Recap

Given a definition of VC

- \( VC(skip, \{Q\}) = Q \)
- \( VC(x = e, \{Q\}) = Q[e/x] \)
- \( VC(C1; C2\{Q\}) = VC(C1\{VC(C2\{Q\})\}) \)
- \( VC(if\ B\ then\ C1\ else\ C2\{Q\}) = (B\ and\ VC(C1\{Q\}))\ or\ (\neg B\ and\ VC(C2\{Q\})) \)
- \( VC(while; e\ do\ c, B) = \)
  \[ l \land \forall x_1, \ldots, x_n I \Rightarrow (e \Rightarrow VC(c, l) \land \neg e \Rightarrow B) \]
  
  - Where \( x_i \) are variables modified in \( c \).
- \( VC(assert\ B\ \{Q\}) = \{Q \land B\} \)
- \( VC(assume\ B\{Q\}) = \{B \Rightarrow Q\} \)

\( (A \Rightarrow VC(P\{B\})) \Rightarrow \{A\}P\{B\} \)

- \( (A \Rightarrow VC(P\{B\})) \) is equivalent to \( VC(assume\ A; P; assert\ B\ \{true\}) \)
Synthesizing invariants

Given a program with pre/post conditions, the only obstacle to making verification automatic is finding the invariants

- (and checking the resulting formula if it happens to be in a complicated logic)
Example

{y=y_0, k=k_0, t=y_0-k_0}
while(t>0){
    y = y-1;
    t = t-1;
}
Assert y <= k_0;
VC as a sketch harness

∀y, y₀, k, k₀, t, y', t', y'', (y = y₀ ∧ k = k₀ ∧ t = y₀ - k₀) ⇒ inv(y, y₀, k, k₀, t)

inv(y', y₀, k, k₀, t') ∧ (t' > 0) ⇒ inv(y' - 1, y₀, k, k₀, t' - 1)

inv(y'', y₀, k, k₀, t'') ∧ ¬(t'' > 0) ⇒ y'' ≤ k₀,

harness void main(int y, int y_0, int k, int k_0, int t,
                  int yp, int tp, int ypp, int tpp ){
    if(y==y_0 && k==k_0 && t==y_0-k_0){
        assert inv(y, y_0, k, k_0, t);
        if(inv(yp, y_0, k, k_0, tp) && tp>0){
            assert inv(yp-1, y_0, k, k_0, tp-1);
        }
        if(inv(ypp, y_0, k, k_0, tpp) && tpp<=0){
            assert ypp <= k_0;
        }
    }
}
VC as a sketch invariant

```c
#include "generators.skh"
pragma options "--bnd-inline-amnt 3"

bit inv(int y, int y_0, int k, int k_0, int t){
    return exprBool({y, y_0, k, k_0, t}, {PLUS});
}
```
If we find a solution that works for all inputs, we know the program verifies

Sketch has no loops so bounded execution is not a problem

Sketch is only considering small inputs
  • To get full guarantees we need to check that the resulting invariant works for *all* inputs
  • We can check that with an SMT solver
What if we want to synthesize the function body too?

Candidate solution:
- Inline all generators
- Pretend that ?? is just another expression
- Generate VC from that program.
- Solve for all unknowns
- Plug them back into the original program

This almost works
Verified synthesis

\{P\}
While(true)\{  \}
\{Q\}

This is valid for any P and Q!
From partial to total correctness

Total correctness judgment

• $\vdash [A] c [B]$
• Just like before, but must also prove termination

What about loops
Ranking function

Function F of the state that

- a) Maps state to an integer (in general, a well-ordered set)
- b) Decreases with every iteration of the loop
- c) Is guaranteed to stay greater than zero
- Also called variant function

If a ranking function exists, program is guaranteed to terminate
We can synthesize that too!
From program verification to program synthesis

Srivastava, Gulwani, Foster
Key ideas

Structured program statements to reduce symmetries
Language supports only the following simple grammar

\[ P := \{ g \rightarrow s \} | P; P | \text{while}(e)P \]

\[ g \rightarrow s \] corresponds to a guarded assignment of the form:

- \( If(g) \{ s_1 \ldots s_k = e_1 \ldots e_k \} \)

- Parallel assignment avoids the symmetries that arise from having to compute all possible permutations of assignments.
Proof-Theoretic Synthesis

Input: Program Scaffold
- Flowgraph template (a nested loop structure)
- Function spec (pre-/post-conditions)
- Other constraints (temp variables, size limit, etc.)

