

Towards Structured Proofs for Program Verification (Ongoing Work)

Lars Noschinski

Technische Universität München, Institut für Informatik
noschinl@in.tum.de

Abstract. The first step in program verification is often a call to a verification condition generator (VCG) which decomposes a program into a set of logical formulas, the verification conditions. These verification conditions must then be discharged manually.

The process of constructing the verification conditions loses much of the structure of the original program. In this article, we develop a labeling to extend VCGs to attach annotations to the verification conditions and hence preserve the program structure. We implemented this approach in the Isabelle theorem prover. The implementation is mostly agnostic of the concrete VCG. This is a step towards replacing unstructured proof scripts by structured Isar proofs.

1 Introduction

A common approach to program verification in an interactive theorem prover is the use of verification condition generators. Given a specification of a program, a *verification condition generator* (VCG) uses a set of rules based on the structure of the programming language (sequential composition, loops, conditional statements, assertions, ...) to decompose the specification first into program fragments and finally into a list of logical formulas. Often, these rules consider the postcondition as given and compute the weakest precondition such that the program satisfies the postcondition (an alternative is the computation of a strongest postcondition).

If there is already a fixed precondition, the VCG applies a *consequence rule*. This generates a proof obligation for the user: They must prove that the given precondition implies the computed precondition (i.e., it is stronger than the weakest precondition). Such proof obligations arise at various points in the program: For example, at the beginning of the program (because a fixed precondition is part of the specification) or at the beginning of a while loop (because the weakest precondition of the loop body must follow from the loop invariant and the loop condition). Other rules can introduce proof obligations as well.

Due to the structural nature of the rules, all generated proof obligations can be associated with certain points in the program. Knowing from which point in the program and from which rule a proof obligation was created helps the user to understand what he needs to prove (because they already have an understanding

of the program). Unfortunately, this information is not directly available. By carefully reading the proof obligation it can be guessed or recovered. However this is tedious, as the proof obligations are often quite large.

The Isabelle theorem prover provides the Isar language for structured proofs. In particular, a proof method can produce *named cases* (the *induct* proof method is an example of this). These give names to the proof obligations, make the premises available as named lemmas and introduce shortcuts for the conclusion to be proved. Moreover, it is possible to nest named cases. This allows grouping proof obligations which belong together.

None of the existing VCGs known to the author generate named cases. As a consequence of this, the user must either do unstructured proofs or state all the assumptions and the goal to be proven in a structured proof by hand. The former is unfortunately not very maintainable: there is no indication which part of the proof belongs to which proof obligation and a failed proof for one proof obligation might break the proofs for all following obligations. The latter is even worse: The proof obligations in program verification are big enough to easily fill the screen. If the program or the specification is modified in minor ways, the amount of work needed for actual changes in the proof can be dwarfed by the effort to restate the now subtly changed assumptions.

In this article, we label terms to attach knowledge of the program structure. We describe a proof method which analyses a goal with labeled proof obligations to generate (nested) named cases. Together, this makes it feasible to write structured proofs for program verification. The author hopes that this makes the proofs both easier to write and to maintain.

The current state of this work is available at http://www21.in.tum.de/~noschin1/VCG_Labeling.

2 Labeled VCG rules

Ideally, the structure of a proof for a program should have the same structure as the program itself. In the following, we assume that the VCG is built upon a set of introduction rules (although the concept should work with other VCGs, too). To attach the program structure to the proof obligations, we introduce the following constant:

definition LABEL :: "prg_ctxt list $\Rightarrow$ nat $\Rightarrow$ nat $\Rightarrow$ 'a $\Rightarrow$ 'a" **where**
 "LABEL ctxt inl outl x $\equiv$ x"

The constant has four parameters: The semantics are solely determined by the last parameter, the program fragment (or the resulting proof obligation). The first parameter is the surrounding context and the two numbers are the first and last “line number” (or rather, an approximation thereof) of the program fragment. The purpose of the first line number is to preserve the order of the program statements; the last line number is needed for computing the line num-

bers of the remainder of the program. The context records the block structure of the program¹:

```

typedecl prg_ctxt_var
typedecl prg_ctxt =
  Block string nat "prg_ctxt_var list" /
  Split "(prg_ctxt_var * prg_ctxt_var) list"

consts Var :: "'v  $\Rightarrow$  prg_ctxt_var"

```

The *Split* constant will be discussed later. The *Block* records takes the name and the starting line number of the block and a list of universally quantified variables introduced in this block. There will be two kinds of blocks: First, those representing the structure of the program (e.g., while loops, if/then, if/else) and second further information about the genesis of the proof obligations (e.g., inside a while loop, there will be blocks for the preservation of the invariant and for the postcondition).

