Rely/guarantee reasoning

Pre  I assume that this lecture starts at 8:30am
Guarantee you will understand to rely/guarantee reasoning
Rely that you ask questions when you don’t understand
Post  finish this lecture at 9:30am

Overview

- Deriving sequential programs
  - Example: Sieve of Eratosthenes
- Deriving concurrent programs
  - Example: Sieve of Eratosthenes
  - Example: Communicating through a circular buffer
- Semantics of concurrent programs

Your background

Logic and set theory
- Propositional logic: $\land$, $\lor$ and $\neg$
- Predicate logic: $\forall$ and $\exists$
- Set theory: $\in$, $\subseteq$, $\cup$, $\cap$ and {...}
- Specification languages: VDM, Z, B and TLA

Reasoning about programs
- Hoare logic: $\{p\} c \{q\}$
- Refinement calculus or B or Event-B: $\sqsubseteq, x: [p, q]$
- Rely/guarantee concurrency
- Separation logic
- Concurrent separation logic
Reasoning about (concurrent) software

Our main tool is abstraction:
- **sequential** specify components using pre/post conditions
  - e.g. sorting
  - precondition $\text{noduplicates}(s)$
  - postcondition $\text{ordered}(s') \land \text{items}(s') = \text{items}(s)$
- **data** use abstractions such as sets and maps
  - decouple the specification of what the user sees from the implementation
  - avoid the details of the implementations, such as, linked lists and trees
- **process** due to interference between processes need more than pre and post

Compositional reasoning

Reasoning about the whole is decomposed into reasoning about the components

**Why?**
- Make reasoning tractable
- Partition the work (e.g. for multiple people to work on different components)
- Avoid reasoning about paths

```plaintext
j := 0;
while j ≠ N do
  if p then s else t;
  j := j + 1
`)
```

- $2^N$ possible paths

Hoare logic is compositional

Structured reasoning about programs
- **Sequential composition**
  \[
  \frac{\{ p \} s \{ q \} \{ q \} t \{ r \}}{\{ p \} s ; t \{ r \}}
  \]
- **While loop using a loop invariant $p$**
  \[
  \frac{\{ p \land b \} s \{ p \}}{\{ p \} \text{while } b \text{ do } s \{ p \land \neg b \}}
  \]

For termination one needs to add a loop variant or well-founded relation

Parallel composition

Interference possible before or after every atomic step $s_i$ and $t_i$

\[
s_1; s_2; \ldots; s_n \parallel t_1; t_2; \ldots; t_n
\]

- **The number of paths in terms of $n$ explodes**
- **If there is no interference between $s$ and $t$**
  \[
  \frac{\{ p_1 \} s \{ q_1 \} \{ p_2 \} t \{ q_2 \} }{\{ p_1 \land p_2 \} s \parallel t \{ q_1 \land q_2 \}}
  \]
- **But this is the easy case**
Example: Sieve of Eratosthenes (sequential)

- Determine primes up to some given \( n \)
- Illustrates:
  - starting with abstract type (a set)
  - using guarantees (even for a sequential program)
  - introducing loops
  - data refinement to an array of small sets that can each fit in a word

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

REM(2)

REM(3)

Specification in refinement calculus style

Concrete syntax

VDM

\[ \text{SIEVE} \equiv \begin{cases} \text{pre} \ s \subseteq 2 \ldots n \Rightarrow [s = s - C] \\ \text{post} \ s' = s - C \end{cases} \]

Refinement calculus

\[ s: [s \subseteq 2 \ldots n, \ s' = s - C] = \begin{cases} \text{equivalent post condition (set theory)} \\ s: [s \subseteq 2 \ldots n, \ s' \subseteq s \land s - s' \subseteq C \land s' \cap C = \emptyset] \end{cases} \]

\[ \subseteq \text{guarantee on every step} \]

\[ (\text{guar} \ s' \subseteq s \land s - s' \subseteq C) \land s: [s \subseteq 2 \ldots n, \ s' \cap C = \emptyset] \]

The guarantee condition is

- reflexive, i.e. \( s' = s \Rightarrow s' \subseteq s \land s - s' \subseteq C \)
- transitive, i.e. \( s' \subseteq s'' \subseteq s \land s - s'' \subseteq C \Rightarrow s' \subseteq s \land s - s' \subseteq C \)
Introducing a loop

First some set theory

Assume $c_i$ is the set of all multiples of $i$, excluding $i$

\[
\begin{align*}
  s' \cap C &= \emptyset \\
  \equiv s' \cap \bigcup \{ j \in \mathbb{N} \mid 2 \leq j \cdot c_i \} &= \emptyset \\
  \equiv \bigcup \{ j \in \mathbb{N} \mid 2 \leq j \cdot (s' \cap c_i) \} &= \emptyset \\
  \forall j \in \mathbb{N} \cdot 2 \leq j \Rightarrow s' \cap c_i &= \emptyset
\end{align*}
\]

Therefore

\[
\begin{align*}
  (\text{guar } s' \subseteq s \wedge s - s' \subseteq C) \Rightarrow \\
  s: [s \subseteq 2 \ldots n, \forall j \cdot 2 \leq j \Rightarrow s' \cap c_i &= \emptyset]
\end{align*}
\]

by above set theory

\[
\begin{align*}
  (\text{guar } s' \subseteq s \wedge s - s' \subseteq C) \Rightarrow \\
  s: [s \subseteq 2 \ldots n, \forall j \cdot 2 \leq j \Rightarrow s' \cap c_i &= \emptyset]
\end{align*}
\]

The refinement now focuses on just the specification (the second line)

Then some number theory

If $2 \leq i \wedge 2 \leq j$ and if $i \cdot j \leq n$ then either

\[ i^2 \leq n \wedge j^2 \geq n \]
\[ i^2 \leq n \wedge j^2 \geq n \]

Hence one only has to remove multiples of $i$ up to the (integer part of) the square root of $i$

\[
[s \subseteq 0 \ldots n \wedge n \leq i^2 \wedge (\forall j \in 2 \ldots i \cdot s \cap c_j = \emptyset) \Rightarrow (\forall j \in \mathbb{N} \cdot 2 \leq j \Rightarrow s \cap c_j = \emptyset)]
\]

The predicate $(\forall j \in 2 \ldots i \cdot s \cap c_j = \emptyset)$ holds if $i$ is 1

Refining the loop body

\[
i, s: [s \subseteq 2 \ldots n \wedge (i + 1)^2 \leq n \wedge i < i' \wedge \\
\forall j \in 2 \ldots i' \cdot s \cap c_j = \emptyset \vee \forall j \in 2 \ldots i \cdot s' \cap c_j = \emptyset]
\]

introduce sequential composition

\[
i := i + 1; \\
s[i \subseteq 2 \ldots n \wedge i^2 \leq n \wedge \\
\forall j \in 2 \ldots i - 1 \cdot s \cap c_j = \emptyset \vee \forall j \in 2 \ldots i \cdot s' \cap c_j = \emptyset]
\]

Refining the specification:

\[
s[i \subseteq 2 \ldots n \wedge i^2 \leq n \wedge \\
\forall j \in 2 \ldots i - 1 \cdot s \cap c_j = \emptyset \vee \forall j \in 2 \ldots i \cdot s' \cap c_j = \emptyset]
\]

to achieve the post condition the elements in $c_i$ need to be removed

\[
s[i \subseteq 2 \ldots n \wedge i^2 \leq n, s' \cap c_i = \emptyset]
\]

recall that $c_i$ contains all the multiples of $i$, excluding $i$

\[
s[i \subseteq 2 \ldots n \wedge i^2 \leq n, \forall j \cdot 2 \leq j \cdot i \leq n \Rightarrow j \cdot i \not\in s']
\]

Reminder: this is all in the context of $(\text{guar } s' \subseteq s \wedge s' \subseteq C)$
Introduce inner loop

Now recall that this was all in the context of a guarantee.

Require $s \subseteq 2 \ldots n \land 2 \cdot i \leq k \cdot i \leq n \Rightarrow k \cdot i \notin s$.

- introduce variable $k$ to be used as a loop index
  
  $$
  \begin{align*}
  \text{var } k &:= 2; \\
  s \subseteq 2 \ldots n \land 2 \cdot i \leq n & \land \quad n < k \cdot i \land \kappa \wedge \kappa' \wedge \\
  k, s &:= \{ s \subseteq 2 \ldots n \land 2 \cdot i \leq k \cdot i \leq n \land \quad \forall j \cdot 2 \cdot i \leq j \cdot i < k \cdot i \Rightarrow \\
  &\quad j \cdot i \notin s \} \quad j \cdot i \notin s'.
  \end{align*}
  $$

- introduce inner loop
  
  $$
  \begin{align*}
  \text{while } k \cdot i \leq n \text{ do} \\
  k, s &:= \{ s \subseteq 2 \ldots n \land 2 \cdot i \leq k \cdot i \leq n \land \quad \forall j \cdot 2 \cdot i \leq j \cdot i < k \cdot i \Rightarrow \\
  &\quad j \cdot i \notin s \} \quad j \cdot i \notin s'.
  \end{align*}
  $$

- refine the inner loop body
  
  Define
  
  $$
  \begin{align*}
  s &\subseteq 2 \ldots n \land 2 \cdot i \leq k \cdot i \leq n \land \quad k \cdot i \leq k' \wedge \\
  k, s &:= \{ s \subseteq 2 \ldots n \land 2 \cdot i \leq k \cdot i \leq n \land \quad \forall j \cdot 2 \cdot i \leq j \cdot i < k \cdot i \Rightarrow \\
  &\quad j \cdot i \notin s \} \quad j \cdot i \notin s'.
  \end{align*}
  $$

- introduce sequential composition
  
  $$
  \begin{align*}
  s &\subseteq 2 \ldots n \land 2 \cdot i \leq k \cdot i \leq n \land \\
  k := k + 1
  \end{align*}
  $$

Now refine the specification

Require $s \subseteq 2 \ldots n \land 2 \cdot i \leq k \cdot i \leq n \land \kappa \wedge \kappa' \wedge \\
\quad s \subseteq 2 \ldots n \land 2 \cdot i \leq k \cdot i \leq n \land \forall j \cdot 2 \cdot i \leq j \cdot i < k \cdot i \Rightarrow \\
\quad j \cdot i \notin s \} \quad j \cdot i \notin s'.

- to achieve the post condition the element $k \cdot i$ must be removed
  
  $$
  \begin{align*}
  s &\subseteq 2 \ldots n \land 2 \cdot i \leq k \cdot i \leq n \land \\
  \text{Rem}(m) \equiv \{ s \subseteq s \land s - s' \subseteq \{ m \} \} \land s \subseteq 2 \ldots n \land m \in 0 \ldots n, m \notin s'.
  \end{align*}
  $$

Define

$$
\begin{align*}
\text{Rem}(m) &\equiv \{ s \subseteq s \land s - s' \subseteq \{ m \} \} \land s \subseteq 2 \ldots n \land m \in 0 \ldots n, m \notin s'.
\end{align*}
$$

The code so far is

Require $s \subseteq 2 \ldots n \land 2 \cdot i \leq k \cdot i \leq n \land \kappa \wedge \kappa' \wedge \\
\quad s \subseteq 2 \ldots n \land 2 \cdot i \leq k \cdot i \leq n \land \forall j \cdot 2 \cdot i \leq j \cdot i < k \cdot i \Rightarrow \\
\quad j \cdot i \notin s \} \quad j \cdot i \notin s'.

- var $i := 1$;
  
  $$
  \begin{align*}
  \text{while } (i + 1)^2 \leq n \text{ do} \\
  i &:= i + 1;
  \end{align*}
  $$

- var $k := 2$;
  
  $$
  \begin{align*}
  \text{while } k \cdot i \leq n \text{ do} \\
  \text{Rem}(k \cdot i); \\
  k &:= k + 1
  \end{align*}
  $$

Remove an element from the set

Now recall that this was all in the context of a guarantee.

