7. Refined Verification Condition Generation

Program Verification

ETH Zurich, Spring Semester 2018
Alexander J. Summers

Weakest Preconditions So Far

- Weakest preconditions provide a means of reducing the question:

  does a program have any failing traces (under a specified precondition)?

to an SMT problem: *is* $\mathrm{pre} \wedge \neg \mathrm{wlp}(\mathrm{s},\mathrm{post})$ *unsatisfiable?*

  - A `sat` (or `unknown`) result means that an error trace (possibly) exists

- Some problems with our definition of weakest preconditions:
  - It can generate *(exponentially) large formulas* (in the size of the program)
  - It doesn't tell us *which assertion(s)* in the program potentially fail
  - Similarly, we don't know *how many assertion violations* are possible

- We will examine some of these issues, and some possible solutions
  - Formula size and error localisation are *also relevant for the second project*

Formula and Expression Duplication

- The definition of wlp causes duplication of formulas and expressions
  - The two main culprits are: *assignment statements* and branching statements
- Recall the rule for assignments: $\mathrm{wlp}(\mathrm{x} := \mathrm{e}, \mathrm{A}) = \mathrm{A}[\mathrm{e}/\mathrm{x}]$
- This has two negative effects:
  - It introduces *copies of the expression* e
  - It produces a *new formula*, with new sub-formulas etc.
- Now consider e.g. non-deterministic choice (if, while are similar):
  - $\mathrm{wlp}(\mathrm{s}_1[]\mathrm{s}_2, \mathrm{A}) = \mathrm{wlp}(\mathrm{s}_1, \mathrm{A}) \wedge \mathrm{wlp}(\mathrm{s}_2, \mathrm{A})$
- This results in two copies of the formula A
  - One might consider rewriting to avoid this duplication (cf. Tseitin CNF)
  - But identical copies of A *might not persist*, due to assignments in s_1 or s_2

Eliminating Assignments

- The above problems could be solved if we could *remove assignments*
- A naïve idea: how about rewriting an assignment as follows?
  - Replace statements $\mathrm{x} := \mathrm{e}$ with $\mathrm{assume\ x{=}e}$
- This transformation doesn't account for the *different values* x takes
  - e.g. $\mathrm{x} := \mathrm{x}+1$ would become $\mathrm{assume\ x{=}x{+}1}$: introduces inconsistency
- *But*, this would work if each variable were assigned to *at most once*
  - Idea: first convert the program into e.g. *static single assignment (SSA) form*
  - Then the above naïve transformation would actually be valid
  - This allows duplicate formulas from wlp to survive/be "factored out"
  - e.g. define $\mathrm{wlp}(\mathrm{s}_1[]\mathrm{s}_2, \mathrm{A}) = (\mathrm{p}\Leftrightarrow\mathrm{A})\Rightarrow\mathrm{wlp}(\mathrm{s}_1, \mathrm{p})\wedge\mathrm{wlp}(\mathrm{s}_2, \mathrm{p})$ (fresh atom p)
- In fact, *Dynamic Single Assignment (DSA)* is sufficient for us…

Converting to Dynamic Single Assignment I

- A program is in dynamic single assignment (DSA) form, if:
  - in *each trace* of the program, each variable is assigned at most once
  - this isn't typically possible for loops with assignments; instead we *desugar loops* first (as was explained in slide 152)
- For example, $(\mathrm{x}{:=}0)[\,](\mathrm{x}{:=}1)$ is in DSA form (but not in SSA form)
- Conversion to DSA can be done by introducing *versions of variables*
  - For example, replace original variable x with versions x_0, x_1, x_2, …
- For straight-line code, the conversion is simple:
  - e.g. $\mathrm{x}{:=}0;\ \mathrm{x}{:=}\mathrm{x}+1;\ \mathrm{x}{:=}\mathrm{x}\text{-}4$ could become $\mathrm{x}_0{:=}0;\ \mathrm{x}_1{:=}\mathrm{x}_0+1;\ \mathrm{x}_2{:=}\mathrm{x}_1\text{-}4$
- Idea: track the latest version of each variable; use this in expressions
  - For assignments, increment version; use the new version for left-hand-side
  - Replace havoc statements with skip but increment the version of the variable

Converting to Dynamic Single Assignment II

- For branching statements (if, non-deterministic choice) we need more
  - Idea*: process each branch independently*, introducing new versions
  - *Per variable*, if *different* final versions are used in the two branches: introduce a version unused in both branches; assign latest value to this in each branch
- For example, $(x{:=}0;\ x{:=}x{+}1)[\,](x{:=}1)$ could become $(x_0{:=}0;\ x_1{:=}x_1{+}1;\ x_2{:=}x_1)[\,](x_0{:=}1;\ x_2{:=}x_0)$ (x_2 is new version of x)
- In this way, we get a new program as follows:
  - *Eliminate all loops* (via their invariants), as shown in slide 152
  - Apply the *DSA transformation* to the resulting program
  - *Eliminate all variable assignments* $x := e$ by replacing with $\mathrm{assume}\ x{=}e$
- Optionally, we can rewrite further (reducing the statement cases):
  - $\mathrm{skip} \rightsquigarrow \mathrm{assume\ true}$ and $\mathrm{if}(b)\{s_1\}\mathrm{else}\{s_2\} \rightsquigarrow (\mathrm{assume}\ b;\ s_1)[\,](\mathrm{assume}\ \neg b;\ s_2)$
  - $\mathrm{wlp}(s_1[\,]s_2\ ,A)$ can be redefined as on slide 160, to avoid formula duplication

