Program Refinement

In practice, the logic of programs is used more often and more effectively to guide program synthesis than in direct program verification. Knowledge of a program’s proof structure constrains the ways that programs are improved. Refinement is the practice of starting with a program that is simple and obviously correct, and improving its performance in ways that are known to preserve correctness, or at least perturb their logical meaning in well confined ways.

Synthesis is a program construction process which, intuitively at least, reverses the process of extracting verification conditions. Beginning with the the assertion

\{PRE\} \[\text{post}\]\{POST\}

In other words, solve a purely logical specification by finding a terminating program \(S\) that makes the assertion.

Although programming is ultimately a creative activity, requiring creative invention, it is principally an engineering discipline in which most of the work can be done methodically. The methodology described in this section is based on a deconstruction of inductive assertions producing, as a byproduct, partial programs satisfying those assertions.

Developing Invariants from Postconditions

In The Science of Programming [Gries 1981, ch. 16], Gries describes techniques for deriving loop invariants from certain forms of postconditions. The essence of the idea is to devise the invariant as a weakening of the postcondition, and then use the loop iterations to make it progressively stronger until the postcondition is recovered.

Four techniques that Gries mentions are below. The application of these techniques is not mechanical; it requires insight on the part of the programmer, beginning with the realization that a loop is needed to establish the desired postcondition.

Deleting a conjunct.

If the postcondition is a conjunction, try using one conjunct for the invariant and the other for the loop test:

\{PRE\} \[\text{post}\]\{POST\}

\[
\text{post} \equiv Q \land R
\]

\{PRE\} while \(A\) \{inv: \(B\)\} do \[\text{post}\]
Generalize a constant to a program variable.

If the postcondition describes a property holding over a range (e.g. an array), \( \equiv \forall k \in [1..10]: P[a_k] \) Replace one of the limits, say the ‘10’, with an unused variable \( v \) to get

\[ \forall 1 \leq k \leq v: v \in [1..10] \land P(a_k) \]

When program variable \( v \) is introduced, an assertion is added restricting its range. Technique 1 suggests using the first conjunct as a loop test and the second as an invariant:

\[
\{\text{PRE}\} \quad \rightarrow \quad \{\text{POST}\}
\]

\[
\text{POST} \equiv \forall k \in [M..N]: P[a_k], \ (M, N \text{ constants})
\]

\[
\{\text{PRE}\} \quad \text{while} \quad v \leq N \quad \{\text{INV}: M \leq v \land \forall 0 \leq k \leq v: P[a_k]\} \quad \text{do} \quad \rightarrow \quad \{\text{POST}\}
\]

Of course, \( v \) must be properly initialized in order to establish INV on entry to the loop and the loop body must take care of advancing \( v \) toward a loop terminating value.

Enlarge the range of a variable.

For instance, the postcondition \( 5 \leq i \leq 100 \) could be weakened to the invariant \( \{\text{INV}: 0 \leq i \leq 100\} \) suggesting the loop test \( i < 10 \).

Introduce a disjunct.

The postcondition \( Q \) can be weakened to \( P \lor Q \), suggesting the loop test \( P \).

Applying the Techniques

These techniques and others are often used in combination. They don’t always work—like mathematics, programming ultimately involves ingenuity—but a programmer should bear them in mind when trying to invent a loop to achieve a goal.
Example 1: Linear Search

Search an array \( A[1..n] \) for the smallest element

Use the scheme

\[
\begin{align*}
\text{begin} \\
\quad \text{make a guess}; \\
\text{while } \neg \text{done} \\
\quad \{\text{this is a reasonable guess}\} \\
\quad \text{do} \\
\quad \quad \text{improve the guess} \\
\text{end}
\end{align*}
\]

Choose “reasonable guess” to mean

“\( A[min] \) is the smallest element seen so far”

\[
\begin{align*}
\text{begin} \\
\quad \text{make a guess}; \\
\text{while } \neg \text{done} \\
\quad \{A[min] \text{ is the smallest element seen so far}\} \\
\quad \text{do} \\
\quad \quad \text{look at another element}; \\
\quad \quad \text{revise “so far”} \\
\quad \text{end}
\end{align*}
\]

So the invariant is

\[
\text{INV1 } \equiv \forall 1 \leq i \leq \text{sofar}: A[min] \leq A[i]
\]

To establish the invariant, we can set \( \text{sofar} = min = 1 \). To make progress, increment \( \text{sofar} \).
begin
sofar := 1;
min := 1;
whilesofar ≠ n
  {∀ 1 ≤ i ≤sofar: A[min] ≤ A[i]}
do begin
  sofar := sofar + 1;
  //maintain the invariant!
end
{sofar = n ∧ ∀ 1 ≤ i ≤sofar: A[min] ≤ A[i]}
end

The postcondition implies
∀ 1 ≤ i ≤ n: A[min] ≤ A[i]
as desired.

begin
sofar := 1;
min := 1;
whilesofar ≠ n
  {∀ 1 ≤ i ≤sofar: A[min] ≤ A[i]}
do begin
  sofar := sofar + 1;
    then min := sofar
    else skip
end
{∀ 1 ≤ i ≤ n: A[min] ≤ A[i]}
end
Example 2: Strength Reduction

Suppose we want to write a program to compute the integer square root of an input \( x \); that is, we want to find an \( S \) such that

\[
\{0 \leq x\} \rightarrow \{z^2 \leq x < (z + 1)^2\}
\]

The and in the postcondition suggests a loop, with one conjunct serving as the loop’s test and the other the loop invariant.

\[
z := 0; \\
\textbf{while} \ x \geq (z + 1)^2 \textbf{ do} \\
\{\text{INV: } z^2 \leq x\} \\
z := z + 1 \\
\{z^2 \leq x < (z + 1)^2\}
\]