Require $s \subseteq s \land s - s' \subseteq \{ m \} \land s \subseteq 2 \ldots n \land m \in 0 \ldots n, m \notin s'$.
Data refinement: representing the set as an array of words

- A finite set contained in $0 \ldots n$ can be represented by a bit map of $n+1$ bits
- Assume a word has $ws$ bits
- A word can represent a set with $ws$ elements
- A word can represent a set contained in the set $0 \ldots ws-1$
- For a large set one needs a vector $v$ of $\left\lceil \frac{n+1}{ws} \right\rceil$ words

The function $\text{retr}(v)$ retrieves the set represented by $v$

$$\text{retr}(v) \equiv \{ j \in 0 \ldots n \mid (j \mod ws) \in v(j \div ws) \}$$

Remove an element from the set

Define

$$\text{Rem}(m) \equiv (\text{guar } s' \subseteq s \land s - s' \subseteq \{m\}) \land s: [s \subseteq 0 \ldots n \land m \in 0 \ldots n, m \notin s']$$

Using the representation as an array $v: \text{array } 0 \ldots \left\lceil \frac{n+1}{ws} \right\rceil - 1$ of $(0 \ldots ws-1)$

$$(\text{guar } \text{retr}(v') \subseteq \text{retr}(v) \land \text{retr}(v) - \text{retr}(v') \subseteq \{m\}) \land v: [\text{retr}(v) \subseteq 0 \ldots n \land m \in 0 \ldots n, m \notin \text{retr}(v')]$$

From the definition of $\text{retr}$

$$m \notin \text{retr}(v') \iff (m \mod ws) \notin v'(m \div ws)$$

Hence the specification can be written as

$$\begin{align*}
\forall v: [\text{retr}(v) \subseteq 0 \ldots n \land m \in 0 \ldots n, (m \mod ws) \notin v'(m \div ws)] \\
\equiv \forall v(m \div ws): [m \in 0 \ldots n, (m \mod ws) \notin v'(m \div ws)]
\end{align*}$$

Removing an element from a set represented as a single word

$$\text{RemW}(\text{var } w: \mathbb{F}(0 \ldots ws - 1), i: 0 \ldots ws - 1) \equiv$$

$$(\text{guar } w' \subseteq w \land w - w' \subseteq \{i\}) \land w: [w \subseteq 0 \ldots ws - 1 \land i \in 0 \ldots ws - 1, i \notin w']$$

Therefore

$$\text{Rem}(m) \subseteq \text{RemW}(v(m \div ws), m \mod ws)$$

$\text{RemW}$ can be implemented using bit-wise operations on a word (exercise)

Conclusions

- Importance of data abstraction
- Guarantee allows one to focus on the interesting part
Example: Parallel SIEVE of Eratosthenes

- Determine primes up to some given \( n \)
- Illustrates:
  - starting with abstract type
  - need to document interference (R)
  - interplay between G/Q
  - development to code (using CAS)
  - symmetric processes (identical R/G)

Intuition

- data abstraction: shared set of \( \mathbb{N}_1 \)
- initialize: all (positive) natural numbers from 2 up to \( n \)
- remove all composites
- for sequential for \( i = 2 \cdots \) post condition of each \( \text{RemMults}(i) \) iteration is easy
  \[ \text{RemMults}(i) \triangleq s : |s' = s - c_i| \]
- for Sieve \( \triangleleft || \text{RemMults}(i) \)
  - need the rely of \( \text{RemMults}(i) \) to be \( s' \subseteq s \)
  - relax the equality in the postcondition of \( \text{RemMults}(i) \) to \( s' \cap c_i = \emptyset \)
  - avoid removing too much with a guarantee of \( \text{RemMults}(i) \) of \( s - s' \subseteq c_i \)
  - because processes are identical, have to add a guarantee of no reinsertion
Rely/Guarantee (R/G) idea is simple

face interference (in specifications and design process)

\[
\begin{align*}
\sigma_0 & \cdots \sigma_i \sigma_{i+1} \cdots \pi_i \sigma_{j+1} \cdots \sigma_f \\
\text{pre} & \quad \cdots \quad \text{rely} & \cdots \quad \text{guar} & \\
\text{post} &
\end{align*}
\]

- assumptions pre/rely
- commitments guar/post

- rely conditions an abstraction of interference to be tolerated
relations are key to R/G

Interference between processes

An example of interference on process \(P\) by process \(Q\)

- One shared variable \(j\)
- process \(Q\) may do atomic steps that either
  - do not change \(j\), i.e. \(j' = j\), or
  - increment \(j\) by one, i.e. \(j' = j + 1\)
- before or after each atomic step of process \(P\), it may observe
  - no steps of \(Q\), i.e. \(j' = j\)
  - one step of \(Q\), i.e. \(j' = j \lor j' = j + 1\)
  - many steps of \(Q\), i.e. \(j \leq j'\)
- Observing that both \(j' = j\) and \(j' = j + 1\) imply \(j \leq j'\)
- Hence we can use \(j \leq j'\) to represent the possible interference from \(Q\) on \(P\)

This abstract view of the interference becomes

- a rely condition of \(P\)
- a guarantee condition of \(Q\)

R/G rethought

"pulling apart" old R/G notation — literally!

\[
\begin{align*}
\text{RemMults}(i) \\
\text{ext wr } s : \mathbb{F} N_i \\
\text{pre } s \subseteq 0 \ldots n \\
\text{rely } s' \subseteq s \\
\text{guar } s' \subseteq s \land \cdots \\
\text{post } s' = s - c_i
\end{align*}
\]

Proof rules (also used a 5-tuple form)

\[
\begin{align*}
\{P, R_i\} & s_i \{G_i, Q_i\} \\
\{P, R_i\} & s_r \{G_r, Q_r\} \\
R \lor G_r & \Rightarrow R_i \\
R \lor G_i & \Rightarrow R_r \\
G_i \lor G_r & \Rightarrow G \\
P \land Q_i \land Q_r \land (R \lor G_i \lor G_r)^* & \Rightarrow Q \\
\{P, R\} & s_l \{G, Q\}
\end{align*}
\]

R/G rethought

R/G decomposed [?, ?]

\[
\begin{align*}
\{s \subseteq 0 \ldots n\} \\
\text{guar}(s' \subseteq s \land \cdots) \bullet \\
\text{rely } s' \subseteq s \bullet \\
s : [s' = s - c_i]
\end{align*}
\]

Now [?]

\[
\begin{align*}
\text{RemMults}(i : N) \\
\{s \subseteq 0 \ldots n\} \\
(\text{rely } s' \subseteq s) \land \\
(\text{guar } s' \subseteq s \land s - s' \subseteq c_i) \land \\
s : [s' \cap c_i = \emptyset]
\end{align*}
\]
R/G rethought

Advantage of the new style: brings out (algebraic) properties

Distribute-G-seq
\[(\text{guar} g) \sqcap (c ; d) = ((\text{guar} g) \sqcap c) ; ((\text{guar} g) \sqcap d)\]

Distribute-G-par
\[(\text{guar} g) \sqcap (c \parallel d) = ((\text{guar} g) \sqcap c) \parallel ((\text{guar} g) \sqcap d)\]

Conjunction-mono
\[c_0 \sqsubseteq c_1 \land d_0 \sqsubseteq d_1 \Rightarrow c_0 \sqcap d_0 \sqsubseteq c_1 \sqcap d_1\]

Conjoin-G:
\[(\text{guar} g_1) \sqcap (\text{guar} g_2) = (\text{guar} g_1 \land g_2)\]

Strengthen-G:
\[(\text{guar} g_1) \sqsubseteq (\text{guar} g_2)\]
\[\text{if } g_2 \Rightarrow g_1\]

Distribute-G:
\[((\text{guar} g) \sqcap || c_i) = || (\text{guar} g) \sqcap c_i\]

Trading rely, guarantee and post

Trading-R-G-Post:
\[(\text{rely} r) \sqcap [(r \lor g)^* \land q] \sqsubseteq (\text{rely} r) \sqcap (\text{guar} g) \sqcap [q]\]

Intro-multi-Par:
\[(\text{rely} r) \sqcap [\land q_i] \sqsubseteq \llbracket (\text{guar} \rho) \sqcap (\text{rely} \rho) \sqcap [q_i] \rrbracket\]
\[\text{if } r \Rightarrow \rho\]

(Some) Laws of the new algebraic R/G

... a few of many!
Refinement calculus style development

$s$ initially contains set of natural numbers from 2 up to some $n$

$C$ is the set of all composite numbers

$(\text{rely } s' = s) \land (s' = s - C)$

set theory

$(\text{rely } s' = s) \land (s' \subseteq s \land s' \subseteq C \land s' \cap C = \emptyset)$

by Trading-R-G-Post as $s' \subseteq s \land s' \subseteq C$ is reflexive and transitive

$(\text{guar } s' \subseteq s \land s' \subseteq C) \land (\text{rely } s' = s) \land (s' \cap C = \emptyset)$

as $s' \cap C = \emptyset \equiv s' \cap \bigcup_i (s' \cap c_i) = \emptyset \equiv \bigcup_i (s' \cap c_i) = \emptyset \forall i. s' \cap c_i = \emptyset$

$(\text{guar } s' \subseteq s \land s' \subseteq C) \land (\text{rely } s' = s) \land (s' \cap C = \emptyset)$

by Intro-multi-Par

$(\text{guar } s' \subseteq s \land s' \subseteq C) \land (\text{rely } s' = s) \land (s' \cap C = \emptyset)$

Distribute-G and Conjoin-G

$f(f(s)) = f(s)$

as

$\text{Intro-multi-Par}$

$\text{Distribute-G}$ and $\text{Conjoin-G}$

$(\text{guar } s' \subseteq s \land s' \subseteq C) \land (\text{rely } s' = s) \land s: [s' \cap c_i = \emptyset]$ $\land$ $\text{Strengthen-G}$

$\bigcup((\text{guar } s' \subseteq s) \land (\text{rely } s' \subseteq s) \land s: [s' \cap c_i = \emptyset])$

Rem-W which removes an element from set (as a word)

$(\text{guar } s' \subseteq s \land s' \subseteq \{m\}) \land (\text{rely } s' \subseteq s) \land s: [m / s']$

The set $c_i$ contains all the multiples of $i$ (except $i \cdot 1$)

$\text{RemMults}(i : \mathbb{N})$

$(s \subseteq 0..n)$

$(\text{guar } s' \subseteq s \land s' \subseteq c_i) \land (\text{rely } s' \subseteq s) \land s: [s' \cap c_i = \emptyset]$

Onwards to code

The interesting part is $\text{Rem}$. Its specification allows interference that removes elements from $s$. It guarantees to remove element $m$, only.

$\text{Rem}(m : \mathbb{N})$

$(s \subseteq 0 \ldots n \land m \in 0 \ldots n)$

$(\text{guar } s' \subseteq s \land s' \subseteq \{m\}) \land (\text{rely } s' \subseteq s) \land s: [m / s']$

Removing an element from a set atomically

The specification of $\text{Rem}$ allows interference that removes elements from $s$. It guarantees to remove element $m$, only.