Efficient Weakest Preconditions

- By slides 159-162 we can *reduce any program* to a new program in DSA form consisting of only the following constructs:
  - $\mathrm{s}_1;\mathrm{s}_2$
  - $\mathrm{assert}\ \mathrm{A}$
  - $\mathrm{assume}\ \mathrm{A}$
  - $\mathrm{s}_1[\,]\mathrm{s}_2$
- Furthermore, we can reduce checking $\vDash\{\mathrm{A}_1\}\ \mathrm{s}\ \{\mathrm{A}_2\}$ to checking the program $\mathrm{assume}\ \mathrm{A}_1;\ \mathrm{s};\ \mathrm{assert}\ \mathrm{A}_2$ has *no failing traces*
  - i.e., checking that $\mathrm{A}_1\wedge\neg\mathrm{wlp}(\mathrm{s};\ \mathrm{A}_2)$ is *unsatisfiable*
- For this class of programs, our wlp operator (refined as on slide 160) generates formulas which are *linear in the size of the program*
  - DSA conversion adds *extra variables*; typically *tightly correlated* with others
- Suppose that $\mathrm{A}_1\wedge\neg\mathrm{wlp}(\mathrm{s};\ \mathrm{A}_2)$ is found to be *satisfiable*
  - This indicates a potential error – how do we decide *where* the error is?

Multiple Errors

- How many errors should we report, for the following program?

$$
\begin{array}{l}
\text{if}(x>0)\ \{ \\
\quad \text{assert } x=2 \\
\}\ \text{else}\ \{ \\
\quad \text{assert } x<0; \\
\quad \text{assert } x\neq 0 \\
\}
\end{array}
$$

- How many different counter-examples *could* the SMT solver produce?
  - *Counter-examples for the program being correct* are models for which it fails
  - *infinitely many* values of x cause first assert to fail – report this as one error
- The third assertion is only false when the second one is
  - We shouldn't report an error for the last one; it *can't be reached by any trace*
- We will consider two different approaches for *localising errors*

Sets of Verification Conditions

- One way to specify error locations is to *split* verification conditions
- Recall the rule for assert statements: $\mathrm{wlp}(\mathrm{assert}\ A_1\ ,A) = A_1 \wedge A$
- This reflects two different ways in which the program could *fail*:
  - The assert statement could cause a failure (if A_1 could be false)
  - The remainder of the program could encounter a failure (if A could be false)
- We reflect that at most one can happen, using $\mathrm{assert}\ A_1;\ \mathrm{assume}\ A_1$
  - Now we get $\mathrm{wlp}(\mathrm{assert}\ A_1;\ \mathrm{assume}\ A_1\ ,A) = A_1 \wedge (A_1 \Rightarrow A)$
  - We replace all original $\mathrm{assert}\ A_1$ statements with $\mathrm{assert}\ A_1;\ \mathrm{assume}\ A_1$
- Idea: suppose we track multiple verification conditions separately
  - we generalise our wlp operator to wlp^* working on *multisets of assertions* Δ
  - For example, we define $\mathrm{wlp}^*(\mathrm{assert}\ A_1\ ,\ \Delta) = \Delta \cup \{A_1\}$
  - Each element of our multiset comes from a *distinct program point* (assert)

Wlp* (Sets of Verification Conditions)

- For *annotated programs*, our wlp^* definition is as follows:

  $\mathrm{wlp}^*(\mathrm{s}_1;\mathrm{s}_2\ ,\Delta) = \mathrm{wlp}^*(\mathrm{s}_1,\ \mathrm{wlp}^*(\mathrm{s}_2,\ \Delta))$
  $\mathrm{wlp}^*(\mathrm{assert}\ \mathrm{A}_1\ ,\Delta) = \Delta \cup \{\mathrm{A}_1\}$
  $\mathrm{wlp}^*(\mathrm{assume}\ \mathrm{A}_1\ ,\Delta) = \{\mathrm{A}_1 \Rightarrow \mathrm{A} \mid \mathrm{A} \in \Delta\}$
  $\mathrm{wlp}^*(\mathrm{s}_1[]\mathrm{s}_2\ ,\Delta) = \mathrm{wlp}^*(\mathrm{s}_1,\ \Delta) \cup \mathrm{wlp}^*(\mathrm{s}_2,\ \Delta)$
  - recall: all other statements can be desugared to these
- To verify a Hoare triple $\{\mathrm{A}_1\}\ \mathrm{s}\ \{\mathrm{A}_2\}$ we check a *set of entailments:*
  - check the entailment $\mathrm{A}_1 \vDash \mathrm{A}$ for each $\mathrm{A} \in \mathrm{wlp}^*(\mathrm{s},\ \{\mathrm{A}_2\})$
- If we also record the *program point* at which each element of our multisets originated, we can now easily *report error locations*
  - Each failing entailment means the originating assert statement could fail

