernel. The C implementation is shown to formally *refine* an abstract specification of its behaviour [4], which is proven to preserve a set of kernel invariants. These invariants describe the correctness of all kernel structures, saying, for example, that distinct objects in memory do not overlap. In the subsequent proofs of security properties, namely integrity and authority confinement [7], the existing lemmas shown during the invariant proofs were heavily used.

The proofs of these properties were done using a monadic variant of Hoare logic [1] , which is used to reason about pre and post conditions of functions. When an invariant is temporarily violated, such as during the creation of new kernel objects, care must be taken to describe this intermediate state, so it can be reasoned that further operations re-establish this invariant. In traditional Hoare logic, correctness of this intermediate state cannot easily be expressed. If we wish to re-establish this temporary invariant violation in subsequent proofs, we will be forced to perform the same reasoning required in the original proof. With powerful vcgs (verification condition generators) and automated reasoning inside a theorem prover, the impact of this duplicated reasoning can be minimal. However, for a sufficiently complex function with corresponding complex intermediate states, significant manual effort is required to prove any

Submitted to:
SSV 2012

```
new-tcb p ≡ do
  i ← alloc;
  tcb ← create-tcb p;
  init-tcb tcb i;
  enqueue-tcb i p;
  return i
od

alloc ≡ do
  ids ← gets ids;
  i ← select ids;
  set-ids (ids − {i});
  return i
od

create-tcb p ≡ do
  tcb ← return empty-tcb;
  return (tcb(|priority := p|))
od

init-tcb tcb i ≡ do
  tcbs ← gets tcbs;
  set-tcbs (tcbs(i ↦ tcb))
od

enqueue-tcb i p ≡ do
  qs ← gets queues;
  q ← return (qs p);
  set-queues (qs(p := i·q))
od
```

Figure 1: A specification for an example tcb allocation function.

given property. Re-establishing the correctness of these intermediate states can then result in significant portions of duplicated proof, which become difficult and costly to maintain as a project evolves.

In this paper, we present a function annotation logic, which allows the correctness of these intermediate states to be shown as function annotations. Properties proven about these intermediate states can then be used in future proofs without any redundant reasoning. These annotations do not need to be explicitly defined, but can be generated as a consequence of an existing Hoare logic proof.

Function annotations are not novel, in [3] Hoare advocates proving the correctness of assertions so they may be assumed in further proofs. Traditionally, however, pre and post conditions are favoured over manually annotating entire functions. In existing verification frameworks, such as VCC [2], annotations are defined alongside the code and a function receives a single set of annotations. The primary contribution of this paper is the automatic extraction of annotations from existing Hoare logic proofs. Additionally, multiple sets of annotations can be defined for orthogonal properties about a function.

2 An example specification

To illustrate function annotations, we introduce a monadic specification for a simple function that might be used in an operating system. An imperative program may be specified as a nondeterministic state monad [1] by defining a record containing a field for each global variable in the program, in addition to relevant pieces of global state. The program is then formalized as a function that takes a state record as an input and yields an updated record, representing global modifications, and a return value. Nondeterministic computations yield a set of return values and state pairs, indicating all possible ways the function could resolve its nondeterminism. We use "do-notation" to phrase these monadic specifications in an imperative style.

Shown in Figure 1 new-tcb creates and enqueues a new tcb (thread control block) kernel object. First, in alloc, an identifier is allocated by selecting one out of the free pool ids, removing it from the pool and then returning. Next, create-tcb creates a tcb with the appropriate priority. init-tcb then associates this new tcb with the previously allocated identifier in the tcbs partial map. Finally enqueue-tcb adds the identifier to the head of the appropriate priority queue in queues. Hence there are 3 pieces of state that are relevant to the behaviour of new-tcb: ids, tcbs and queues. These are represented as fields in a state record, where gets x returns the x field of the state and set-x sets that field. return simply returns the

$$\begin{array}{ll} \mathsf{valid\text{-}id}\ i\ s & \equiv\ i \in \mathsf{ids}\ s \longleftrightarrow i \notin \mathsf{dom}\ (\mathsf{tcbs}\ s) \\ \mathsf{valid\text{-}free}\ s & \equiv\ \forall id.\ \mathsf{valid\text{-}id}\ id\ s \\ \mathsf{valid\text{-}free\text{-}except}\ i\ s & \equiv\ \forall id.\ (id \neq i \longrightarrow \mathsf{valid\text{-}id}\ id\ s) \wedge i \notin \mathsf{dom}\ (\mathsf{tcbs}\ s) \wedge i \notin \mathsf{ids}\ s \end{array}$$

Figure 2: Describing a valid set of free identifiers.

given expression, leaving the state unmodified. Given a set S, select S nondeterministically selects a value from S.