$\text{Rem}(m : \mathbb{N})$

$(s \subseteq 0 \ldots n \land m \in 0 \ldots n)$

$(\text{guar } s' \subseteq s \land s' \subseteq \{m\}) \land (\text{rely } s' \subseteq s) \land s: [m / s']$

Represent the set $s$ as an array $v$ of words each representing part of the set

$v : \text{array } 0..[\frac{n+1}{ws}] - 1 \text{ of } \mathbb{F}(0..ws - 1)$

Representation relation

$\text{retr}(v) = \{j \in 0 \ldots n \mid j \text{ mod } ws \in v(j \text{ div } ws)\}$

Implementation using $\text{RemW}$ which removes an element from set (as a word)

$\text{RemW}(v(m \text{ div } ws), m \text{ mod } ws)$

Specification

$\text{RemW}(\text{var } w : \mathbb{F}(0..ws - 1), i : 0..ws - 1)$

$(\text{guar } w' \subseteq w \land w - w' \subseteq \{i\}) \land (\text{rely } w' \subseteq w) \land w: [i / w']$

Compare and swap

The implementation without locks makes use of a compare-and-swap (CAS)

$\text{CAS}(\text{var } w, lw, nw, \text{var } done) \equiv$

$(\text{rely } lw' = lw \land nw' = nw \land done' = done) \land$

$w, done : \left< (w = lw \Rightarrow w' = nw \land done') \land (w \neq lw \Rightarrow w' = w \land \neg done') \right>$

Under rely condition $w' \subseteq w$ assuming $lw$, $nw$ and $done$ are local

$w, done : \left< (w \subseteq lw \land nw = lw - \{i\}) \land (w \neq lw \Rightarrow w' \subseteq lw) \right> \land$

$\text{CAS}(w, lw, nw, \_);$

Note that the first parameter is a var parameter, i.e. call-by-reference
Refining the second specification

\[
\{ w \subseteq 0..ws - 1 \land i \in 0..ws - 1 \} \\
\begin{align*}
\text{(gaur } w' \subseteq w \land w - w' \subseteq \{ i \}) & \land (\text{rely } w' \subseteq w) \land w : [ i \not\in w'] \\
\text{invariant true and variant } w \supset w'
\end{align*}
\]

while \( i \in w \) do

\[
\begin{align*}
\text{(gaur } w' \subseteq w \land w - w' \subseteq \{ i \}) & \land (\text{rely } w' \subseteq w) \land w : [ i \not\in w'] \\
\text{Refine loop body}
\end{align*}
\]

\[
\begin{align*}
\text{(gaur } w' \subseteq w \land w - w' \subseteq \{ i \}) & \land (\text{rely } w' \subseteq w) \land w : [ w \supset w' \lor i \not\in w'] \\
\text{strengthen guarantee, introduce local variable } lw \\
\text{var } lw : \\
\begin{align*}
\text{(gaur } w' = w \lor w' = w - \{ i \}) & \land (\text{rely } w \supseteq w') \land \\
(lw : [ w \supseteq lw \supseteq w'] : w : lw \supseteq w, lw \supset w' \lor i \not\in w') \\
\end{align*}
\]Refining the first specification

\[
\begin{align*}
\text{(gaur } w' = w \lor w' = w - \{ i \}) & \land (\text{rely } w' \supseteq w') \land lw : [ lw \supseteq lw' \supseteq w'] \\
\text{lw := w}
\end{align*}
\]

Introducing a Compare-and-Swap (CAS)

\[
\begin{align*}
\text{CAS}(w, lw, nw, \text{var } done) & \equiv \\
\text{(rely } lw' = lw \land nw' = nw \land done' = done) & \land \\
w, done : \\
\begin{align*}
\{ w = lw \Rightarrow w' = nw \land done' = done \} & \land \\
\{ w \neq lw \Rightarrow w' = w \land \neg done' \} & \}
\end{align*}
\]

The variables \( lw \) and \( nw \) are local so the rely is satisfied; \( done \) isn't used

\[
\begin{align*}
\text{(gaur } w' = w \lor w' = w - \{ i \}) & \land (\text{rely } w \supseteq w') \land \\
w : [ lw \supseteq w \land nw = lw - \{ i \} , lw \supset w' \lor i \not\in w'] \\
\text{CAS}(w, lw, nw, \_)
\end{align*}
\]
Removing an element from a (small) set atomically

Specification

\[ \text{RemW}( \text{var } w : F(0..ws - 1), i : 0..ws - 1) \]
\[ (\text{guar } w' \subseteq w \land w - w' \subseteq \{i\}) \land (\text{rely } w' \subseteq w) \land w : [i \notin w'] \]

Code

\[ \text{while } i \in w \text{ do invariant } \text{true} \]
\[ \text{var } lw := w; \]
\[ \text{var } nw := lw - \{i\}; \quad \text{– stable because variables local} \]
\[ \text{CAS}(w, lw, nw, _); \quad \text{– refines } w : \begin{cases} lw \subseteq w \land nw = lw - \{i\}, \\ (w = lw \Rightarrow w' \subseteq w - \{i\}) \land \\ (w \neq lw \Rightarrow w' \subset lw) \end{cases} \]
\[ \{i \notin w\} \]

Termination

\[ \text{while } i \in w \text{ do invariant true wf-relation } (w' \subset w) \text{ OR } (\#w' < \#w) \]
\[ \text{var } lw := w; \]
\[ \text{var } nw := lw - \{i\}; \quad \text{– stable because variables local} \]
\[ \text{CAS}(w, lw, nw, _); \quad \text{– refines } w : \begin{cases} lw \subseteq w \land nw = lw - \{i\}, \\ (w = lw \Rightarrow w' \subseteq w - \{i\}) \land \\ (w \neq lw \Rightarrow w' \subset lw) \end{cases} \]
\[ \{i \notin w\} \]

Conclusions

- Rely/guarantee provides a simple but effective abstraction of concurrency
- Importance of data abstraction
- New algebraic style makes proving new laws simpler
- Interesting links/similarities to process algebras (SCCS)
- New style allows new forms of specifications
The With and Await statements

The `with x do c` statement ensures that the updates of `x` are atomic. There is no interference on `x` during the update.

```
with x do c ⇔ idle; ((demand x' = x) ⊓ c); idle
with x do c ⇔ (|x|); ((demand x' = x) ⊓ c); idle
```

This allows id steps forever, even when `x` isn’t in use elsewhere.

The `await` statement delays until its condition evaluates to true. It may fail by evaluating to false any number of times.

```
await b ⇔ [¬ b]; [b]
```

where `[[b]]` succeeds if and only if `b` evaluates to true. Equivalent to

```
await b = while ¬ b do nil
```

Doing nothing under interference

The command `idle` only makes a finite number of program steps that do not change the environment

If `r` maintains `p`, i.e. `r ⇒ (p ⇒ p')`, then

```
(rely r) ⊓ [p, r* ∧ p'] ⊑ idle
```

For example, the rely condition `(buf' suffix buf)` maintains `#buf < N`, and hence

```
(rely r) ⊓ [#buf < N, buf' suffix buf ∧ #buf' < N] ⊑ idle
```

Similarly, if `r` maintains `p`, and `r` maintains `b`,

```
(rely r) ⊓ [p, r* ∧ p' ∧ b'] ⊑ await b
```

Multi-place buffer of size N

```
module Buffer
var buf : seq Value
invariant #buf ≤ N
initially buf = []

write(v : Value)
rely buf' suffix buf ⊓ – single writer
guar buf prefix buf' ⊓
with buf await #buf < N do
  buf : buf' = buf ∧ [v]

read():res : Value
rely buf prefix buf' ⊓ – single reader
guar buf' suffix buf' ⊓
with buf await #buf ≠ 0 do
  res, buf : [buf = [res] ∧ buf']
```
Multi-place buffer implementation

The buffer $b$ has $N+1$ slots but one is always unused. We define the notation $a \oplus b = (a + b) \mod (N + 1)$. The slots start at $r$ and $w$ is the index of the next slot to be written, so that

- if $r = w$ the buffer is empty and
- if $r = w \oplus 1$ the buffer is full.

The retrieve function is defined by

$$\text{retr}(b, r, w) = \begin{cases} [\ ] & \text{if } r = w \\ [b[r]] \uplus \text{retr}(b, r \oplus 1, w) & \text{otherwise} \end{cases}$$

The implementation of the buffer is given by

module BufferI implements Buffer
var b : (0 .. N) -> Value;
  r, w : 0 .. N;
initially r = 0 \land w = 0;
representation buf = retr(b, r, w)

Write in a circular buffer

write(v : Value)
rely buf' suffix buf \includes – single writer
guar buf prefix buf' \includes

with buf await #buf < N do
  buf : [buf' = buf \uplus [v]]

rely buf' suffix buf \includes
guar buf prefix buf' \includes

with buf do
  buf : [buf < N, buf' = buf \uplus [v]]

Initial refinement of write

write(v : Value)
rely buf' suffix buf \includes – single writer
guar buf prefix buf' \includes
with buf await #buf < N do
  buf : [buf' = buf \uplus [v]]

rely buf' suffix buf \includes
guar buf prefix buf' \includes

await #buf < N; -- await buffer not full – stable under rely
with buf do
  buf : [#buf < N, buf' = buf \uplus [v]]

Initial refinement of read

read() res : Value
rely buf prefix buf' \includes – single reader
guar buf' suffix buf \includes
with buf await #buf \neq 0 do
  res, buf : [buf = [res] \uplus buf']

rely buf prefix buf' \includes
guar buf' suffix buf \includes
with buf do
  buf : [#buf \neq 0, buf' = tl(buf)]
Read in circular buffer

```plaintext
read() \text{res} : \text{Value}
rely \text{buf} prefix \text{buf}' \in\
\text{guar} \text{buf}' suffix \text{buf} \in\n\text{await} \#\text{buf} \neq 0; \quad \text{– await buffer not empty – stable under rely}
\text{res} : [\text{res}' = \text{hd}(\text{buf})]
\text{with do do} : \left[\#\text{buf} \neq 0, \text{buf}' = \text{tl}(\text{buf})\right] \quad \text{– atomic update of \text{buf}}
```

is data refined by

```plaintext
\text{rely} \ r' = r \land (r = w \odot 1 \Rightarrow w' = w) \land \text{retr}(\text{b}, r, w) \text{prefix} \text{retr}(\text{b}', r', w') \in\n\text{guar} w' = w \land b' = b \land (r = w \Rightarrow r' = r) \in\n\text{await} r \neq \langle w \rangle; \quad \text{– await buffer non-empty – stable under rely}
\text{res} ::= b[r];
\text{var nr} ::= r \odot 1;
\quad \text{– Ensure \text{b}[r] has been fully read before updating \text{r}}
with r do r := nr \quad \text{– atomic update of \text{r}}
```

Write in a circular buffer with size

```plaintext
write(\text{v} : \text{Value})
\text{rely \text{buf}'} suffix \text{buf} \in\n\text{await} \#\text{buf} < N; \quad \text{– stable under rely}
\text{with do do} : \left[\#\text{buf} < N, \text{buf}' = \text{buf} \mapsto [\text{v}]\right]
```

is data refined using representation \text{buf}' = \text{retr}_w(\text{b}, w, s) by

```plaintext
\text{rely} w' = w \land b' = b \land 0 \leq s' \leq s \in\n\text{guar} r' = r \land s \leq s' \leq N \land \text{retr}_r(\text{b}, r, s) \text{prefix} \text{retr}_r(\text{b}', r', s') \in\n\text{await}(s) < N; \quad \text{– await buffer not full – stable under rely}
\text{b}[w] ::= \text{v};
\quad \text{– Ensure \text{b}[w] is flushed before updating \text{s}}
(\text{w} := (\text{w} + 1) \mod N \parallel \text{with do s} ::= s + 1) \quad \text{– atomic update of \text{s}}
```

Note that the representation relation is broken during the last parallel assignment but restored on completion of both assignments. Contention on update of \text{s} via a compare-and-swap bounded by reader decreasing size to 0.

Multi-place buffer implementation with size

The buffer \text{b} has \text{N} slots and keeps a separate variable \text{s} to track its current size. The slots start at \text{r} and \text{w} is the index of the next slot to be written, so that

- if \text{s} = 0 the buffer is empty and
- if \text{s} = \text{N} the buffer is full.