Let us consider some examples. We consider program specifications of the form $\{P\} c \{Q\}$ where P is the precondition, Q the postcondition and c the program. We start with the trivial labeling:

```

LABEL [] 0 out1 ( $\{P\} c \{Q\}$ )

```

As an first example, we consider the rule for sequential composition. The programming language we use here is the one used in the AutoCorres [4] tool – the imperative state is encapsulated in a monad, variable assignments are expressed by lambda abstractions:

$$\frac{\{Q \ x\} \ g \ x \ \{R\} \quad \{P\} \ f \ \{Q\}}{\{P\} \ f \ >>= (\lambda x. \ g \ x) \ \{R\}}$$

Here, $>>=$ is the monadic bind operation. By applying this rule, we get two new proof obligations. The second is for the first command f (the reversed order is due to the computation of weakest preconditions), the first for g applied to the result of f . Now, f has starts at the same line and in the same context as the composed program. g however starts at the line following the last line of f and ends with the last line of the composed program. In imperative programs, sequential composition is usually not considered as opening a new block; however, x is introduced here (by the universal quantifier; as the result of f), which is the reason we open a new block *bind* here (the scope of x is exactly the program fragment g). These considerations lead to the following labeling:

```

lemma labeled_bind:
  assumes " $\bigwedge x. \text{LABEL } (\text{Block } \text{'bind'} \ (\text{Suc } \text{out11}) \ [\text{Var } x] \ \# \ \text{ctxt})$ 
     $(\text{Suc } \text{out11}) \ \text{out1} \ (\{Q \ x\} \ g \ x \ \{R\})$ "
  and " $\text{LABEL } \text{ctxt} \ \text{in1} \ \text{out11} \ (\{P\} \ f \ \{Q\})$ "
  shows " $\text{LABEL } \text{ctxt} \ \text{in1} \ \text{out1} \ (\{P\} \ f \ >>= (\lambda x. \ g \ x) \ \{R\})$ "

```

¹ There is no deeper reason for *prg_ctxt* to be a datatype instead of a type declaration. This is just a detail of our current implementation.

A more complicated labeling is necessary for the rule for while loops. The *PREC* constant annotates names to the various preconditions; *land* is a conjunction operator.

```

lemma labeled_while:
  fixes ctxt inl
  defines "ctxt'  $\equiv$   $\lambda a$ . Block ''while'' inl [Var a] # ctxt"
  assumes " $\bigwedge r m$ . LABEL (Block ''invariant'' inl [Var m] # ctxt' r) inl outl
     $\{ \text{PREC ''inv'' (A r)}$ 
       $\text{land PREC ''cond'' (c r)}$ 
       $\text{land PREC ''var'' } (\lambda s. M r s = m) \}$ 
    b m r
     $\{ \lambda r s. A r s \wedge M r s < m \}$ "
  and " $\bigwedge r s. A r s \implies \neg c r s \implies$ 
    LABEL (Block ''postcondition'' (Suc outl) [Var s] # ctxt' r)
    (Suc outl) (Suc outl) (Q r s)"
  shows "LABEL ctxt inl (Suc outl)  $\{ A r \}$  while A M c b r  $\{ Q \}$ "

```

Here, *while* is a loop with invariant *A*, termination measure *M*, loop condition *c*, and loop body *b*. The initial value of the loop state (i.e., the variables modified in this loop) is *r*. Again, we get two proof obligations. Both are related to the loop, so they are annotated with a *while* block; the loop state (the first universally quantified variable in both goals), is common to both of them, so we bind it in this block. The blocks *invariant* and *postcondition* correspond to the two proof obligations arising for every while loop: invariant preservation and the fact that after leaving the loop the postcondition is fulfilled. In both of these cases, an additional variable is introduced and therefore recorded in context.

The loop state often consists of more than one variable. This is usually expressed by using a tuple. As a result, the loop body often has the form $\lambda(a\ b).$ *f a b*, or – if written without syntax sugar – *prod_case f*. To allow further recursive decomposition of *f*, the *prod_case* constant is removed by the following rule:

```

lemma labeled_prod_case:
  assumes " $\bigwedge x y$ . LABEL (Split [(Var v, Var (x,y))]) # ctxt) inl outl
     $\{ P\ x\ y \} B\ x\ y\ \{ Q \}$ "
  shows "LABEL ctxt inl outl
     $\{ \text{case } v \text{ of } (x, y) \Rightarrow P\ x\ y \} \text{ case } v \text{ of } (x, y) \Rightarrow B\ x\ y\ \{ Q \}$ "

```

This rule introduces two new variables *x* and *y* such that the tuple (x,y) is now used in place of *v*. The previously mentioned *Split* context records exactly this fact.

Note that the labeled rules we have presented here need to be derived by hand from the existing unlabeled rules. For the rules the author has labeled so far, no functional changes in the rule were necessary, so the proofs are trivial (by unfolding the definition of *LABEL*). Finding the right labeling is usually straightforward.