To verify this function, we may wish to first reason that new-tcb preserves the validity of the pool of free identifiers ids with respect to tcbs. We introduce an invariant valid-free, defined in Figure 2, which states that an identifier is free iff it is not in the domain of tcbs. In other words, no element of ids points to a tcb and all free identifiers are necessarily in ids.

We describe the preservation of this invariant as the following Hoare triple:

$$\{\!|\mathsf{valid\text{-}free}|\!\}\ \mathsf{new\text{-}tcb}\ p\ \{\!|\lambda\text{-}.\ \mathsf{valid\text{-}free}|\!\} \tag{1}$$

It states that if, before new-tcb runs, the pool of identifiers is valid then it remains valid afterwards. Here λ-. binds the return value in the post condition to a dummy variable, effectively ignoring it. To prove this triple, it is sufficient to show that the operations of new-tcb satisfy a collection of Hoare triples that can be composed together.

$$\begin{array}{c} \{\!|\mathsf{valid\text{-}free}|\!\}\ \mathsf{alloc}\ \{\!|\lambda i\ s.\ \mathsf{valid\text{-}free\text{-}except}\ i\ s|\!\} \\ \{\!|\mathsf{valid\text{-}free\text{-}except}\ i|\!\}\ \mathsf{create\text{-}tcb}\ p\ \{\!|\lambda\text{-}.\ \mathsf{valid\text{-}free\text{-}except}\ i|\!\} \\ \{\!|\mathsf{valid\text{-}free\text{-}except}\ i|\!\}\ \mathsf{init\text{-}tcb}\ obj\ i\ \{\!|\lambda\text{-}.\ \mathsf{valid\text{-}free}|\!\} \\ \{\!|\mathsf{valid\text{-}free}|\!\}\ \mathsf{enqueue\text{-}tcb}\ i\ p\ \{\!|\lambda\text{-}.\ \mathsf{valid\text{-}free}|\!\} \end{array} \tag{2}$$

This demonstrates a temporary violation of valid-free between the allocation and initialization of an identifier i, which is characterized by valid-free-except i.

In this small example it's clear that re-establishing valid-free-except i just prior to init-tcb in a future proof would be a trivial application of existing Hoare triples. In the context of real software verification, where preconditions become large collections of properties, re-establishing the correctness of these intermediate states can result in large pieces of duplicated proof. Moreover, for large functions it is not practical to write an individual lemma establishing these predicates for every point in the function. Additionally there is no established mechanism in Hoare logic for phrasing such a lemma without manually adding assertions to the program text. In the next section we will demonstrate how we can generate a function annotation from this proof which establishes these intermediate properties, making them available for re-use during later proofs over the same function.

Monadic Hoare Logic

Formally, a Hoare triple over a nondeterministic state monad is defined as follows:

$$\{\!|P|\!\}\ f\ \{\!|Q|\!\} \equiv \forall s.\ P\ s \longrightarrow (\forall (r, s') \in f\ s.\ Q\ r\ s') \tag{3}$$

This states that a function f satisfies a Hoare triple if, from all states s satisfying $P\ s$, we have that all possible computation paths of f satisfy $Q\ r\ s'$, where r is the return value of f.

To combine the results in (2) we use the *split rule*, which states that the precondition of a function can be used as the postcondition of the previous function.

```
new-tcb-valid-free-ann p ≡ doA
  i ← {|valid-free|} alloc;
  tcb ← {|valid-free-except i|} create-tcb p;
  {|valid-free-except i|} init-tcb tcb i;
  {|valid-free|} enqueue-tcb i p;
  {|valid-free|} return i
odA
```

Figure 3: An annotation of new-tcb created during the proof of valid-free.

$$\frac{\forall x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\} \qquad \{\!|A|\!\}\ f\ \{\!|B|\!\}}{\{\!|A|\!\}\ \mathsf{do}\quad x \leftarrow f;\quad g\ x\ \mathsf{od}\ \{\!|C|\!\}}\ \text{WP-SPLIT}$$

Where g is quantified over possible return values of f.