We define two retrieve functions, one for read and one for write. I have no idea what the theory is but the write and write processes have different views of the buffer.

```plaintext
\text{retr}_r(\text{b}, r, s) = (\lambda i \in 0..s - 1 \cdot b[(r + i) \mod \text{N}])
\text{retr}_w(\text{b}, w, s) = (\lambda i \in 0..s - 1 \cdot b[(w + i + n - s) \mod \text{N}])
```

```plaintext
\text{module BufferI implements Buffer}
\text{var b : (0..\text{N} - 1) \rightarrow Value};
\text{r, w : 0..\text{N} - 1};
\text{s : 0..\text{N}};
\text{initially s = 0 \land r = 0 \land w = 0};
\text{representation buf' = retr_r(\text{b}, r, s) = retr_w(\text{b}, w, s)}
```

Read in circular buffer with size

```plaintext
read() \text{res} : \text{Value}
\text{rely \text{buf} prefix \text{buf}' \in\n\text{guar} \text{buf}' suffix \text{buf} \in\n\text{await} \#\text{buf} \neq 0; \quad \text{– stable under rely}
\text{res} : [\text{res}' = \text{hd}(\text{buf})]
\text{with do do} : \left[\#\text{buf} \neq 0, \text{buf}' = \text{tl}(\text{buf})\right] \quad \text{– atomic update of \text{buf}}
```

is data refined using representation \text{buf}' = \text{retr}_r(\text{b}, r, s) by

```plaintext
\text{rely} r' = r \land s \leq s' \leq N \land \text{retr}_r(\text{b}, r, s) \text{prefix} \text{retr}_r(\text{b}', r', s') \in\n\text{guar} w' = w \land b' = b \land 0 \leq s' \leq s \in\n\text{await}(s) \neq 0; \quad \text{– await buffer non-empty – stable under rely}
\text{res} ::= b[r];
\quad \text{– Ensure \text{b}[r] has been fully read before updating \text{s or r}}
(r := (r + 1) \mod \text{N} \parallel \text{with do s} ::= s - 1) \quad \text{– atomic update of \text{s}}
```

Note that the representation relation is broken during the last parallel assignment but restored on completion of both assignments. Contention on update of \text{s} via a compare-and-swap bounded by reader increasing size to \text{N}. 57/1 58/1 59/1 60/1
Find least first element of an array that satisfies \( P \)

The objective is, given an array \( v \) with indices in the range \( 0 \ldots N - 1 \), to find the least index \( t \) for which a predicate \( P(v(t)) \) holds,\(^1\) or if \( P \) does not hold for any element of \( v \), to set \( t \) to \( N \).

\[
\text{findp} \triangleq t : 
\begin{cases} 
(t' = N \lor \text{satp}(v, t')) \land \text{notp}(v, 0 \ldots N - 1, t') 
\end{cases}
\]

where

\[
\begin{align*}
\text{satp}(v, t) & \triangleq t \in 0 \ldots N - 1 \land P(v(t)) \\
\text{notp}(v, s, t) & \triangleq (\forall i \in s \cdot i < t \implies \neg P(v(i)))
\end{align*}
\]

\(^1\)For brevity, it is assumed here that \( P(x) \) is always defined (undefinedness is considered by \([?)\] but it has little bearing on the actual design).

Representing the result using two variables

Two variables \( \text{ot} \) and \( \text{et} \) are introduced with the intention that on termination the minimum of \( \text{ot} \) and \( \text{et} \) will be the least index satisfying \( p \).

\[
\begin{align*}
(\text{rely } v' = v \land t' = t) \land t : [ (t' = N \lor \text{satp}(v, t')) \land \text{notp}(v, 0 \ldots N - 1, t') ] \quad \triangleright
\end{align*}
\]

by Law variable-rely-guarantee for \( \text{ot} \) and \( \text{et} \)

\[
\begin{align*}
\text{var } \text{ot}, \text{et} : 
(\text{rely } v' = v \land t' = t \land \text{ot'} = \text{ot} \land \text{et'} = \text{et}) \land \\
\text{ot}, \text{et}, t : [ (\text{min}(\text{ot'}, \text{et'}) = N \lor \text{satp}(v, \text{min}(\text{ot'}, \text{et'}))) \land \text{notp}(v, 0 \ldots N - 1, \text{min}(\text{ot'}, \text{et'})) ] \quad \triangleright
\end{align*}
\]

\[
t := \text{min}(\text{ot'}, \text{et'})
\]

Using a guarantee invariant

A guarantee invariant is a guarantee that states a predicate \( p \) is invariant.

\[
(\text{guar-inv } p) \triangleq (\text{guar } p \implies p')
\]

A guarantee invariant of

\[
\text{min}(\text{ot}, \text{et}) = N \lor \text{satp}(v, \text{min}(\text{ot}, \text{et}))
\]

(1)

can be employed; the invariant is established by setting both \( \text{ot} \) and \( \text{et} \) to \( N \).

\[
\begin{align*}
(\text{rely } v' = v \land \text{ot'} = \text{ot} \land \text{et'} = \text{et}) \land \\
\text{ot}, \text{et} : [ (\text{min}(\text{ot'}, \text{et'}) = N \lor \text{satp}(v, \text{min}(\text{ot'}, \text{et'}))) \land \text{notp}(v, 0 \ldots N - 1, \text{min}(\text{ot'}, \text{et'})) ] \quad \triangleright
\end{align*}
\]

by Law trade-rely-guarantee-invariant; Law rely-sequential

\[
\begin{align*}
\text{ot} := N; \text{et} := N; \\
( (\text{guar-inv } \text{min}(\text{ot}, \text{et}) = N \lor \text{satp}(v, \text{min}(\text{ot}, \text{et}))) \land \\
(\text{rely } v' = v \land \text{ot'} = \text{ot} \land \text{et'} = \text{et}) ) \land \\
\text{ot}, \text{et} : [ \text{notp}(v, 0 \ldots N - 1, \text{min}(\text{ot'}, \text{et'})) ] \quad \triangleright
\end{align*}
\]
The motivation for the parallel algorithm comes from the observation that the set of indices to be searched, \(0 \ldots N - 1\), can be partitioned into the odd and even indices, namely \(\text{evens}(N)\) and \(\text{odds}(N)\), respectively, which can be searched in parallel.

\[
\notp(v, \text{odds}(N), \min(ot', et')) \land \notp(v, \text{evens}(N), \min(ot', et')) \Rightarrow
\notp(v, 0 \ldots N - 1, \min(ot', et'))
\]

The next step is the epitome of rely-guarantee refinement: splitting the specification command.

\[
\text{(rely } v' = v \land ot' = ot \land et' = et) \land
\text{ot, et: } \notp(v, 0 \ldots N - 1, \min(ot', et')) \downarrow
\]

\[
\text{by Law introduce-parallel-spec-weaken-rely}
\]

\[
(\text{guar } ot' \leq ot \land et' = et) \land (\text{rely } et' \leq et \land ot' = ot \land v' = v) \land
\text{ot, et: } [\notp(v, \text{odds}(N), \min(ot', et'))] \downarrow
\]

\[
(\text{guar } et' \leq et \land ot' = ot) \land (\text{rely } ot' \leq ot \land et' = et \land v' = v) \land
\text{ot, et: } [\notp(v, \text{evens}(N), \min(ot', et'))] \downarrow
\]

For the first branch of the parallel, the guarantee \(et' = et\) is equivalent to removing \(et\) from the frame of the branch.

\[
(\text{guar } ot' \leq ot \land et' = et) \land (\text{rely } et' \leq et \land ot' = ot \land v' = v) \land
\text{ot, et: } [\notp(v, \text{odds}(N), \min(ot', et'))] =
\]

\[
(\text{guar } ot' \leq ot) \land (\text{rely } et' \leq et \land ot' = ot \land v' = v) \land
\text{ot: } [\notp(v, \text{evens}(N), \min(ot', et'))] \downarrow
\]

The body of this can be refined to sequential code, however, because the specification refers to \(et'\) it is subject to interference from the parallel (evens) process which may update \(et\). That interference is however bounded by the rely condition which assumes the parallel process never increases \(et\).

The guarantee invariant combined with the postcondition \(oc' \geq \min(ot', et')\) implies the postcondition of the above specification. The postcondition \(oc' \geq \min(ot', et')\) uses \(\geq\) rather than \(=\) because the parallel process may decrease \(et\).

\[
(\text{rely } et' \leq et \land oc' = oc \land ot' = ot \land v' = v) \land
\text{oc, ot: } [\notp(v, \text{odds}(N), \min(ot', et'))] \downarrow
\]

\[
\text{by Law variable-rely-guarantee for oc}
\]

\[
\var{oc}::
\]

\[
(\text{rely } et' \leq et \land oc' = oc \land ot' = ot \land v' = v) \land
\text{oc, ot: } [\notp(v, \text{odds}(N), \min(ot', et'))] \downarrow
\]

\[
\text{At this point a guarantee invariant}
\]

\[
\notp(v, \text{odds}(N), oc) \land \text{bnd}(oc, N)
\]

(2)

is introduced where the bounding conditions on \(oc\) follow.

\[
\text{bnd}(oc, N) \equiv 1 \leq oc \leq N + 1
\]

This guarantee invariant is established by setting \(oc\) to one.
Law for introducing a while loop

Given

- a loop invariant $p$ that is a state predicate
- a rely condition $r$ that is a reflexive, transitive relation on states
- a variant function $v$ of type $T$ and a binary relation $\_ \triangleright \_ \text{ on } T$
- a boolean expression $b$ and predicates $b_0$ and $b_1$

if

- $p$ is $r$-stable, i.e. $r \Rightarrow (p \Rightarrow p')$
- $\_ \triangleright \_ \text{ is well-founded on } p$, i.e. $p \triangleleft (\_ \triangleright \_ \text{ is well-founded}$
- $v$ is non-increasing under $r$ on $p$, i.e. $p \wedge r \Rightarrow v' \leq v$
- $b$ is single reference, i.e. it has only a single reference to a non-stable variable
- $p \wedge b \Rightarrow b_0$ and $p \wedge r \Rightarrow (b_0 \Rightarrow b'_0)$
- $p \wedge \neg b \Rightarrow b_1$ and $p \wedge r \Rightarrow (b_1 \Rightarrow b'_1)$

then

\[
\text{(rely } r\text{) } \land [p, p' \wedge b_1 \wedge v' \leq v]
\]

\begin{itemize}
  \item while $b$ do
  \begin{itemize}
    \item (rely $r\text{) } \land [p \wedge b_0, p' \wedge v' < v]
  \end{itemize}
\end{itemize}

The specification of the loop body only involves variables which are stable under interference.

\[
\text{(rely } et' \leq et \wedge oc' = oc \wedge ot' = ot \wedge v' = v) \land \text{ }
\]

\[
\text{oc}, \text{ot: } [oc < ot, -1 \leq ot' - oc' < ot - oc]
\]

\begin{itemize}
  \item by Law weaken-rely
  \begin{itemize}
    \item (rely $oc' = oc \wedge ot' = ot \wedge v' = v) \land \text{ }
    \text{oc}, \text{ot: } [oc < ot, -1 \leq ot' - oc' < ot - oc]
  \end{itemize}
\end{itemize}

A while loop is introduced using Law rely-loop. Only the first conjunct of the loop guard $oc < ot \wedge oc < et$ is preserved by the rely condition because $et$ may be decreased. Hence the boolean expression $b_0$ for this application of the law is $oc < ot$. However, the loop termination condition $oc \geq ot \lor oc \geq et$ is preserved by the rely condition as decreasing $et$ will not falsify it. Hence $b_1$ is $oc \geq ot \lor oc \geq et$, which ensures $oc \geq min(ot, et)$ as required. For loop termination a well-founded relation reducing the variant $ot - oc$ is used.

\[
\text{(rely } et' \leq et \wedge oc' = oc \wedge et' = et \wedge v' = v) \land \text{ }
\]

\[
\text{oc}, \text{ot: } [oc' \geq min(ot', et')]
\]

\begin{itemize}
  \item by Law rely-loop
  \begin{itemize}
    \item while $oc < ot \land oc < et$ do
    \begin{itemize}
      \item (rely $et' \leq et \wedge oc' = oc \wedge et' = et \wedge v' = v) \land \text{ }
        \text{oc}, \text{ot: } [oc < ot, -1 \leq ot' - oc' < ot - oc]
    \end{itemize}
  \end{itemize}
\end{itemize}

At this stage we bring back in the guarantee invariants introduced above.