To get rid of the expensive term \((z + 1)^2\), introduce an “auxiliary” variable, \( u \), to hold this value. The invariant is strengthened to \(\{z^2 \leq x \land u = (z+1)^2\}\) and the loop is adapted to maintain this stronger condition. Let \( u' \) and \( z' \) denote the values of \( u \) and \( z \) after the next loop iteration. The analyses (simultaneous in general)

\[
z' = z + 1
\]

eliminates the squaring operation\(^1\)

Of course, \( u \) must be properly initialized.

\[
z, u := 0, 1; \\
\textbf{while} \ x \geq u \textbf{ do} \\
\{\text{INV: } z^2 \leq x \land u = (z + 1)^2\} \\
\textbf{begin} \\
z := z + 1; \\
u := u + 2z + 1 \\
\textbf{end}
\]

While we can certainly argue that the elimination of \((z + 1)^2\) improves the program, it is still linear in the magnitude of its input. Faster convergence requires an inherently better algorithm, such as a non-restoring square root, which usually is not directly synthesizable directly from the postcondition.

Loop invariants are a formal device for declaring \textit{intent}. They are used here tactically for the more limited purpose of reasoning about incremental computation.

\(^1\)A local optimization analysis might convert \( z, u := z + 1, u + 2(z + 1) + 1 \) to \( z := z + 1; u := u + z + 1 \).
Example 3: Wensley’s Algorithm

Find the quotient $z$ of real numbers $0 \leq x < y \leq 1$ to within tolerance $t$ (without using division, of course).

\[
\{0 \leq x < y \leq 1\} \implies \{z \leq x/y < z + t\}
\]

Programming strategy:

\[
\{0 \leq x < y \leq 1\}
\]

begin

make a guess;

while $\neg$good-enough

\{this is a reasonable guess\}

do

improve the guess

end

\{z \leq x/y < z + t\}

Introduce a variable $d$ to represent the known accuracy of the current guess.

\[\text{INV } \equiv z \leq x/y < z + d\]

Establish the invariant.
\{0 \leq x < y \leq 1\}
begin
z := 0;
d := 1;
while d > t do
\{z < x/y < z + d\}
begin
improve z and d
end
\{z \leq x/y < z + t\}
end

Let’s try a binary search (!?) If \(z + \frac{1}{2}d > x/y\), then \(z \leq x/y < z + \frac{1}{2}d\); otherwise, \(z + \frac{1}{2}z \leq \frac{d}{y} < z + d\). Either way, we know the quotient to within \(\frac{1}{2}d\), so

\{0 \leq x < y \leq 1\}
begin
z := 0;
d := 1;
while d > t do
\{z < x/y < z + d\}
begin
\text{if } z + \frac{1}{2}d > x/y \text{ then } z := z \text{ else } z := z + \frac{1}{2}d;
d := \frac{1}{2}d
end
end
\{z \leq x/y < z + t\}

[Add either a skip statement or a one-branch if to the programming language if the assignment \(z := z\) bothers you.] Fix the “cheat” in the if-test. Get rid of the division by multiplying through by \(y\).

\{0 \leq x < y \leq 1\}
begin
z := 0;
d := 1;
while d > t do
\{z < x/y < z + d\}
begin
\text{if } zy + \frac{1}{2}dy > x \text{ then skip }
\text{else } z := z + \frac{1}{2}d;
d := \frac{1}{2}d
end
end
\{z \leq x/y < z + t\}
The test seems costly. Introduce "trailer variables" \( u \) and \( v \) to hold get rid of the multiplications, subject to invariants \( u = zy \) and \( v = \frac{1}{2}dy \). Now we are obligated to maintain these invariants, depending on the test:

**Case A:** \( u + v > x \):

\[
\begin{align*}
d' &= \frac{1}{2}d \\
z' &= z \\
u' &= z' \cdot y = z \cdot y = u \\
v' &= \frac{1}{2}(d') \cdot y = \frac{1}{2}(\frac{1}{2}d)y = \frac{1}{2}v
\end{align*}
\]

**Case B:** \( u + v \leq x \):

\[
\begin{align*}
d' &= \frac{1}{2}d \\
z' &= z + \frac{1}{2}d = z + d' \\
u' &= z' \cdot y = (z + \frac{1}{2}d)y = zy + \frac{1}{2}dy = u + v \\
v' &= \frac{1}{2}(d') \cdot y = \frac{1}{2}(\frac{1}{2}d)y = \frac{1}{2}v
\end{align*}
\]

\( \{0 \leq x < y \leq 1\} \)

begin
\[ z := 0; \]
\[ d := 1; \]
\[ u := 0; \]
\[ v := \frac{1}{2}y; \]
while \( d > t \) do
\[ \{ z \leq x/y < z + d \land u = zy \land v = \frac{1}{2}dy \} \]
begin
\[ \begin{aligned} & \text{Establish the invariant} \\ & d := \frac{1}{2}d; \\ & \text{if } u + v > x \text{ then skip} \\ & \text{else begin } z := z + d; \ u := u + v; \text{ end;} \\ & v := \frac{1}{2}v \\ & \text{end} \]
end
\[ \{ z \leq x/y < z + t \} \]

This is called Wensley's algorithm [Wensley 58] for real number division, specifically computing the fractional parts of floating point representations.

Since it reduces integer division to addition and divide-by-two, it is a candidate for use in a typical processor, which will have instructions for these operations.

However, this algorithm is not ideally suited for implementation in hardware because addition takes too long (\( \mathcal{O}(n) \) for \( n \)-bit operands). Later, we will see a series of optimizations that improve on Wensley's algorithm that are used in floating point hardware.