Wlp* Advantages and Disadvantages

- The wlp^* idea outlined here is a simple way to localise errors
  - Since it is simple, you might want to *use it in your second project*
- For *purely propositional* (i.e. boolean) programs:
  - It amounts to splitting the search for a model into several smaller searches
  - Each one repeats structure from "earlier in the program"; some redundancy
  - This *might be faster (or slower)* than performing a single search using wlp
- For large general programs (SMT, rather than SAT) it can be slow
  - *Theory-specific work may be repeated* for each entailment checked
  - Similarly, *quantifier instantiations might be repeated* for each entailment
- We'll examine one alternative approach to error localisation
  - requires some mild cooperation from the SMT solver, but e.g. Z3 supports it

Adding Labels to Assertions I

This slide was not covered in the lecture; the material here is not examinable

- Consider the following transformation on assert statements:
  - For each $\mathrm{assert}\ \mathrm{A}_1$ statement, pick a fresh propositional atom l, and replace the statement with $\mathrm{assert}\ \mathrm{A}_1 \vee \mathrm{l}$
  - We call l the *label for the assert statement*
- The original program has a failing trace iff the new one does
  - If we could violate e.g. A_1 above, then take a similar model in which l is false
  - As assert statement can *only* lead to a failing trace *if its label is made false*
- Recall, a (potential) failure is detected when we get `sat` or `unknown`
  - in both cases, we can ask the SMT solver for a model (of $\mathrm{pre} \wedge \neg \mathrm{wlp}(\mathrm{s},\mathrm{post})$)
  - for `unknown`, we still get a *candidate model* (might not satisfy quantifiers)
  - Idea: in either case, check this model for any *false labels* (literals $\neg \mathrm{l}$)
  - Are these guaranteed to identify the failing assertions?...

Adding Labels to Assertions II

This slide was not covered in the lecture; the material here is not examinable

- Are false labels a good way to identify the failing assert statements?
  - *Not yet: there are several technical problems…*
- *Problem 1*: labels will be *pure literals* in the SMT input
  - there will be a single negative (*why?*) occurrence in the SMT query generated
  - recall: *pure literals can be eliminated* (cf. SAT algorithms)
  - even if not eliminated, the solver might *eagerly choose these to be false*
  - we could get *no negative labels*, or *too many*
- Solution: Z3 (and Simplify) have explicit syntax for *specifying labels*
  - Solver won't eliminate these, and will *postpone deciding on them*
  - Effectively, given $\mathrm{assert}\ \mathrm{A}_1 \lor \mathrm{l}$ the solver will only make l false once it's already managed to make A_1 false, at which point making l false *gives a failing trace*

Adding Labels to Assertions III

This slide was not covered in the lecture; the material here is not examinable

- *Problem 2*: we will only identify one failing assertion this way
  - the negative label $\neg \mathrm{l}$ returned will identify a failing assertion
- Solution: we can ask for a *different potential error* as follows:
  - Add the extra assumption that l is *true* to our original SMT query
  - This has the effect of "switching off" the assert statement $\mathrm{assert}\ A_1 \vee \mathrm{l}$
  - If we get a *new potential failure* (and negative label), we can iterate
- Z3 (as well as other SMT solvers) supports *interactive mode:*
  - After a query, the solver *retains its internal state*
  - Extra assumptions can be added, and extra `check-sat` commands made
  - Results of clause learning, theories, quantifier instantiation etc. are retained
- Using interactive mode, we can *efficiently generate the set of errors*

Advanced Verification Condition Generation - Summary

- We have seen improvements to the weakest precondition approach
  - Two main problems addressed: *size of formulas* and *error localisation*
- Converting to DSA (or SSA) form allows *eliminating assignments*
  - We can *then prevent formula duplication*, similarly to Tseitin CNF conversion
- One way to localise errors: generating *multiple verification conditions*
  - This approach is simple, but can result in *repeated work* for the SMT solver
- The *labels* mechanism provides an alternative approach
  - To be effective, it requires *native support from the SMT solver*
- With these tricks, *efficient verification conditions* can be generated
  - The DSA and labels ideas are used in industrial-strength tools, such as Boogie
- We will look next at the *Boogie intermediate verification language*
  - The Boogie verifier uses (extensions of) the techniques we have learned here

Refined Verification Condition Generation – References

- Weakest Preconditions:
  - *Guarded commands, nondeterminacy and formal derivation of programs.* Edsger W. Dijkstra (1975)
  - *Avoiding exponential explosion: generating compact verification conditions.* Cormac Flanagan, James B. Saxe (2001)
  - *Weakest-precondition of unstructured programs.* Mike Barnett, K. Rustan M. Leino (2005)

- Labels for Error Localisation:
  - *Generating error traces from verification-condition counterexamples.* K. Rustan M. Leino, Todd Millstein, James B. Saxe (2005)

- Other teaching material:
  - *Synthesis, Analysis, and Verification.* Viktor Kuncak (EPFL)