The refinement is now uses Law rely-conditional.

\[
\text{(guar-inv } min(ot, et) = N \lor satp(v, \text{min}(ot, et))) \land \text{ }
\]

\[
\text{(guar-inv } notp(v, odds(N), oc) \land bnd(oc, N)) \land \text{ }
\]

\[
\text{(rely } oc' = oc \wedge et' = et \wedge v' = v) \land \text{ }
\]

\[
\text{oc}, \text{ot: } [oc < ot, -1 \leq ot' - oc' < ot - oc]
\]

\begin{itemize}
  \item if $P(v(oc))$ then
  \begin{itemize}
    \item (guar-inv $min(ot, et) = N \lor satp(v, \text{min}(ot, et))) \land \text{ }
      \text{guar-inv } notp(v, odds(N), oc) \land bnd(oc, N)) \land \text{ }
      \text{(rely } oc' = oc \wedge et' = et \wedge v' = v) \land \text{ }
      \text{oc}, \text{ot: } [P(v(oc)) \wedge oc < ot, -1 \leq ot' - oc' < ot - oc]
  \end{itemize}
\end{itemize}

else

\[
\text{(guar-inv } min(ot, et) = N \lor satp(v, \text{min}(ot, et))) \land \text{ }
\]

\[
\text{(guar-inv } notp(v, odds(N), oc) \land bnd(oc, N)) \land \text{ }
\]

\[
\text{(rely } oc' = oc \wedge et' = et \wedge v' = v) \land \text{ }
\]

\[
\text{oc}, \text{ot: } [\neg P(v(oc)) \wedge oc < ot, -1 \leq ot' - oc' < ot - oc]
\]
Finally, Law assignment-rely-guarantee can be applied to each of the branches. Each assignment ensures the guarantee invariant
\[
\min(\text{ot, et}) = N \lor \text{satp}(v, \min(\text{ot, et})) \land \text{notp}(v, \text{odds}(N), \text{oc}) \land \text{bnd}(\text{oc}, N) \text{ is maintained.}
\]

\[
\begin{array}{c}
\text{if } P(v(\text{oc})) \text{ then } \text{ot} := \text{oc} \text{ else } \text{oc} := \text{oc} + 2
\end{array}
\]

The development of the “evens” branch of the parallel composition follows the same pattern as that of the “odds” branch given above but starts at zero. The collected code follows.

\[
\begin{array}{c}
\text{var ot, et} \\
\text{ot} := N; \\
\text{et} := N; \\
\text{var oc} \\
\text{oc} := 1; \\
\text{while } \text{oc} < \text{ot} \land \text{oc} < \text{et} \text{ do} \\
\quad \text{if } P(v(\text{oc})) \text{ then } \text{ot} := \text{oc} \\
\quad \text{else } \text{oc} := \text{oc} + 2
\end{array}
\]

\[
\begin{array}{c}
\text{var ec} \\
\text{ec} := 0; \\
\text{while } \text{ec} < \text{ot} \land \text{ec} < \text{et} \text{ do} \\
\quad \text{if } P(v(\text{ec})) \text{ then } \text{et} := \text{ec} \\
\quad \text{else } \text{ec} := \text{ec} + 2
\end{array}
\]

\[
t := \min(\text{ot, et})
\]

\[
\begin{array}{c}
\text{Treiber stack}
\end{array}
\]

Abstract state is a sequence of values

\[
\text{var } A : \text{seq } Val
\]

Specification uses atomic step style

\[
\text{Push}(v: Val)
\]

\[
\begin{array}{c}
\langle \text{id} \rangle^a : A : \langle A' = [v] \land A \rangle; \langle \text{id} \rangle^* \\
\text{rely } A' = A \sqcap (\langle \text{id} \rangle^*; A : \langle A' = [v] \land A \rangle; \langle \text{id} \rangle^*)
\end{array}
\]

\[
\text{Pop}(r: [Val])
\]

\[
\begin{array}{c}
\langle \text{id} \rangle^a : A, r : \langle A = [r'] \land A' \land (A = []) = A' \land r' = \text{null} \rangle; \langle \text{id} \rangle^* \\
\text{rely } A' = A \sqcap (\langle \text{id} \rangle^*; A, r : \langle A = [r'] \land A' \land (A = []) = A' \land r' = \text{null} \rangle; \langle \text{id} \rangle^*)
\end{array}
\]

\[
\text{Treiber stack representation}
\]

Representation as a linked list

\[
\text{type } \text{Node} = \{ \text{data} : Val, \text{next} : *\text{Node} \}
\]

\[
\text{var } s : *\text{Node}
\]

Abstraction relation

\[
\text{stack}(s : *\text{Node}, A : \text{seq Val}) =
\text{(s = null } \land A = [] \} \lor
(\exists v, n : s \mapsto \text{Node}(v, n) \land \text{head}(A) = v \land \text{stack}(n, \text{tail}(A)))
\]
Repeat statement semantics

\[
\text{repeat } c \text{ until } b = ((\text{id} \ast ; c; [\neg b]) \omega ; (\text{id} \ast ; c ; [b])
\]

Push specification (possibly nonterminating)

\[
\langle \text{id} \rangle \omega ; A : \langle A' = [v] \cap A \rangle ; (\text{id} \ast )
\]

= \((\text{id} \ast )\omega ; (\text{id} \ast ) : A : \langle A' = [v] \cap A \rangle ; (\text{id} \ast )
\]

To implement this specification as a repeat statement, we want

\[
(\text{id} \ast ) \subseteq (\text{id} \ast ) : c : [\neg b]
\]

\((\text{id} \ast ) : A : \langle A' = [v] \cap A \rangle ; (\text{id} \ast ) \subseteq (\text{id} \ast ) : c ; [b]
\]

but needs change of representation as well

Push specification (possibly nonterminating)

\[
\langle \text{id} \rangle \omega ; A : \langle A' = [v] \cap A \rangle ; (\text{id} \ast )
\]

= \((\text{id} \ast )\omega ; (\text{id} \ast ) : A : \langle A' = [v] \cap A \rangle ; (\text{id} \ast )
\]

but needs change of representation as well

Matching the refinement conditions

\[
x, done : (\text{true} \ast )
\]

\[
(\text{id} \ast ) ; \text{var } t : * Node ; (t := s) ; x.next := t ; \text{CAS}(s, t, x, done) ; [\neg done]
\]

\[
x, done : (\text{true} \ast ) ; x, s, done : \exists A, A' . A' = [v] \cap A \land stack(s, A) \land \text{stack}(s', A')
\]

\[
(\text{id} \ast ) \subseteq (\text{id} \ast ) ; \text{var } t : * Node ; (t := s) ; x.next := t ; \text{CAS}(s, t, x, done) ; [done]
\]

Theory for rely/guarantee concurrency motivated by

- Abstract algebra
- Program algebras
- Aczel traces and their synchronous parallel operator

Overview
Your algebra background

What algebras do you know?
- Groups
- Semi-groups
- Monoids
- Lattices – ordered plus infimum (meet) and supremum (join)
- Kleene Algebra – algebra of regular expressions
- Kleene Algebra with Tests (KAT)
- Concurrent Kleene Algebra (CKA)

Monoids

From mathematics we have abstract algebras
- Monoid \((S, \oplus, e)\) over a set \(S\) with binary operator \(\oplus: S \times S \rightarrow S\)
  - Associative: \(x_0 \oplus (x_1 \oplus x_2) = (x_0 \oplus x_1) \oplus x_2\)
  - Identity: \(x \oplus e = x = e \oplus x\)
- Examples of monoids
  - \((\mathbb{N}, +, 0)\)
  - \((\mathbb{N}, *, 1)\)
  - \((\text{Programs}, ;, \text{nil})\)
  - \((\text{Programs}, ||, \text{skip})\)
  - \((\text{Programs}, \&\&, \text{chaos})\)
- All except \((\text{Programs}, ;, \text{nil})\) are commutative monoids
  - Commutative: \(x_0 \oplus x_1 = x_1 \oplus x_0\)

Kleene algebra - the algebra of regular expressions

<table>
<thead>
<tr>
<th>Alternatives</th>
<th>Regular expressions</th>
<th>Relations</th>
<th>Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>(e_0 \mid e_1)</td>
<td>(e_0, e_1)</td>
<td>(r_0 \cup r_1)</td>
<td>(c_0 \cap c_1)</td>
</tr>
<tr>
<td>Sequence</td>
<td>(e^*)</td>
<td>(r^*)</td>
<td>(c^*)</td>
</tr>
<tr>
<td>Kleene star</td>
<td>(\epsilon)</td>
<td>(\text{id})</td>
<td>(\text{nil})</td>
</tr>
<tr>
<td>Identity of sequence</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\top)</td>
</tr>
<tr>
<td>Identity of alternation</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
<td>(\top)</td>
</tr>
<tr>
<td>Basic elements</td>
<td>(a)</td>
<td>((x, y))</td>
<td>(\Pi(\sigma_0, \sigma_1))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\mathcal{E}(\sigma_0, \sigma_1))</td>
</tr>
</tbody>
</table>

where \(a\) is a symbol; \(x\) and \(y\) are elements of the base type of the relation; and \(\sigma_0\) and \(\sigma_1\) are program states.

Structure of concurrent program algebra

- Concurrent refinement algebra \((\cap, \cup, ;, ||, \&\&\&\&, \top)\)
- Plus tests – a subset of commands that forms a boolean algebra
  - like Kozen's Kleene Algebra with Tests (KAT)
- Plus atomic steps – a subset of commands that forms a boolean algebra
  - Program/environment steps – partitions atomic steps
  - Relational instantiation
Operators

- \( c \cap d \) non-deterministic choice (lattice infimum or meet)
  
  \[
  (c_0 \cap c_1) \cap c_2 = c_0 \cap (c_1 \cap c_2)
  \]
  - associative

  \( c_0 \cap c_1 = c_1 \cap c_0 \)
  - commutative

  \( c \cap c = c \)
  - idempotent

  \( c \cap \top = c = \top \cap c \)
  - identity \( \top \)

- \( c \cup d \) lattice supremum or join
  - associative, commutative, idempotent, identity \( \bot \)

- \( c \parallel d \) parallel composition
  - associative, commutative, identity \texttt{skip}

- \( c \otimes d \) weak conjunction
  - associative, commutative, idempotent, identity \texttt{chaos}

- \( c ; d \) sequential composition (sometimes elided to \( c d \) below)
  - associative, identity \texttt{nil}

\( \cap \) and \( \cup \) have the same precedence, which is lower than \( \parallel \) and \( \otimes \), which are lower than \; .

Aczel traces

- Represent
  - a program doing a step from \( \sigma_0 \) to \( \sigma_1 \) by \( \Pi(\sigma_0, \sigma_1) \) and
  - its environment doing a step from \( \sigma_0 \) to \( \sigma_1 \) by \( \mathcal{E}(\sigma_0, \sigma_1) \).