Output:
- Synthesized code fragments
- Associated loop invariants and ranking functions

Synthesis Condition:
- Generalization of verification conditions
- A valid program exists iff. the SC has a satisfying solution
Example: Square Root

\[
\text{IntSqrt}(\text{int } x) \{ \\
v := 1; i := 1; \\
\text{while}^\tau \varphi (v \leq x) \\
\quad | v := v + 2i + 1; i++; \\
\text{return } i - 1;
\}
\]

\textbf{Invariant }^\tau:\ \\
v = i^2 \land x \geq (i-1)^2 \land i \geq 1

\textbf{Ranking function } \varphi:\ \\
x - (i-1)^2
Program Scaffold

Pre: $x \geq 1$

choose $\{g_1 \rightarrow s_1\}$;
while $\tau;\rho (g_0)$ {
    choose $\{g_2 \rightarrow s_2\}$;
};
choose $\{g_3 \rightarrow s_3\}$

Post: $(i - 1)^2 \leq x < i^2$

(Assume only one branch in each fragment)
Verification Condition

\[
\begin{align*}
  x \geq 1 & \land g_1 \land s_1 \quad \Rightarrow \quad \tau' \\
  \tau \land g_0 \land g_2 \land s_2 & \quad \Rightarrow \quad \tau' \\
  \tau \land \neg g_0 \land g_3 \land s_3 & \quad \Rightarrow \quad (i' - 1)^2 \leq x' \land x' < i'^2
\end{align*}
\]

choose \{ \langle g_1 \rightarrow s_1 \rangle \};
while_{\tau;\varphi} (g_0) {
  choose \{ \langle g_2 \rightarrow s_2 \rangle \};
};
choose \{ \langle g_3 \rightarrow s_3 \rangle \}
Well-formedness Condition

choose $\{[] g_1 \rightarrow s_1\}$;
while $\tau; \phi (g_0)$ {
    choose $\{[] g_2 \rightarrow s_2\}$;
};
choose $\{[] g_3 \rightarrow s_3\}$

$g_1, g_2, g_3 = \text{true and } \text{valid}(s_1) \land \text{valid}(s_2) \land \text{valid}(s_3)$
Ranking Function

choose $\{\llbracket g_1 \rightarrow s_1 \rrbracket \}$;

while $\tau; \varphi \ (g_0)$ {
  choose $\{\llbracket g_2 \rightarrow s_2 \rrbracket \}$;
};

choose $\{\llbracket g_3 \rightarrow s_3 \rrbracket \}$

$r_i = \varphi_i \land (\tau \Rightarrow r_i \geq 0) \land (\tau \land g_0 \land g_2 \land s_2 \Rightarrow r_i' > \varphi_i')$
Synthesis Conditions

\[ sc = \text{SafetyCond}(exp, F) \land \text{WellFormCond}(exp) \land \text{RankCond}(exp) \]

\[
x \geq 1 \land s_1 \Rightarrow \tau'
\]

\[
\tau \land g_0 \land s_2 \Rightarrow \tau'
\]

\[
\tau \land \neg g_0 \land s_3 \Rightarrow (i' - 1)^2 \leq x' \land x' < i'^2
\]

\[
\text{valid}(s_1) \land \text{valid}(s_2) \land \text{valid}(s_3)
\]

\[
r_l = \varphi_l \land (\tau \Rightarrow r_l \geq 0) \land (\tau \land g_0 \land s_2 \Rightarrow r'_l > \varphi'_l)
\]

Theorem (soundness and completeness)

The synthesis conditions are satisfiable iff. there exists a program that is valid w.r.t. the scaffold.