3 Using Annotations

The function annotation framework is designed to be effectively transparent with respect to Hoare logic, which enables the re-use of these intermediate properties while requiring minimal changes to existing proofs. The goal annotation new-tcb-valid-free-ann is shown in Figure 3. The first step is to re-phrase the top level Hoare triple as an *annotator*:

$$\{\!|\text{valid-free}|\!\}\ \text{new-tcb}\ p\ \{\!|\lambda\text{-. valid-free}|\!\}\ \langle\text{new-tcb-valid-free-ann}\ p\rangle \tag{4}$$

Which states that, in addition to satisfying (1), new-tcb adheres to the annotations given in new-tcb-valid-free-ann given valid-free as a precondition. The additional proof obligations, showing that these annotations are satisfied, are trivially shown as a result of the existing proof. An existing proof of (1) can therefore be modified to instead show this result by mechanically exchanging Hoare logic rules for analogous annotator rules. In Section 4 we will see how these annotations can be automatically created by Isabelle, rather than having to be explicitly specified.

The annotation itself is a monad which tracks an additional piece of state: a boolean which indicates annotation failure. The annotation is checked at every step of the computation, but does not affect the behaviour of the underlying function. This annotation can therefore be reasoned against as if it were the function it annotates. Additionally, assuming some precondition, if we can show that no annotation will fail then we may assume the properties stated in the annotation.

To reason about an annotation we introduce an *annotated triple*:

$$\|P\|\ F\ \|Q\| \equiv \forall s.\ \neg\ \mathsf{afails}\ (F\ s) \longrightarrow P\ s \longrightarrow (\forall (r, s') \in \mathsf{dropA}\ F\ s.\ Q\ r\ s')$$

Here, F is an annotation like new-tcb-valid-free-ann and P and Q are pre and post conditions respectively. dropA F is the underlying function that F annotates and afails $(F\ s)$ is true whenever F has an annotation that is not satisfied when proceeding from s. This is similar to a standard Hoare triple, with the exception that it may take non-failure of the annotation for granted. Once an annotation is shown to be satisfied (using an annotator as in (4)) we can carry a result from an annotated triple over it to the underlying function. Proving an annotated triple over an annotation is necessarily more straightforward than a standard Hoare triple, because all the individual assertions about the intermediate states may now be assumed to hold.

To see how annotations are used, we introduce another invariant valid-queues shown in Figure 4. It states that, for every scheduler queue, all the identifiers in that queue correspond to a tcb with the

$$\begin{array}{l}\mathsf{valid\text{-}queues}\ s \quad \equiv\ \forall p.\ \forall i{\in}\mathsf{queues}\ s\ p.\ \mathsf{tcb\text{-}at\text{-}prio}\ i\ p\ s \\ \mathsf{tcb\text{-}at\text{-}prio}\ i\ p\ s\ \equiv\ i \in \mathsf{dom}\ (\mathsf{tcbs}\ s) \wedge \mathsf{priority}\ (\mathsf{the}\ (\mathsf{tcbs}\ s\ i)) = p\end{array}$$

Figure 4: Describing the correctness of scheduler queues.

appropriate priority. We can see that we will require valid-free as a precondition if new-tcb is to preserve valid-queues, otherwise alloc could select an identifier that is already enqueued. We can therefore make use of the annotation new-tcb-valid-free-ann shown previously. The goal Hoare triple is as follows:

$$\{\!|\mathsf{valid\text{-}queues}\ \textit{and}\ \mathsf{valid\text{-}free}|\!\}\ \mathsf{new\text{-}tcb}\ p\ \{\!|\lambda\text{-}.\ \mathsf{valid\text{-}queues}|\!\}$$

However, as a result of (4) which shows that new-tcb satisfies annotations related to valid-free, we can instead prove this annotated triple: [1]

$$\|\mathsf{valid\text{-}queues}\ \textit{and}\ \mathsf{valid\text{-}free}\|\ \mathsf{new\text{-}tcb\text{-}valid\text{-}free\text{-}ann}\ p\ \|\lambda\text{-}.\ \mathsf{valid\text{-}queues}\|$$

We prove these sorts of annotated triples by decomposing them with an analogous rule to WP-SPLIT and then converting them into ordinary Hoare triples. The annotation on a function $\{\!|P|\!\}\ f$ can be read as "f, given P". The following rule allows such an annotation in an annotated triple to be assumed as a precondition, while converting into a standard Hoare triple:

$$\frac{\{\!|R\ \textit{and}\ P|\!\}\ f\ \{\!|Q|\!\}}{\|R\|\ \{\!|P|\!\}\ f\ \|Q\|} \tag{5}$$

This allows a standard Hoare triple to be applied as if that precondition was established in a traditional Hoare logic proof.