Every step of parallel synchronises steps of the two processes

\[
\mathcal{E}(\sigma_0, \sigma_1), \Pi(\sigma_1, \sigma_2), \mathcal{E}(\sigma_2, \sigma_3), \mathcal{E}(\sigma_3, \sigma_4), \mathcal{E}(\sigma_4, \sigma_5) \parallel
\mathcal{E}(\sigma_0, \sigma_1), \mathcal{E}(\sigma_1, \sigma_2), \Pi(\sigma_2, \sigma_3), \mathcal{E}(\sigma_3, \sigma_4), \Pi(\sigma_4, \sigma_5)
\]

\[
\mathcal{E}(\sigma_0, \sigma_1), \Pi(\sigma_1, \sigma_2), \mathcal{E}(\sigma_2, \sigma_3), \mathcal{E}(\sigma_3, \sigma_4), \Pi(\sigma_4, \sigma_5)
\]

Every step of a weak conjunction synchronises steps of the two processes

\[
\mathcal{E}(\sigma_0, \sigma_1), \Pi(\sigma_1, \sigma_2), \mathcal{E}(\sigma_2, \sigma_3), \mathcal{E}(\sigma_3, \sigma_4), \Pi(\sigma_4, \sigma_5) \otimes
\mathcal{E}(\sigma_0, \sigma_1), \Pi(\sigma_1, \sigma_2), \mathcal{E}(\sigma_2, \sigma_3), \mathcal{E}(\sigma_3, \sigma_4), \Pi(\sigma_4, \sigma_5)
\]

Complete lattice

- For any set of commands \( C \)
  - \( \cap C \) is the infimum (greatest lower bound) of the set of commands
  - \( \cup C \) is the supremum (least upper bound) of the set of commands

Primitive atomic commands

For a binary relation \( r \subseteq \Sigma \times \Sigma \) on states

- \( \pi(r) \) can perform any single atomic program step \( \Pi(\sigma, \sigma') \) for \( (\sigma, \sigma') \in r \)
- \( \epsilon(r) \) can perform any single atomic environment step \( \mathcal{E}(\sigma, \sigma') \) for \( (\sigma, \sigma') \in r \)

For example,

- \( \pi(\text{id}) \) is a single stuttering program step (\text{id} is the identity relation)
- \( \pi = \pi(\text{univ}) \) can perform any single program step (\text{univ} is the universal relation)
- \( \epsilon = \epsilon(\text{univ}) \) can perform any single environment step
- \( \pi(\emptyset) = \epsilon(\emptyset) = \top \) is infeasible (magic)

Atomic steps form a boolean algebra

\[
\pi(r_0) \cap \pi(r_1) = \pi(r_0 \cup r_1)
\]

\[
\pi(r_0) \cup \pi(r_1) = \pi(r_0 \cap r_1)
\]

\( \neg \pi(r) = \pi(\top) \cap \epsilon \)
Tests as a boolean algebra

For a set of states \( p \subseteq \Sigma \),
\[ \tau(p) \]
terminates immediately if \( p \) holds but is infeasible otherwise.

For example,
- \( \tau(\Sigma) = \text{nil} \)
- \( \tau(\emptyset) = \top \)
- \( \tau(p_1 \cap p_2) = \tau(p_1 \cup p_2) \)
- \( \tau(p_1 \cup p_2) = \tau(p_1) \cap \tau(p_2) = \tau(p_1) \parallel \tau(p_2) = \tau(p_1 \cap p_2) \)
- \( \neg \tau(p) = \tau(\overline{p}) \)

Assertions/preconditions: for a test \( t \)
- \( \text{pre} t = t \cap \neg t ; \bot \)
- \( \{p\} = \text{pre} \tau(p) = \tau(p) \cap \tau(\overline{p}) ; \bot \)

Assumptions

For a set of states \( p \subseteq \Sigma \),
\[ \tau(p) \]
terminates immediately if \( p \) holds but is infeasible otherwise.

For example,
- \( \tau(\Sigma) = \text{nil} \)
- \( \tau(\emptyset) = \top \)
- \( \tau(p_1 \cap p_2) = \tau(p_1 \cup p_2) \)
- \( \tau(p_1 \cup p_2) = \tau(p_1) \cap \tau(p_2) = \tau(p_1) \parallel \tau(p_2) = \tau(p_1 \cap p_2) \)
- \( \neg \tau(p) = \tau(\overline{p}) \)

Assertions/preconditions: for a test \( t \)
- \( \text{pre} t = t \cap \neg t ; \bot \)
- \( \{p\} = \text{pre} \tau(p) = \tau(p) \cap \tau(\overline{p}) ; \bot \)

Assumptions

For a set of states \( p \subseteq \Sigma \),
\[ \tau(p) \]
terminates immediately if \( p \) holds but is infeasible otherwise.

For example,
- \( \tau(\Sigma) = \text{nil} \)
- \( \tau(\emptyset) = \top \)
- \( \tau(p_1 \cap p_2) = \tau(p_1 \cup p_2) \)
- \( \tau(p_1 \cup p_2) = \tau(p_1) \cap \tau(p_2) = \tau(p_1) \parallel \tau(p_2) = \tau(p_1 \cap p_2) \)
- \( \neg \tau(p) = \tau(\overline{p}) \)

Assertions/preconditions: for a test \( t \)
- \( \text{pre} t = t \cap \neg t ; \bot \)
- \( \{p\} = \text{pre} \tau(p) = \tau(p) \cap \tau(\overline{p}) ; \bot \)

Assumptions

For a set of states \( p \subseteq \Sigma \),
\[ \tau(p) \]
terminates immediately if \( p \) holds but is infeasible otherwise.

For example,
- \( \tau(\Sigma) = \text{nil} \)
- \( \tau(\emptyset) = \top \)
- \( \tau(p_1 \cap p_2) = \tau(p_1 \cup p_2) \)
- \( \tau(p_1 \cup p_2) = \tau(p_1) \cap \tau(p_2) = \tau(p_1) \parallel \tau(p_2) = \tau(p_1 \cap p_2) \)
- \( \neg \tau(p) = \tau(\overline{p}) \)

Assertions/preconditions: for a test \( t \)
- \( \text{pre} t = t \cap \neg t ; \bot \)
- \( \{p\} = \text{pre} \tau(p) = \tau(p) \cap \tau(\overline{p}) ; \bot \)

Synchronise atomic steps

For atomic commands \( a \) and \( b \) (think \( \pi \) and \( \epsilon \) commands) and arbitrary commands \( c \) and \( d \)

\[
(a; c) \cap (b; d) = (a \cap b); (c \cap d)
\]

\[
(a; c) \cap \text{nil} = \top
\]

\[
a \cap \bot = \bot
\]

Laws

\[
a^* \cap b^* = (a \cap b)^*
\]

\[
a^* ; c \cap b^* ; d = (a \cap b)^* ((c \cap d) \cap (a ; a^* ; c \cap d) \cap (c \cap b ; b^* ; d))
\]

For program and environment steps

\[
\pi(r_1) \parallel \pi(r_2) = \top
\]

\[
\pi(r_1) \parallel \epsilon(r_2) = \pi(r_1 \cap r_2)
\]

\[
\epsilon(r_1) \parallel \epsilon(r_2) = \epsilon(r_1 \cap r_2)
\]

\[
\pi(r) \parallel \bot = \bot
\]

\[
\epsilon(r) \parallel \bot = \bot
\]

Note that program and environment steps partition atomic steps.
Interchange laws

Weak conjunction interchange sequential

\((c_0 : c_1) \sqcap (d_0 : d_1) \sqsubseteq (c_0 \sqcap d_0) : (c_1 \sqcap d_1)\)

Weak conjunction interchange parallel

\((c_0 \parallel c_1) \sqcap (d_0 \parallel d_1) \sqsubseteq (c_0 \sqcap d_0) \parallel (c_1 \sqcap d_1)\)

Iteration zero or more times, \(c^\omega\), allows finite iteration, \(c^*\), or infinite iteration, \(c^\infty\)

\[ c^\omega = c^* \sqcap c^\infty \] (3)

Examples

\(\pi^*\) performs a finite number of program steps

\((\pi \sqcap \epsilon)^*\) performs a finite number of steps

\(\epsilon^\infty\) performs an infinite sequence of environment steps

\(\text{skip}\) is the identity of parallel and \(\text{chaos}\) is the identity of weak conjunction

\[ \text{skip} = \epsilon^\omega \]

\[ \text{chaos} = (\pi \sqcap \epsilon)^\omega \]

Asynchronised atomic step

\((r) = \epsilon^\omega; \pi(r); \epsilon^\omega\)

For example

\((r_1) \parallel (r_2) = \epsilon^\omega; (\pi(r_1) \parallel \pi(r_2)); \epsilon^\omega \sqcap (r_1) \parallel (r_2) \sqcap (r_1) = (r_1) \parallel (r_2) \sqcap (r_2) \sqcap (r_1)\)

Rely/guarantee

\[
\begin{align*}
\text{pre} & \quad \epsilon \quad \pi \quad \epsilon \quad \pi \quad \epsilon \\
\text{post} & \quad g \quad g \quad g
\end{align*}
\]
Guarantee and rely

For relations $g$ and $r$

\[
\text{guar } g = (\pi(g) \cap \epsilon)^\omega
\]
\[
\text{rely } r = (\pi \cap \epsilon(r) \cap \epsilon(\top))\omega
\]

\[= (\text{assume } ! \epsilon(\top))\omega\]

recalling $\text{assume } a = a \cap ! a : \bot$ and $\epsilon(\top) = \pi \cap \epsilon(r)$

For example, $c \parallel \text{guar } g \parallel \text{rely } r$ imposes a guarantee of $g$ on $c$ and assumes the environment steps satisfy $r$.

Lemmas for specifications

- Frames on commands
  
  \[x : c = (\text{guar id}(x)) \cap c\]

- Atomic operation
  
  \[(q) = \epsilon^\omega : \pi(q) : \epsilon^\omega\]

- Non-atomic specification (relational post)
  
  \[\{q\} = \prod_{\sigma \in \Sigma} \tau(\{\sigma\}) ; \text{term} ; \tau(\{\sigma' \in \Sigma \mid (\sigma, \sigma') \in q\})\]

Specification commands

Parallel introduction

The command $\text{term}$ allows only a finite number of program steps but does not rule out infinite pre-emption by its environment.

\[\text{term} = (\epsilon^\omega ; \pi)^\ast ; \epsilon^\omega\]

The refinement

\[\text{term} \sqsubseteq c\]

states that $c$ terminates if the environment does not interrupt it forever, e.g.

\[\text{term} \sqsubseteq x := 1\]

Proof

\[(\text{rely } r) \sqcap [q_1 \wedge q_2] \sqsubseteq (\text{rely } r \cup r_1) \sqcap [q_1] \sqcap (\text{guar } r \cup r_2) \parallel (\text{rely } r \cup r_2) \sqcap [q_2] \sqcap (\text{guar } r \cup r_1)\]

by Lemma Y (twice)

\[(\text{rely } r \cup r_1) \sqcap [q_1] \parallel (\text{guar } r \cup r_2) \sqcap [q_1] \parallel (\text{guar } r \cup r_1) \parallel (\text{rely } r \cup r_2) \parallel [q_2]

\equiv \text{conjunction-interchange-parallel } (c_1 \parallel q_2) \sqcap (d_1 \parallel d_1) \sqsubseteq (c_1 \sqcap d_1) \parallel (c_2 \sqcap d_2)

by Lemma Q1

\[(\text{rely } r \cup r_1) \sqcap [q_1] \parallel (\text{guar } r \cup r_2) \parallel (\text{rely } r \cup r_2) \sqcap [q_2]\]
Lemma Y

\[(\text{rely } r) \cap [q] \subseteq ((\text{rely } r) \cap [q]) \parallel ((\text{guar } r) \cap \text{term})\]

Proof

\[(\text{rely } r) \cap [q] \subseteq ((\text{rely } r) \parallel ((\text{guar } r)) \cap ([q] \parallel \text{term})] \subseteq \text{conjunction-interchange-parallel} \ (c_1 || d_1) \cap (c_2 || d_2) \cap ([q] \parallel ((\text{guar } r) \cap \text{term})]

Lemmas X and Q

Lemma X

\[(\text{rely } r) \subseteq (\text{rely } r) \parallel (\text{guar } r)\]

Lemma Q1

\[[q] \cap \text{term} = [q]\]

Lemma Q2

\[[q] \parallel \text{term} = [q]\]

Applying to process algebras

The above approach can also be applied to CSP-style processes.

