Let $a$, $b$, and $c$ be integer variables. Express the following as simply as possible without using quantifiers, assignments, or dependent compositions. Prove.

\[ a := a + b. \quad b := a - b. \quad c := a - b \]

§

\[ a := a + b. \quad b := a - b. \quad c := a - b \]

expansion last assignment

\[ \equiv a := a + b. \quad b := a - b. \quad a' = a - (a - b) \land b' = a - b \land c' = c \]

Substitution Law

\[ \equiv a' = b \land b' = (a + b) - b \land c' = c \]

subtract

\[ \equiv a' = b \land b' = a \land c' = c \]

2. Using the notations and methods of this course, write a program to find the remainder after natural division using only comparison, addition, and subtraction (not multiplication or division or $mod$). Given numerator $n: nat$ and denominator (divisor) $d: nat+1$, find the remainder when $n$ is divided by $d$.

(a) Write a formal specification. Include recursive time.

§

Let $n$ be a natural state variable whose initial value is the numerator and whose final value is the remainder. Let $d: nat+1$ be the divisor (a constant). Let $t$ be time. The problem is $P$ defined as

\[ P \equiv n' < d \land (\exists m: nat \cdot n = m \times d + n') \land t' \leq t + n/d \]

or if you prefer

\[ P \equiv n': 0, \ldots, d \land n: nat\times d + n' \land t' \leq t + n/d \]

(b) Refine your specification.

§

\[ P \iff \text{if } n < d \text{ then ok else } n := n - d. \quad t := t + 1. \quad P \text{ fi} \]

(c) Prove your refinement.

§

The proof is by parts and cases. I have to prove the following 6 refinements.

\[ n' < d \iff n' < d \land ok \]

\[ n' < d \iff n \geq d \land (n := n - d. \quad t := t + 1. \quad n' < d) \]

\[ \exists m: nat \cdot n = m \times d + n' \iff n < d \land ok \]

\[ \exists m: nat \cdot n = m \times d + n' \iff n \geq d \land (n := n - d. \quad t := t + 1. \quad \exists m: nat \cdot n = m \times d + n') \]

\[ t' \leq t + n/d \iff n < d \land ok \]

\[ t' \leq t + n/d \iff n \geq d \land (n := n - d. \quad t := t + 1. \quad t' \leq t + n/d) \]

First, turning it around:

\[ n < d \land ok \implies n' < d \quad \text{expand } ok \]

\[ \equiv n < d \land n' = n \land t' = t \implies n' < d \quad \text{context} \]

\[ \equiv n < d \land n' = n \land t' = t \implies n < d \quad \text{specialization} \]

\[ \equiv T \]

Next, turning it around:

\[ n \geq d \land (n := n - d. \quad t := t + 1. \quad n' < d) \implies n' < d \quad \text{substitution twice} \]

\[ \equiv n \geq d \land n' < d \implies n' < d \quad \text{specialization} \]

\[ \equiv T \]

Next, turning it around:

\[ n < d \land ok \implies \exists m: nat \cdot n = m \times d + n' \quad \text{expand } ok \]

\[ \equiv n < d \land n' = n \land t' = t \implies \exists m: nat \cdot n = m \times d + n' \quad \text{context} \]

\[ \equiv n < d \land n' = n \land t' = t \implies \exists m: nat \cdot n = m \times d + n \quad \text{generalization: } 0 \text{ for } m \]

\[ \equiv n < d \land n' = n \land t' = t \implies n = 0 \times d + n \quad \text{arithmetic} \]

\[ \equiv n < d \land n' = n \land t' = t \implies T \quad \text{base} \]

\[ \equiv T \]
Next, starting with its right side:

\[ n \geq d \land (n := n - d. \ t := t + 1. \ \exists m. \ n = m \times d + n') \quad \text{substitution twice} \]

\[ n \geq d \land \exists m. \ n = m \times d + n' \quad \text{specialize and arithmetic} \]

\[ \Rightarrow \exists m. \ n = (m + 1) \times d + n' \quad \text{change local variable} \]

\[ \exists p. \ n = p \times d + n' \quad \text{widen domain} \]

\[ \Rightarrow \exists m. \ n = m \times d + n' \quad \text{change local variable back to } m \]

Next, turning it around:

\[ n < d \land \text{ok} \Rightarrow t' \leq t + n/d \quad \text{expand } \text{ok} \]

\[ n < d \land n' = n \land t' = t \Rightarrow t' \leq t + n/d \quad \text{context} \]

\[ n < d \land n' = n \land t' = t \Rightarrow t \leq t + n/d \quad \text{arithmetic} \]

\[ n < d \land n' = n \land t' = t \Rightarrow 0 \leq n/d \quad \text{n: nat and } d: \text{nat+1} \]

\[ n < d \land n' = n \land t' = t \Rightarrow T \quad \text{base} \]

\[ = T \]

Last, turning it around:

\[ n \geq d \land (n := n - d. \ t := t + 1. \ \exists m. \ n = m \times d + n') \Rightarrow t' \leq t + n/d \quad \text{substitution twice} \]

\[ n \geq d \land t' \leq t + 1 + (n - d)/d \Rightarrow t' \leq t + n/d \quad \text{arithmetic} \]

\[ n \geq d \land t' \leq t + n/d \Rightarrow t' \leq t + n/d \quad \text{specialization} \]

\[ = T \]