A strong splitting rule, which combines the annotated triple splitting rule and (5), can be directly applied to decompose an annotated triple into a collection of standard Hoare triples that are strengthened with annotations.

$$\frac{\forall x.\ \|B\ x\|\ G\ x\ \|C\| \qquad \{\!|A\ \textit{and}\ P|\!\}\ f\ \{\!|B|\!\}}{\|A\|\ \mathsf{doA}\quad x \leftarrow \{\!|P|\!\}\ f;\quad G\ x\ \mathsf{odA}\ \|C\|}\ \text{WP-STRONG-SPLIT}$$

To show that new-tcb preserves valid-queues we first establish the precondition for enqueue-tcb to preserve valid-queues:

$$\{\!|\mathsf{valid\text{-}queues}\ \textit{and}\ \mathsf{tcb\text{-}at\text{-}prio}\ i\ p|\!\}\ \mathsf{enqueue\text{-}tcb}\ i\ p\ \{\!|\lambda\text{-}.\ \mathsf{valid\text{-}queues}|\!\}$$

We can then demonstrate that init-tcb preserves valid-queues assuming the identifier being initialized is not already enqueued:

$$\{\!|\mathsf{valid\text{-}queues}\ \textit{and}\ \mathsf{not\text{-}queued}\ i|\!\}\ \mathsf{init\text{-}tcb}\ \textit{obj}\ i\ \{\!|\lambda\text{-}.\ \mathsf{valid\text{-}queues}|\!\}$$

where not-queued $i\ s \equiv \forall p.\ i \notin$ queues $s\ p$. If applied directly, this will require additional reasoning that create-tcb preserves the fact that the identifier is not enqueued and that alloc selects an identifier that is not enqueued. Recall, however, that this proof is over an annotated function, and at this point we have that valid-free-except i holds. WP-STRONG-SPLIT therefore will have added valid-free-except i to the precondition for create-tcb. Using the implication valid-queues $s \wedge$ valid-free-except $i\ s \longrightarrow$ not-queued $i\ s$ we may strengthen the precondition to instead assume valid-free-except i:

$$\{\!|\mathsf{valid\text{-}queues}\ \textit{and}\ \mathsf{valid\text{-}free\text{-}except}\ i|\!\}\ \mathsf{init\text{-}tcb}\ \textit{tcb}\ i\ \{\!|\lambda\text{-}.\ \mathsf{valid\text{-}queues}|\!\}$$

This uses the annotation granted from WP-STRONG-SPLIT, therefore the only precondition that needs to be propagated is valid-queues, and we no longer have to reason about i not being enqueued. In a larger function this could potentially avoid propagating this precondition up through several operations and duplicating a significant amount of existing reasoning.

[1] This is formally justified by the rule (6) introduced later in Section 4

```
new-tcb-valid-queues-ann p ≡ doA
  i ← {|valid-queues|} alloc;
  tcb ← {|valid-queues|} create-tcb p;
  {|valid-queues and not-queued i and (λ-. priority tcb = p)|}
  init-tcb tcb i;
  {|valid-queues and tcb-at-prio i p|} enqueue-tcb i p;
  {|valid-queues|} return i
odA
```

Figure 5: Annotations for new-tcb from the proof of valid-queues.

```
new-tcb-valid-free-ann p ⋈ new-tcb-valid-queues-ann p = doA
  i ← {|valid-free and valid-queues|} alloc;
  tcb ← {|valid-free-except i and valid-queues|} create-tcb p;
  {|valid-free-except i and valid-queues and not-queued i and
    (λ-. priority tcb = p)|}
  init-tcb tcb i;
  {|valid-free and valid-queues and tcb-at-prio i p|} enqueue-tcb i p;
  {|valid-free and valid-queues|} return i
odA
```

Figure 6: A combination of two annotations for new-tcb.