- II(a) is interpreted as an atomic event or action a
- \(\mathcal{E}(a)\) is a corresponding environment event
- \(\pi(A)\) allows any event II(a) for any a \(\in\ A\)
- \(\epsilon(A)\) allows any environment event \(\mathcal{E}(a)\) for any a \(\in\ A\)

CSP

Synchronising on common events

- \(\pi(A) \parallel \pi(B) = \pi(A \cap B)\)

Alphabet A for a command c - environment can do only events in \(\mathcal{A}\) independently

- \(A : c = c \parallel \epsilon(\mathcal{A})\)

Hoare's parallel for a process c with alphabet A and process d with alphabet B

- \(A : c \parallel B : d\)

Roscoe's parallel alphabetised by A

- \(c \parallel_A d = A : c \parallel A : d\)
Iteration

Finite iteration zero or more times, \( c^* \), possibly infinite iteration zero or more times, \( c^\omega \), and infinite iteration, \( c^\infty \), are defined via their usual recursive equations and have the following unfolding and induction properties.

\[
\begin{align*}
c^* & \equiv \nu x \cdot \text{nil} \sqcap c \cdot x \\
c^* & = \text{nil} \sqcap c^* \\
x \sqsubseteq d \sqcap c \cdot x & \Rightarrow x \sqsubseteq c^* \cdot d \\
c^* & = \text{nil} \sqcap c^* \cdot c \\
x \sqsubseteq d \sqcap x \cdot c & \Rightarrow x \sqsubseteq d \sqcap c^* \\
c^\omega & \equiv \mu x \cdot \text{nil} \sqcap c \cdot x \\
c^\omega & = \text{nil} \sqcap c \cdot c^\omega \\
d \sqcap c \cdot x \sqsubseteq x & \Rightarrow c^\omega \sqcap d \sqcap x \\
c^\infty & \equiv \mu x \cdot c \cdot x \\
c^\infty & = c \cdot c^\infty \\
c \cdot x \sqsubseteq x & \Rightarrow c^\infty \sqsubseteq x
\end{align*}
\]

Some basic commands

\[
\begin{align*}
\text{update}(x, v) & \equiv \pi(x' = v \land \text{id}(x)) \\
\text{skip} & \equiv \epsilon^\omega \\
\text{chaos} & \equiv (\pi \sqcap \epsilon)^\omega \\
\text{term} & \equiv (\pi \sqcap \epsilon)^* \cdot \text{skip} \\
\text{idle} & \equiv (\pi(\text{id}) \sqcap \epsilon)^* \cdot \text{skip} \\
\langle r \rangle & \equiv \text{skip} : \pi(r) : \text{skip}
\end{align*}
\]

Relies and guarantees

\[
\begin{align*}
\text{guar } g & \equiv (\pi(g) \sqcap \epsilon)^\omega \\
\text{rely } r & \equiv (\pi \sqcap \epsilon(r)^\omega ; (\text{nil} \sqcap \epsilon(\tau) ; \bot) \\
x : c & \equiv (\text{guar } \text{id}(\tau)) \sqcap c
\end{align*}
\]

The process \((\text{guar } g) \sqcap c\) behaves as both \((\text{guar } g)\) and as \(c\), unless at some point \(c\) aborts, in which case \((\text{guar } g) \sqcap c\) aborts; note that \((\text{guar } g)\) cannot abort. For example, the guarantee \((\text{guar } w' \supset w \land w - w' \subseteq \{i\})\) ensures that no step of the process may add elements to \(w\) or remove elements other than \(i\).

Tests and steps

\[
\begin{align*}
\tau(p_1) ; \tau(p_2) & = \tau(p_1 \land p_2) \\
\tau(p) ; \pi(r) & = \pi(p \land r) \\
\tau(p) ; \epsilon(r) & = \epsilon(p \land r) \\
\pi(r \land p') ; \tau(p) & = \pi(r \land p') \\
\epsilon(r \land p') ; \tau(p) & = \epsilon(r \land p')
\end{align*}
\]
Invariance

If \( c; \tau(p) \sqsubseteq \tau(p); c \), then
\[
\tau(p); c; \tau(p) = \tau(p); c.
\]

Proof.
\[
\tau(p); c; \tau(p) \sqsubseteq \tau(p); \tau(p); c = \tau(p); c = \tau(p); c; \tau(p).
\]

Invariance over steps

If \( r \Rightarrow (p \Rightarrow p') \), then both the following hold.
\[
\begin{align*}
\pi(r); \tau(p) &\sqsubseteq \tau(p); \pi(r) \\
\epsilon(r); \tau(p) &\sqsubseteq \tau(p); \epsilon(r)
\end{align*}
\]  \quad (20)  \quad (21)

Proof.
The assumption ensures \( p \land r \land p' = p \land r \). We give the proof for (20) which uses (22). The proof for (21) is similar but uses (23).
\[
\begin{align*}
\pi(r); \tau(p) &= \nil; \pi(r); \tau(p) \sqsubseteq \tau(p); \pi(r); \tau(p) = \pi(p \land r); \tau(p) \\
&= \pi(p \land r \land p') \land \tau(p) = \pi(p \land r) = \tau(p); \pi(r)
\end{align*}
\]

Invariance over iterations

If \( c; \tau(p) \sqsubseteq \tau(p); c \), then both
\[
\begin{align*}
c' \sqsubseteq \tau(p); c' &\sqsubseteq \tau(p); c' &\quad (22) \\
c' \sqsubseteq \tau(p) &\sqsubseteq \tau(p); c' &\quad (23)
\end{align*}
\]

Proof.
Property (22) holds by \( \omega \)-induction (22) if \( \tau(p) \sqcap c; \tau(p); c' \sqsubseteq \tau(p); c' \), which can be proven using the assumption and \( \omega \)-folding (22).
\[
\tau(p) \sqcap c; \tau(p); c' \sqsubseteq \tau(p) \sqcap \tau(p); c; c' = \tau(p); (\nil \sqcap c; c') = \tau(p); c'
\]

Property (23) holds by \( \ast \)-induction (23) if \( c' \sqsubseteq \tau(p) \sqsubseteq \tau(p) \sqcap c' \sqcap \tau(p); c \), which can be proven using the assumption and \( \ast \)-folding (23).
\[
\tau(p) \sqcap c' \sqcap \tau(p); c \sqsubseteq \tau(p) \sqcap c' \sqcap c; \tau(p) = (\nil \sqcap c' \sqcap c); \tau(p) = c' \sqcap \tau(p)
\]

Rely-invariant

If \( r \Rightarrow (p \Rightarrow p') \), then
\[
((\text{rely } r) \sqcap \text{idle}) \sqsubseteq \tau(p) \sqsubseteq ((\text{rely } r) \sqcap \text{idle})
\]

Proof.
The proof uses the definitions of rely \( r \) (112) and idle (110) and then pushes the test \( \tau(p) \) left using applications of Lemma invariance-iteration. Note that the identity relation \( \text{id} \) maintains any invariant \( p \).
\[
\begin{align*}
((\text{rely } r) \sqcap \text{idle}) \sqsubseteq \tau(p) &= ((\pi \sqcap \epsilon(r))^\omega; (\nil \sqcap \epsilon(r); \bot) \sqcap (\pi + \epsilon(r))^{\ast}; \epsilon^\omega; \tau(p) \\
&= (\pi + \epsilon(r))^{\ast}; \epsilon^\omega; (\nil \sqcap \epsilon(r); \bot); \tau(p) \\
&= (\pi \sqcap \epsilon(r))^{\ast}; \epsilon^\omega; (\tau(p) \sqcap \epsilon(r); \bot); \tau(p) \\
&= (\pi \sqcap \epsilon(r))^{\ast}; \epsilon^\omega; (\epsilon^r \sqcap \epsilon(r); \bot); \tau(p) \\
&= (\pi \sqcap \epsilon(r))^{\ast}; \epsilon^\omega; (\tau(p) \sqcap \epsilon(r); \bot) \\
&= (\tau(p) \sqcap \epsilon(r))^{\ast}; \epsilon^\omega; (\nil \sqcap \epsilon(r); \bot) \\
&= (\tau(p) \sqcap \epsilon(r))^{\ast}; \epsilon^\omega; (\nil \sqcap \epsilon(r); \bot) \\
&= (\tau(p) \sqcap \epsilon(r))^{\ast}; \epsilon^\omega; (\nil \sqcap \epsilon(r); \bot)
\end{align*}
\]

\( = \tau(p) \sqsubseteq ((\text{rely } r) \sqcap \text{idle}) \)
Defining expressions

\[[\kappa]_v \equiv \text{idle} : \tau(\kappa = v) : \text{idle}\]

\[[x]_v \equiv \text{idle} : \tau(x = v) : \text{idle}\]

\[[\ominus e]_v \equiv \bigcap \{v_1 \mid v = \text{eval}(\ominus, v_1) \cdot [[e]]_{v_1}\}\]

\[[e_1 \oplus e_2]_v \equiv \bigcap \{v_1, v_2 \mid v = \text{eval}(\oplus, v_1, v_2) \cdot [[e_1]]_{v_1} \parallel [[e_2]]_{v_2}\}\]

Stable-expression

An expression is stable under \(r\) if its evaluation is not affected by interference satisfying \(r\). For example, assuming access to \(x\) is atomic, the absolute value of \(x\), |\(x|\), is stable under interference satisfying \(x' = x \lor x' = -x\), and \((x \mod N)\) is stable under interference satisfying \(x' = x \lor x' = x + N\).

Definition (stable-expression)

An expression \(e\) is stable under \(r\) if, for fresh \(v\),

\[r \Rightarrow (e = v \Rightarrow e' = v)\]

Stable expression

In the context of interference represented by a rely condition \(r\), an expression \(e\) is stable if all the variables used in \(e\) are stable under \(r\). If a variable \(x\) is not subject to change, access to it does not need to be atomic.

- ▶ A constant \(\kappa\) is trivially stable.
- ▶ A variable \(x\) is stable under \(r\) if for fresh \(v\), \(r \Rightarrow (x = v \Rightarrow x' = v)\).
- ▶ A unary expression \(\ominus e\) is stable under \(r\) if \(e\) is.
- ▶ A binary expression \(e_1 \oplus e_2\) is stable under \(r\) if both \(e_1\) and \(e_2\) are.

Rely stable expression

If an expression \(e\) is stable under \(r\), then for any value \(v\) where \(v\) does not occur free in \(e\),

\[(\text{rely } r) \sqcap (\text{idle}; \tau(e = v)) \sqsubseteq (\text{rely } r) \sqcap (\tau(e = v); \text{idle})\]

Proof.

This lemma follows directly from Definition stable-expression and Law rely-invariant.
Single-reference expressions

Evaluating an expression in the context of interference may lead to anomalies because evaluation of an expression such as \( x + x \) may retrieve different values of \( x \) for each of its occurrences and hence it is possible for \( x + x \) to evaluate to an odd value even though \( x \) is an integer variable. Such anomalies may be avoided in the case that expressions are single reference \( \texttt{?} \). If \( x \) is subject to modification then \( x + x \) is not single-reference but \( 2 \times x \) is. An expression being stable under \( r \) is considered a special case of it being single reference so, for example, if \( x \) is not subject to interference then \( x + x \) is single-reference.

Definition (single-reference-expression)
The definition is based on the syntactic form of \( e \).

- A constant \( \kappa \) is single reference.
- A variable \( x \) is single reference provided access to \( x \) is atomic.
- A unary expression \( \ominus e \) is single reference under \( r \) if \( e \) is.
- A binary expression \( e_1 \oplus e_2 \) is single reference under \( r \) if either \( e_1 \) is single reference under \( r \) and \( e_2 \) is stable under \( r \), or vice versa.

If an expression \( e \) is single-reference then for any evaluation of \( e \), its value is the same as the evaluation of \( e \) in the single state \( \sigma \) in which the single-reference variable \( (x) \) is accessed.