3 A Proof Method

So far we have explained only the labeling, without considering how to turn it into structured proofs. We implemented a proof method which generates named cases from a labeled goal. As an example, we consider a program for multiplication by iterated addition:

```

definition "mult P m n  $\equiv$ 
  DO
    (_, r)  $\leftarrow$  while
      (mult_inv P m n) (* invariant*)
      ( $\lambda(i, \_) \_. i$ ) (* variant *)
      ( $\lambda(i, \_) \_. 0 < i$ ) (* loop condition *)
      ( $\lambda(i, r). DO$  (* loop body *)
        return (i - 1, r + m)
      OD)
    (m, 0); (* initial values of loop variables *)
  return r
OD"
```

This will give us the following structure of named cases:

```

case (weaken s)
case (while i r) {
  case (invariant m) {
    case (weaken s)
  }
}
next
  case (postcondition s)
}
```

The *weaken* cases are generated by the consequence rule we mentioned in the introduction. The while loop has no surrounding block and contains two other blocks (as can be directly seen from the while rule we discussed above). Further decomposition of the *invariant* proof obligation required the use of the consequence rule, so there is a case nested in there.

For a nested while loop, then the cases for the inner loop would appear where we expect them: Inside the *invariant* case for the outer loop. In this example, we had no need for the line numbers. But for two successive while loops, the while cases would have been disambiguated by marking the second with a 2.

Now, in each of the named cases, the variable *?case* is bound to the conclusion of the proof obligation, while the premises are made available as a named lemma. In our implementation, we also introduced a constant to label the individual premises of the proof obligations (for example, see the *PREC* constant in the while rule), so these get predictable names, too.

The variable names attached to each case fix the names of the universally bound variables in each proof obligation, as recorded by the variable part of the *Block* annotations. If we consider the while rule again, we see that it only binds

one variable, not two: However, our proof method sees that this variable is a tuple and splits it into its components, before producing the named cases.

For decomposing the loop body, the rule for *prod_case* must be used. As a result, the variables bound by the *while* case are not the ones which are actually used in the loop body (as the *prod_case* rule introduced new ones). However, the *Split* annotation ties these variables together and our proof method massages the proof obligation (by reordering the bound variables and dropping the unused ones), such that the *while* case now refers to the newly introduced ones (the proof method must ensure that for both sets of variables, all tuples are split into the same components).

4 Future Work

The technique described in this paper already allows us to expose which proof obligation belongs to which program position and make a partly structured proof. In particular, intermediate lemmas necessary to discharge one of the proof obligations can be stated using the full power of Isar, without resorting to old style methods like *subgoal_tac*.

The proof method producing the nested cases should be mostly agnostic of the VCG used. It would be interesting to evaluate a wide range of VCGs to see whether this is actually the case.

For the VCG we considered, conditional rules do not create proof obligations on their own. As a result, the branching does not (always) become explicit in the structure of the named cases, but stays hidden in one of the proof obligations. Doing this prevents exponential blowup of the number of proof obligations, so this is often a preferred way of stating the rules. Similar effects occur with assertions embedded in the program. Therefore, we will investigate what is the best way of presenting this structure to the users.

Moreover, a VCG often generate large subterms, for example for modified program state. In many proofs, the user will need to refer to these terms, so these should be made available to the user as term bindings. In his work on implementing separation logic on top of Simpl, Gast [3] extracted such terms. It remains to be seen whether this can be done in a way which keeps the generation of nested cases agnostic of the VCG.

Related to these considerations, the generated proof obligations can often considerably simplified by using a few weak proof methods (for example, *clarsimp*), making the goal more readable or even solving trivial proof obligations [2]. One needs to take care however, that this does not destroy the structure we carefully annotated. Also, one important property of methods generating structured proofs is that they have a predictable behaviour; uncontrolled simplification often destroys this. It would be worthwhile to develop a sensible setup here.

Isar's concept of nested named cases proved to be a good fit for the kind of structure we want to expose. Occasionally, the author wished for a bit more expressiveness. For example: writing *case (while (i,r))* instead of *case (while i r)* would be nice; maybe even splitting the variables only on demand. However,

this would considerably complicate the proof language, so it is dubious whether this would yield a net benefit.

An ambitious project would be introducing a dedicated UI for program verification: Instead of using named cases to hint at the right part of the program, such an UI would display the program, leaving gaps at those points where a manual proof is needed. The user could fill these gaps with a proof, similar to the way programs are annotated for automated code verifiers like VCC [1], but providing the full power of an interactive prover.

For another approach of structuring program verification proofs, albeit with a different focus, see the work on Ribbon Proofs [5].

Acknowledgements The author thanks Peter Lammich for many fruitful discussions.

References

1. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Proceedings of TPHOLs 2009, pp. 23–42. Springer
2. Correnson, L.: Qed. computing what remains to be proved. In: Proceedings of NFM 2014, pp. 215–229. Springer (2014)
3. Gast, H.: Semi-automatic proofs about object graphs in separation logic. Electronic Communications of the EASST 53(0) (Jun 2012)
4. Greenaway, D., Andronick, J., Klein, G.: Bridging the gap: Automatic verified abstraction of C. In: Proceedings of ITP 2012, pp. 99–115. Springer (Jan 2012)
5. Wickerson, J., Dodds, M., Parkinson, M.: Ribbon proofs for separation logic. In: Programming Languages and Systems, pp. 189–208. Springer (2013)