Similar to the proof for valid-free we can create annotations regarding valid-queues as shown in Figure 5. Note that only properties specifically related to valid-queues are used as annotations. To combine two annotated functions we define a merge operation $\bowtie$, which produces a new annotated function that is simply the conjunction of both annotations, and can be used whenever both their preconditions are established. The result of merging both of the previous annotations for new-tcb is shown in Figure 6.

4 Creating Annotations

In this section we describe how function annotations were formalized using Isabelle and how they were incorporated into an existing Hoare logic vcg to allow them to be seamlessly integrated with existing proofs. Additionally we show how function annotations can be easily generated by Isabelle and exported by using schematic lemmas.

Function annotations are implemented by creating a new function which, effectively, has an assertion between every operation. To demonstrate that an annotation is valid (i.e. create an annotator like (4)) it is sufficient to show that none of these assertions will fail under a given precondition. An extra flag is tracked, in addition to the global state record, which indicates failure of an annotation. An annotation, therefore, simply evaluates the function as normal but also tests annotations.

$$\{|P|\}\, f = (\lambda s.\ (f\, s, \neg\, P\, s))$$

To evaluate the composition of two annotated functions we compose their inner functions and then set the failure flag if any possible nondeterministic branching of the composition can fail. The definition is given in Figure 7 with relevant lemmas to characterize dropA and afails. can-fail-from $F\ G\ s$ runs G against all possible results of F from s and then tests annotation failure on all possible outcomes.

A function is said to satisfy an annotation under some precondition if, by asserting that precondition

at the beginning of the function, no assertions will fail. To formalize this, we define a partial ordering $\sqsubseteq$ on annotated functions that can be described by the strength of the annotations:

$$F \sqsubseteq G \equiv \forall s.\ (\neg\ \mathsf{afails}\ (F\ s) \longrightarrow \mathsf{dropA}\ F\ s = \mathsf{dropA}\ G\ s) \wedge (\mathsf{afails}\ (G\ s) \longrightarrow \mathsf{afails}\ (F\ s))$$

which states that the annotation F is stronger than G if, under non-failure, they annotate the same function and failure of G always implies failure of F. It is best characterized by the following rule:

$$\frac{\forall s.\ P\ s \longrightarrow Q\ s}{\{\!|P|\!\}\ f \sqsubseteq \{\!|Q|\!\}\ f}$$

Hence f satisfies some annotation F under P if $\{\!|P|\!\}\ f \sqsubseteq F$. When a function satisfies an annotation, one can reason about the function by reasoning about the annotation via an annotated triple, which only evaluates under non-failure and thus all annotations may be assumed.

$$\frac{\{\!|P|\!\}\ f \sqsubseteq F \qquad \|P\|\ F\ \|Q\|}{\{\!|P|\!\}\ f\ \{\!|Q|\!\}} \tag{6}$$

To prove adherence to an annotation we use an annotator. It simply states that, in addition to satisfying the given Hoare triple, the function adheres to the given annotation.

$$\{\!|P|\!\}\ f\ \{\!|Q|\!\}\ \langle F \rangle \equiv \{\!|P|\!\}\ f\ \{\!|Q|\!\} \wedge \{\!|P|\!\}\ f \sqsubseteq F \tag{7}$$

There are two motivations for proving annotation adherence this way. As will be explained in the next section, this approach allows the annotation to be "collected" by Isabelle rather than provided explicitly by the user. Additionally it provides a clear notion of an *input* function that is being proved against and an *output* annotation that is being produced/satisfied. In the general case the function may itself already have annotations; by phrasing the triple in this way we can distinguish annotations we may use during the proof of the Hoare triple and annotations that we are producing/satisfying. To illustrate this, recall the proof that new-tcb preserves valid-queues. While using the annotation created during the proof of valid-free we also want to produce an annotation for valid-queues. This can be phrased with a strong annotator, which has an analogous definition to the one given in (7):

$$\|\mathsf{valid\text{-}queues}\|\ \mathsf{new\text{-}tcb\text{-}valid\text{-}free\text{-}ann}\ p\ \|\lambda\text{-}.\ \mathsf{valid\text{-}queues}\|\ \langle \mathsf{new\text{-}tcb\text{-}valid\text{-}queues\text{-}ann}\ p \rangle$$