Defining commands

\[
\begin{align*}
x & := e & \equiv & \prod_{v \in \text{Val}} [e]_v : \text{update}(x, v) : \text{idle} & \quad (28) \\
\text{if } b \text{ then } c \text{ else } d & \equiv & (\left\lbrack \text{true} \right\rbrack_c \cap \left\lbrack \text{false} \right\rbrack_d) : \text{idle} & \quad (29) \\
\text{while } b \text{ do } c & \equiv & (\left\lbrack \text{true} \right\rbrack_c \cup \left\lbrack \text{false} \right\rbrack_d) : \text{idle} & \quad (30) \\
[q] & \equiv & \prod_{\sigma \in \Sigma} \tau(\{\sigma\}) : \text{term} : \tau(\{\sigma' \mid (\sigma, \sigma') \in q\}) & \quad (31) \\
[p, q] & \equiv & \{p\} : [q] & \quad (32)
\end{align*}
\]
An expression $e$ is single reference under interference satisfying the rely condition $r$ if the value of the expression corresponds to its value in one of the states during its evaluation and hence one can derive the following law.

If $e$ is a single-reference expression under $r$,

$$(\text{rely } r) \cap (\text{idle} : \tau(e = \kappa) ; \text{idle}) \sqsubseteq [e]_{\kappa}.$$ 

Proof.
The proof is by structural induction of the structure of the expression. □

If rely condition $r$ is such that $r \Rightarrow (p \Rightarrow p')$,

$$(\text{rely } r) \cap [p \wedge p'] \sqsubseteq (\text{rely } r) \cap \text{idle}.$$ 

Proof.
All environment steps of the right side are assumed to satisfy $r$ and all program steps satisfy the identity relation, and hence the right side guarantees to maintain $p$ and satisfies $r^*$. □

For a single-reference boolean expression $b$, predicates $p$ and $b_0$, and relation $r$, if $r$ maintains $p$, $p \wedge b \Rightarrow b_0$, and $p \wedge r \Rightarrow (b_0 \Rightarrow b'_0)$,

$$(\text{rely } r) \cap [p \wedge r^* \wedge p' \wedge b'_0] \sqsubseteq [b]_{\text{true}}.$$ 

Proof.
The proof uses Law rely-sequential and Law rely-idle.

Let $r$ be a rely condition, $x$ a variable that is stable under $r$, and $e$ a single-reference expression such that $x$ does not occur free in $e$ and “$\approx$” a reflexive, transitive binary relation, such that $r \Rightarrow (e \approx e')$, then

$$(\text{rely } r) \cap x : [e \approx x' \approx e'] \sqsubseteq x := e$$

For example, the relation may be equality (so that $e$ is stable) and we have $e = x' = e'$, or the relation may be may be “$\supseteq$”, so the postcondition becomes $e \supseteq x' \supseteq e'$. □
Proof.
The proof uses Law rely-sequential and Law rely-idle and the definition of assignment (127).

\[
\begin{align*}
\text{(rely } r\text{) } & \text{x : } e \approx x' \approx e' \\
\text{(rely } r\text{) } & \text{x : } \exists \mathcal{V} \cdot e \approx \mathcal{V} \approx e' \land x' = \mathcal{V} \\
\text{(rely } r\text{) } & \text{x : } \exists \mathcal{V} \cdot e \approx \mathcal{V} \approx e' \land x' = \mathcal{V} \\
\text{(rely } r\text{) } & \text{x : } e \approx e' ; \mathcal{V} = e = \mathcal{V} \\
\text{(rely } r\text{) } & \text{x : } e \approx e' ; \mathcal{V} = e = \mathcal{V} \land x' = \mathcal{V} \\
\text{(rely } r\text{) } & \text{x : } e \approx e' ; \mathcal{V} = e = \mathcal{V} \land x' = \mathcal{V} \\
\text{x := } & e \\
\end{align*}
\]

Handling tests under interference

To handle the possible instability of \( b \) within a test, a weaker but stable predicate \( b_0 \) can be used, i.e. \( b \Rightarrow b_0 \) and \( r \Rightarrow (b_0 \Rightarrow b_0') \). More generally, if condition \( b \) is only ever evaluated in states satisfying a precondition \( p \) that is maintained by \( r \), these conditions can be relaxed to the following.

\[
p \land b \Rightarrow b_0 \quad \text{ and } \quad p \land r \Rightarrow (b_0 \Rightarrow b_0')
\]

When handling the negation of the condition, one needs an additional stable predicate \( b_1 \) that is implied by the negation of \( b \).

\[
p \land \lnot b \Rightarrow b_1 \quad \text{ and } \quad p \land r \Rightarrow (b_1 \Rightarrow b_1')
\]

For example, the negation of the earlier example is \( \text{oc} \geq \text{ot} \lor \text{oc} \geq \text{et} \) and that is maintained by interference that may only decrease \( \text{et} \). Note that

\[
p \Rightarrow (p \land b) \lor (p \land \lnot b) \Rightarrow b_0 \lor b_1
\]

but there may be states in which both \( b_0 \) and \( b_1 \) hold. For the above example, taking \( b_0 \) as \( \text{oc} \prec \text{ot} \) and \( b_1 \) as \( \text{oc} \geq \text{ot} \lor \text{oc} \geq \text{et} \), both conditions hold in states satisfying \( \text{oc} \prec \text{ot} \land \text{oc} \geq \text{et} \).

Loops

The invariant and the variant

The Hoare logic rule for reasoning about a loop, \( \text{while } b \text{ do } c \), for sequential programs utilises an invariant \( p \) that is maintained by the loop body whenever \( b \) holds initially. To show termination a variant expression \( v \) is used. The loop body must strictly decrease \( v \) according to a well-founded relation \( (\prec , \succ) \) whenever \( b \) holds initially.

The law for while loops needs to be strengthened to rule out the interference invalidating the loop invariant \( p \) or increasing the variant \( v \). The requirements on the invariant \( p \) and variant \( v \) to tolerate interference satisfying the rely condition \( r \) may be stated as follows.

\[
\begin{align*}
\text{r} \Rightarrow (p \Rightarrow p') \\
p \land r \Rightarrow v \succ v'
\end{align*}
\]

127/1
Rely finite iteration

For predicate \( p \), and relation \( q \), if \( r \) maintains \( p \),
\[
(\text{rely } r) \sqcap [p, p' \land q^*] \sqsubseteq (\text{rely } r) \sqcap [p, p' \land q]^* 
\]

Proof.
The proof is via finite iteration induction (??) and the refinement holds if,
\[
(\text{rely } r) \sqcap [p, p' \land q^*] \sqsubseteq \text{n} \sqcap ((\text{rely } r) \sqcap [p, p' \land q]); (\text{rely } r) \sqcap [p, p' \land q^*])
\]
which holds by Law rely-sequential because \( q \sqsubseteq q^* \). \( \square \)

Rely loop

Given predicates \( p, b_0 \) and \( b_1 \), a relation \( r \), a variant expression \( v \) of type \( T \) and a
relation \( (\sqsubseteq \sqsubseteq) \subseteq T \times T \) that is well-founded on states satisfying \( p \), if \( b \) is a
single-reference boolean expression under interference satisfying \( r \), and
\[
\begin{align*}
p \land r & \Rightarrow p' \\
p \land b & \Rightarrow b_0 \\
p \land r \land b_0 & \Rightarrow b_0' \\
p \land r & \Rightarrow v \sqsupseteq v' \\
p \land \neg b & \Rightarrow b_1 \\
p \land r \land b_1 & \Rightarrow b_1'
\end{align*}
\]
then
\[
(\text{rely } r) \sqcap [p, p' \land b_1' \land v \sqsupseteq v'] \sqsubseteq \text{while } b \text{ do } ((\text{rely } r) \sqcap [p \land b_0, p' \land v \triangleright v'])
\]

Rely well-founded iteration

For predicate \( p \), variant expression \( v \) of type \( T \), and a relation \((\sqsubseteq \sqsubseteq) \subseteq T \times T \) that
is well-founded on \( p \), if \( r \) maintains \( p \), and \( v \) is non-increasing under \( r \),
\[
(\text{rely } r) \sqcap [p, p' \land v \triangleright v'] \sqsubseteq (\text{rely } r) \sqcap [p, p' \land v \triangleright v']^\omega
\]

Proof.
\[
\begin{align*}
(\text{rely } r) \sqcap [p, p' \land v \triangleright v']^\omega \\
= \text{ isolation, i.e. } c^\omega = c^* \sqcap c^\infty \\
= \text{well-founded infinite iteration is infeasible}
\end{align*}
\]

\( \square \)

Proof.
\[
\begin{align*}
(\text{rely } r) \sqcap [p, p' \land v \triangleright v'] \\
\sqsubset \text{by Law rely-finite-iteration} \\
(\text{rely } r) \sqcap [p, p' \land v \triangleright v']^\omega \\
\sqsubseteq \text{by Law rely-sequential} \\
((\text{rely } r) \sqcap [p, p' \land v \triangleright v']) ; ((\text{rely } r) \sqcap [p, p' \land v \triangleright v']) \\
\sqsubset \text{by Law rely-test using the assumptions on } b_1 \\
((\text{rely } r) \sqcap [p, p' \land v \triangleright v']) ; [\neg b]_{\text{true}} \\
\sqsubset \text{by Law rely-finite-iteration} \\
((\text{rely } r) \sqcap [p, p' \land v \triangleright v'])^\omega ; [\neg b]_{\text{true}} \\
\sqsubseteq \text{by Law rely-finite-sequential as } (v \triangleright v') ; (v \triangleright v') \Rightarrow v \triangleright v' \\
(((\text{rely } r) \sqcap [p, p' \land b_0' \land v \triangleright v']) ; ((\text{rely } r) \sqcap [p \land b_0, p' \land v \triangleright v']))^\omega ; [\neg b]_{\text{true}} \\
\sqsubset \text{by Law rely-test using the assumptions on } b_0 \\
([b] ; ((\text{rely } r) \sqcap [p \land b_0, p' \land v \triangleright v'])^\omega ; [\neg b]_{\text{true}} \\
= \text{definition of loop (??)} \\
\text{while } b \text{ do } ((\text{rely } r) \sqcap [p \land b_0, p' \land v \triangleright v'])
\end{align*}
\]

\( \square \)
Given predicates $p$, $b_0$, $b_1$ and $b_2$, a relation $r$, a variant expression $v$ of type $T$ and a relation $(\_ \succ \_ \subseteq T \times T)$ that is well-founded on states satisfying $p$, if $b$ is a single-reference boolean expression under interference satisfying $r$, and

\[
\begin{align*}
  p \land r & \Rightarrow p' \\
  p \land r & \Rightarrow v \geq v' \\
  p \land \neg b & \Rightarrow b_1 \\
  p \land r \land b_1 & \Rightarrow b_1' \\
  p \land b_2 & \Rightarrow \neg b \\
  p \land r \land b_2 & \Rightarrow b_2'
\end{align*}
\]

then

\[
(\text{rely } r) \ominus [p, p' \land b_1] \subseteq \text{while } b \text{ do } ((\text{rely } r) \ominus [p \land b_0, p' \land (v \succ v' \lor b_2')])
\]

This rule may be shown using Law rely-loop by taking as the variant the ordered pair $(\neg b_2, v)$ under the lexicographical ordering, where $true \succ false$.


Conclusions

- One can develop algebras of programs
- Focus on the algebraic properties first, then semantics
- Need a semantics to show that the algebraic theories are consistent
- Start from a (refinement) lattice and add $\parallel$, $\ominus$, $;$
- For rely/guarantee, start with very primitive commands ($\tau(p), \pi(r), \epsilon(r)$)
- Links to process algebras, in particular Milner’s Synchronous CCS (SCCS)
- We are developing Isabelle theories for the algebras