3[15] Let do P od be a loop notation, and within P let exit n when b mean that if b is T then execution exits n enclosing loops, and if b is ⊥ then execution continues with whatever follows exit n when b. Here is a nest of loops. All exits are shown. What refinements need to be proven in order to prove that this nest of loops refines specification S? Hint: further specifications will be needed.
Each loop needs a specification. Using $P$, $Q$, and $R$ for the inner loops,

\[
S \leftarrow A.\ P
\]
\[
P \leftarrow B.\ Q
\]
\[
Q \leftarrow C.\ \text{if } u \text{ then } F.\ \text{if } w \text{ then } H.\ \text{if } x \text{ then } \text{ok}
\]
\[
\quad \text{else } I.\ R \text{ fi}
\]
\[
\quad \text{else } G.\ P \text{ fi}
\]
\[
\quad \text{else } D.\ \text{if } v \text{ then } H.\ \text{if } x \text{ then } \text{ok}
\]
\[
\quad \text{else } I.\ R \text{ fi}
\]
\[
\quad \text{else } E.\ Q \text{ fi fi}
\]
\[
R \leftarrow J.\ \text{if } y \text{ then } \text{ok}
\]
\[
\quad \text{else } K.\ \text{if } z \text{ then } M.\ S
\]
\[
\quad \text{else } L.\ R \text{ fi fi}
\]

4[9] Prove that the following two versions of $\text{nat}$ induction are equivalent.

\[
P_0 \lor \exists n: \text{nat} \cdot \neg Pn \land P(n+1) \iff \exists n: \text{nat} \cdot Pn
\]
\[
\exists n: \text{nat} \cdot \neg Pn \land P(n+1) \iff \exists n: \text{nat} \cdot \neg P_0 \land Pn
\]

\[
(\neg \neg P_0 \lor \exists n: \text{nat} \cdot \neg Pn \land P(n+1)) \iff \exists n: \text{nat} \cdot Pn)
\]

5 Let $a$, $b$ and $x$ be natural variables. Variables $a$ and $b$ are implementer's variables, and $x$ is a user's variable for the operations

\[\text{start} \quad a:= a+1.\ b:= b+2\]
\[\text{ask} \quad x:= a+b\]

Reimplement this theory replacing the two old implementer's variables $a$ and $b$ with one new natural implementer's variable $c$.

(a)[6] What is the data transformer?

\[c = a+b\]

(b)[12] Using your data transformer, transform \text{step}.

\[
\forall a, b.\ c = a+b \Rightarrow \exists a', b'.\ c' = a'+b' \land (a':= a+1.\ b':= b+2)
\]

replace assignment in old variables
\[
= \forall a, b.\ c = a+b \Rightarrow \exists a', b'.\ c' = a'+b' \land a'=a+1 \land b'=b+2 \land x'=x \quad \text{one-point } a', b'
\]
\[
= \forall a, b.\ c = a+b \Rightarrow c' = a+1+b+2 \land x'=x \quad \text{one-point } a \text{ using } a = c-b
\]
\[
= \forall b.\ c' = c+3 \land x'=x\]
\[
= c:= c+3 \quad \text{This is an assignment in the new variable}
The program
\[
\text{chan } c : \text{ int } \quad c? \quad c! 5
\]
decares a fresh channel \( c \), then attempts to read from the channel before writing anything to the channel. Prove that execution of the program deadlocks.

\[\]
§ Inserting the wait for input,
\[
\text{chan } c : \text{ int } \quad t := \text{ max } t (\mathcal{T}_r + 1) \quad c? \quad c! 5
\]

= \[\exists \mathcal{M} : \text{ int}^* \quad \exists \mathcal{T} : \text{ xnat}^* \quad \text{ var } r, w : \text{ xnat} := 0 \cdot \]
\[
t := \text{ max } t (\mathcal{T}_r + 1) \quad r := r+1 \quad \mathcal{M}_w = 5 \land \mathcal{T}_w = t \land (w := w+1)
\]
First, expand \text{ var} and \( w := w+1 \), taking \( r \), \( w \), \( x \), and \( t \) as the state variables.
(\text{It doesn't matter if we leave out } x.)

= \[\exists \mathcal{M} : \text{ int}^* \quad \exists \mathcal{T} : \text{ xnat}^* \quad \exists r, r', w, w' : \text{ xnat} \cdot \]
\[
r := 0 \quad w := 0 \quad t := \text{ max } t (\mathcal{T}_r + 1) \quad r := r+1
\]
\[
\mathcal{M}_w = 5 \land \mathcal{T}_w = t \land r' = r \land w' = w+1 \land x' = x \land t' = t
\]
Now use the Substitution Law four times.

= \[\exists \mathcal{M} : \text{ int}^* \quad \exists \mathcal{T} : \text{ xnat}^* \quad \exists r, r', w, w' : \text{ xnat} \cdot \]
\[
\mathcal{M}_0 = 5 \land \mathcal{T}_0 = \text{ max } t (\mathcal{T}_0 + 1) \land r' = 1 \land w' = 1 \land x' = x \land t' = \text{ max } t (\mathcal{T}_0 + 1)
\]
The conjunct \( \mathcal{T}_0 = \text{ max } t (\mathcal{T}_0 + 1) \) tells us that \( \mathcal{T}_0 \geq \mathcal{T}_0 + 1 \) and therefore \( \mathcal{T}_0 = \infty \).

= \[x' = x \land t' = \infty\]
The theory tells us that execution takes forever because the wait for input is infinite.