Once this proof is complete, we now have that new-tcb, in addition to satisfying new-tcb-valid-free-ann under valid-free, satisfies new-tcb-valid-queues-ann under valid-queues *and* valid-free. To merge two annotations, as seen previously in Figure 6, we simply take the disjunction of their failure flags:

$$\begin{array}{lcl}
\mathsf{doA} & \equiv & \lambda s.\ ((\mathsf{do} \\
\quad x \leftarrow F; & & \qquad x \leftarrow \mathsf{dropA}\ F; \\
\quad G\ x & & \qquad \mathsf{dropA}\ (G\ x) \\
\mathsf{odA} & & \quad \mathsf{od}) \\
 & & \quad s, \\
 & & \quad \mathsf{can\text{-}fail\text{-}from}\ F\ G\ s \vee \mathsf{afails}\ (F\ s)) \\
\mathsf{can\text{-}fail\text{-}from}\ F\ G\ s & \equiv & \mathsf{True} \in \mathsf{afails}\ `\ (\lambda(x, y).\ G\ x\ y)\ `\ \mathsf{dropA}\ F\ s \\
\mathsf{dropA}\ \{\!|P|\!\}\ f & = & f \\
\mathsf{afails}\ (\{\!|P|\!\}\ f\ s) & = & \neg\ P\ s
\end{array}$$

Figure 7: Composing annotated functions.

$$\begin{array}{lcl} \mathsf{doA} & \bowtie & \mathsf{doA} \\ \quad x \leftarrow \{\!|P|\!\}\, f; & & \quad x \leftarrow \{\!|Q|\!\}\, f; \\ \quad G\, x & & \quad G'\, x \\ \mathsf{odA} & & \mathsf{odA} \end{array} \quad = \quad \begin{array}{l} \mathsf{doA} \\ \quad x \leftarrow \{\!|P \textit{ and } Q|\!\}\, f; \\ \quad G\, x \bowtie G'\, x \\ \mathsf{odA} \end{array}$$

Figure 8: Distributing an annotation merge across functions.

$$F \bowtie G \equiv \lambda s.\ (\mathsf{dropA}\ F\ s, \mathsf{afails}\ (F\ s) \vee \mathsf{afails}\ (G\ s))$$

Note that only one of the functions F is actually evaluated. This is simply because merging is only sensible between two annotations over the same function, and so $\mathsf{dropA}\ F\ s = \mathsf{dropA}\ G\ s$ is implicitly assumed. We show in Figure 8 that this merge operation distributes across function composition, so merging two annotations annotates the individual operations. This can be repeatedly applied to show the result in Figure 6. new-tcb can then be shown to satisfy this merged annotation using the following rule:

$$\frac{\{\!|P|\!\}\, f \sqsubseteq F \qquad \{\!|Q|\!\}\, f \sqsubseteq F'}{\{\!|P \textit{ and } Q|\!\}\, f \sqsubseteq F \bowtie F'}$$

This states that a function satisfies the merge of two annotations under the conjunction of their preconditions.

Automatic Annotations

Significant amounts of a Hoare logic proof can be automated by a vcg. At the core of any Hoare logic vcg is a collection of rules that decompose a Hoare triple into a collection of Hoare triples over the individual steps of a function. In Isabelle, the weakest precondition vcg *wp* processes a function bottom-up, computing the precondition necessary for the last operation to establish the postcondition. This computed precondition then becomes the postcondition for the next iteration, and so on. This is accomplished by repeated applications of the split rule WP-SPLIT.

To modify the vcg to work with annotators, we introduce an analogous split rule which additionally saves the computed precondition as an annotation of f.

$$\frac{\forall x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\}\ \langle G\ x\rangle \qquad \{\!|A|\!\}\ f\ \{\!|B|\!\}}{\{\!|A|\!\}\ \mathsf{do} \quad x \leftarrow f; \quad g\ x\ \mathsf{od}\ \{\!|C|\!\}\ \langle \mathsf{doA} \quad x \leftarrow \{\!|A|\!\}\, f; \quad G\ x\ \mathsf{odA}\rangle} \tag{8}$$

Note that, after this rule is applied, only a standard Hoare triple is required to be shown of f. Repeated applications of this rule, combined with some additional rules for control flow, result in a collection of Hoare triples as proof obligations. When solved, the preconditions for these Hoare triples are stored in the function annotations.

To avoid ever explicitly stating the definition of an annotation we can use Isabelle's *schematic lemma* feature, which allows terms in a lemma statement to be left unspecified and instantiated during the proof. The bind rule given in (8) will, after applied over the entire function, instantiate such a term to the computed function annotation.

For example, we rephrase an existing Hoare triple proof as an annotator, leaving the annotation as a schematic variable.

$$\{\!|\mathsf{even}\ i|\!\}\ \mathsf{double\text{-}plus}\ i\ \{\!|\lambda\text{-}.\ \mathsf{even}\ i|\!\}\ \langle ?L\rangle$$

After the proof is finished, the schematic has been instantiated to a function annotation.

$$\{\!|\mathsf{even}\ i|\!\}\ \mathsf{double\text{-}plus}\ i\ \{\!|\lambda\text{-}.\ \mathsf{even}\ i|\!\}\ \langle \mathsf{doA} \quad \{\!|\mathsf{even}\ i|\!\}\ i{+}{+}; \quad \{\!|\mathsf{odd}\ i|\!\}\ i{+}{+}\ \mathsf{odA}\rangle$$

Complex Annotations

Function annotations defined in this way are simply extensions of existing functions, which track an additional boolean across all possible nondeterministic branches. Due to this construction, they are necessarily as expressive as the underlying monadic formalization. In particular one can annotate a recursive function (creating a recursive annotation) or a map over an inductive datatype. This would serve to simplify induction hypotheses while reasoning about these functions as necessary correctness invariants could be assumed across all iterations of the function. These annotations could still be collected automatically from existing proofs, although the process would be slightly more involved. The use of function annotations in this context seems promising, but has not been fully explored and has been left for future work.

5 Case Study: seL4

Interesting properties cannot always be expressed in Hoare logic. In the ongoing proof of confidentiality for the seL4 microkernel [6], a proof calculus was used to formalize an upper bound on the information that a function can read. This involves reasoning about multiple executions of a function, which cannot be easily expressed in Hoare logic. Function annotations can still be used when proving such a property by explicitly turning annotations into assertions, effectively converting back into a standard function while retaining some information from the annotations. The standard confidentiality calculus [6] can then be applied and make use of the assertions.

The confidentiality proof for seL4 builds on previous invariant proofs in addition to proofs of integrity and authority confinement [7]. In general, for a function to be confidentiality-preserving it must be well-behaved, which requires reasoning about it in the presence of invariants. These invariants are then required as preconditions for confidentiality and we must therefore demonstrate that certain invariants are preserved at intermediate points of a function. In the majority of cases there are sufficient existing lemmas to easily re-play this reasoning, however some functions have behaviours which are difficult to characterize. Such a function is invoke-untyped, which changes the type of kernel objects in a region of memory. The proof that invoke-untyped preserves the invariants is 300 lines of Isabelle proof script. Some manual effort was involved to re-phrase the top-level precondition (approximately 5 lines of Isabelle) in order to produce a function annotation from this proof. Additionally, the generated annotation needed to be manually modified in order to remove extraneous properties. Ultimately a single line was added to the script with 40 lines of manual annotation modifications. When this annotation was applied to the proofs of integrity and authority confinement, the number of lines of proof script was reduced from 192 to 56. This produced another annotation which, when applied to the proof of confidentiality, reduced the lines of proof script from 215 to to 39. In this case it is clear that most of the logic in these proofs was simply re-proving the preservation of the invariants. Some manual effort is still required to effectively use these annotations, so they are currently in an experimental branch of the seL4 proof. There has been some development in adding more automation to this process, but it is still ongoing.

6 Related and Future Work

Similar to our use of nondeterministic state monads [1], Sprenger et al. [8] formalize Hoare logic over state monads in the verification of a security protocol. They similarly use a shallow embedding to exploit the significant amount of proof automation already present in Isabelle/HOL.

Swierstra [9] encodes the correctness of a specification in its type, creating a *strong specification*. The construction of the specification itself guarantees correctness, with respect to a given property. In our logic, the relationship between a function and its annotations is analogous to the relationship between a specification and its strong specification.

Function annotations are merely one approach to solving the general problem of reusing results about the intermediate states of a function. Alternatively, assertions could be explicitly provided as part of the specification, as done by Mossakowski et al. [5] in their formalization of Hoare logic on monads. Although this approach is more general, it introduces barriers to splitting up the same proof among multiple persons [1]. Additionally, extending these assertions to include further properties may incur significant proof maintenance overhead. In contrast, our formalization produces annotations as an orthogonal artifact to the specification itself.

Another approach would be to have intermediate states explicitly labelled and assertions made against those labels. This would enable the ad-hoc collection of facts about intermediate states, as is done by using function annotations. Rather than constructing several annotated versions of a function, the user would instead be creating an assertion cache for the intermediate states. Such an approach would allow for more dynamic use of assertions, as the user would not have to decide at the start of a proof which annotation sets he wished to use. The cost, however, comes in the initial overhead of defining these labels. The most straightforward approach would be to modify the monad formalization to label intermediate states, which would incur an initial overhead for any existing proofs. In the case of a large proof effort like seL4 this could potentially affect hundreds of thousands of lines of proof script. The advantage of function annotations is their low overhead, as they do not require any modification to the underlying formalization.

Function annotations as presented here have been shown to work well in practice, however some manual mechanical effort is still required to make use of them. In future work we plan to make function annotations more opaque to the end user. Ideally, rather than being explicitly exported through schematic lemmas, they could implicitly be generated as part of standard Hoare logic proofs. During further proofs over the same function, annotations could then implicitly be used by the vcg, or asserted if explicit reasoning needs to be done against them. Additionally, properties could be tagged as "annotation-worthy" and will otherwise not be pushed into automatic annotation generation. With these improvements, the user would not need to be aware that they are using function annotations, but could transparently exploit previously established reasoning with a collection of tactics and through the vcg.

7 Conclusion

In this paper we have presented a function annotation logic which can be used to prove properties about intermediate points of functions. It is designed to generate annotations from existing Hoare logic proofs with minimal modifications to these proofs, allowing the reasoning within those proofs to be easily re-used. Annotations can be used to prove additional, stronger annotations and unrelated annotations can be merged together as a conjunction of their individual assertions. They are useful when a function is too complex for a vcg alone and re-using the reasoning of previous proofs can save significant effort. The seL4 proofs for two properties of a single function were reduced from 407 lines of proof script to 95 by using function annotations. This resulted in more comprehensible proofs that better captured the logic of the properties being shown. By extending the use of function annotations to additional functions of similar complexity, we expect to see comparable improvements to existing and future proofs.

Acknowledgements

Thanks to Toby Murray and Gerwin Klein for valuable feedback on earlier drafts of this paper.

References

[1] 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.

[2] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte & Stephan Tobies (2009): *VCC: A Practical System for Verifying Concurrent C*. In Stefan Berghofer, Tobias Nipkow, Christian Urban & Markus Wenzel, editors: *22nd TPHOLs*, *LNCS* 5674, Springer-Verlag, Munich, Germany, pp. 23–42.

[3] C. A. R. Hoare (1983): *An axiomatic basis for computer programming*. *Commun. ACM* 26(1), pp. 53–56, doi:10.1145/357980.358001.

[4] 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.

[5] Till Mossakowski, Lutz Schrder & Sergey Goncharov (2008): *A Generic Complete Dynamic Logic for Reasoning About Purity and Effects*. In Jos Fiadeiro & Paola Inverardi, editors: *Fundamental Approaches to Software Engineering*, *Lecture Notes in Computer Science* 4961, Springer Berlin / Heidelberg, pp. 199–214, doi:10.1007/978-3-540-78743-3_15.

[6] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie & Gerwin Klein (2012): *Noninterference for Operating System Kernels*. In: *2nd Int. Conf. Certified Programs & Proofs*. To appear.

[7] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick & Gerwin Klein (2011): *seL4 Enforces Integrity*. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz & Freek Wiedijk, editors: *2nd ITP*, *LNCS* 6898, Springer-Verlag, Nijmegen, The Netherlands, pp. 325–340, doi:10.1007/978-3-642-22863-6_24.

[8] Christoph Sprenger & David Basin (2007): *A Monad-Based Modeling and Verification Toolbox with Application to Security Protocols*. In Klaus Schneider & Jens Brandt, editors: *Theorem Proving in Higher Order Logics*, *Lecture Notes in Computer Science* 4732, Springer Berlin / Heidelberg, pp. 302–318, doi:10.1007/978-3-540-74591-4_23.

[9] Wouter Swierstra (2009): *A Hoare Logic for the State Monad*. In Stefan Berghofer, Tobias Nipkow, Christian Urban & Makarius Wenzel, editors: *Theorem Proving in Higher Order Logics*, *Lecture Notes in Computer Science* 5674, Springer Berlin / Heidelberg, pp. 440–451, doi:10.1007/978-3-642-03359-9_30.