An introduction to rely/guarantee reasoning about concurrency

Ian J. Hayes (and Cliff B. Jones)

The University of Queensland, Australia (and Newcastle University, UK)

SETSS Spring School
Chongqing, China
2017-04-17–19

Rely/guarantee

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: $\wedge$, $\vee$ 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: $\sqsubseteq$, $x\colon[p\,,\ q]$
- 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 $noduplicates(s)$
- postcondition $ordered(s') \land items(s') = items(s)$

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

$j := 0;$
while $j \neq N$ **do**
  if p **then** s **else** t;
  $j := j + 1$

- 2^N possible paths

Hoare logic is compositional

Structured reasoning about programs
- Sequential composition

$$\frac{\{p\}\, s\, \{q\} \quad \{q\}\, t\, \{r\}}{\{p\}\, s\,;t\, \{r\}}$$

- While loop using a loop invariant p

$$\frac{\{p \land b\}\, s\, \{p\}}{\{p\}\ \textbf{while}\ b\ \textbf{do}\ s\, \{p \land \neg\, b\}}$$

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

$$s_1;\ s_2;\ \ldots;\ s_n \parallel t_1;\ t_2;\ \ldots;\ t_n$$

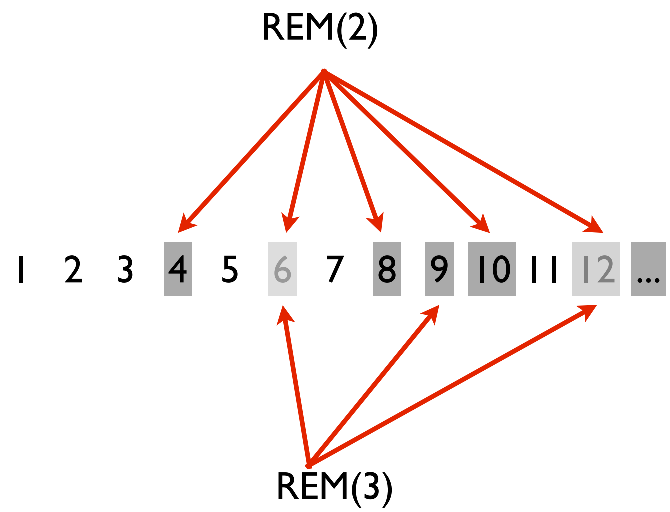
- The number of paths in terms of n explodes
- If there is no interference between s and t

$$\frac{\{p_1\}\, s\, \{q_1\} \quad \{p_2\}\, t\, \{q_2\}}{\{p_1 \land p_2\}\, s \parallel t\, \{q_1 \land q_2\}}$$

- 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



Specification in refinement calculus style

Concrete syntax

VDM

SIEVE
ext wr $s : \mathbb{F}\,\mathbb{N}_1$
pre $s \subseteq 2\,..\,n$
post $s' = s - C$

Refinement calculus

$SIEVE \mathrel{\widehat{=}}$
$\{s \subseteq 2\,..\,n\}\, s\colon \left[s' = s - C\right]$

Sieve of Eratosthenes - sequential

- Precondition $s \subseteq 2\,..\,n$ holds initially
- Assume that C is the set of all composite numbers (non-primes)
- Postcondition $s' = s - C$

$$
\begin{array}{ll}
 & s\colon \left[s \subseteq 2\,..\,n,\ s' = s - C\right] \\
= & \text{equivalent post condition (set theory)} \\
 & s\colon \left[s \subseteq 2\,..\,n,\ s' \subseteq s \wedge s - s' \subseteq C \wedge s' \cap C = \emptyset\right] \\
\sqsubseteq & \text{guarantee on every step} \\
 & (\mathbf{guar}\ s' \subseteq s \wedge s - s' \subseteq C) \Cap s\colon \left[s \subseteq 2\,..\,n,\ s' \cap C = \emptyset\right]
\end{array}
$$

The guarantee condition is

- reflexive, i.e. $s' = s \Rightarrow s' \subseteq s \wedge s - s' \subseteq C$
- transitive, i.e. $s' \subseteq s'' \subseteq s \wedge s - s'' \subseteq C \wedge s'' - s' \subseteq C \Rightarrow s' \subseteq s \wedge s - s' \subseteq C$

First some set theory

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

$$\begin{aligned} & s' \cap C = \emptyset \\ \equiv\ & s' \cap \bigcup\{j \in \mathbb{N} \mid 2 \le j \cdot c_j\} = \emptyset \\ \equiv\ & \bigcup\{j \in \mathbb{N} \mid 2 \le j \cdot (s' \cap c_j)\} = \emptyset \\ \equiv\ & \forall j \in \mathbb{N} \cdot 2 \le j \Rightarrow s' \cap c_j = \emptyset \end{aligned}$$

Therefore

$$\begin{aligned} & (\mathbf{guar}\ s' \subseteq s \wedge s - s' \subseteq C) \Cap \\ & \quad s\colon \left[s \subseteq 2\,..\,n,\ s' \cap C = \emptyset\right] \\ \sqsubseteq\ & \quad \textcolor{blue}{\text{by above set theory}} \\ & (\mathbf{guar}\ s' \subseteq s \wedge s - s' \subseteq C) \Cap \\ & \quad s\colon \left[s \subseteq 2\,..\,n,\ \forall j \cdot 2 \le j \Rightarrow s' \cap c_j = \emptyset\right] \end{aligned}$$

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

Then some number theory

If $2 \le i \wedge 2 \le j$ and if $i * j \le n$ then either

- $i^2 \le n \wedge j^2 \ge n$ or
- $j^2 \le n \wedge i^2 \ge n$

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

$$\begin{aligned} & s \subseteq 0\,..\,n \wedge n \le i^2 \wedge (\forall j \in 2\,..\,i \cdot s \cap c_j = \emptyset) \\ \Rightarrow\ & \\ & (\forall j \in \mathbb{N} \cdot 2 \le j \Rightarrow s \cap c_j = \emptyset) \end{aligned}$$

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

Introducing a loop

$$\begin{aligned} & s\colon \left[s \subseteq 2\,..\,n,\ \forall j \cdot 2 \le j \Rightarrow s' \cap c_j = \emptyset\right] \\ \sqsubseteq\ & \quad \textcolor{blue}{\text{introduce variable } i \text{ to be used as loop index}} \\ & \mathbf{var}\ i := 1; \\ & i, s\colon \left[\begin{matrix} s \subseteq 2\,..\,n \wedge \\ \forall j \in 2\,..\,i \cdot s \cap c_j = \emptyset \end{matrix},\ \begin{matrix} n < (i+1)^2 \wedge \\ \forall j \in 2\,..\,i \cdot s' \cap c_j = \emptyset \end{matrix}\right] \\ \sqsubseteq\ & \quad \textcolor{blue}{\text{introduce while loop}} \\ & \mathbf{while}(i+1)^2 \le n\,\mathbf{do} \\ & \quad i, s\colon \left[\begin{matrix} s \subseteq 2\,..\,n \wedge (i+1)^2 \le n \wedge \\ \forall j \in 2\,..\,i \cdot s \cap c_j = \emptyset \end{matrix},\ \begin{matrix} i < i' \wedge \\ \forall j \in 2\,..\,i \cdot s' \cap c_j = \emptyset \end{matrix}\right] \end{aligned}$$

Refining the loop body

$$\begin{aligned} & i, s\colon \left[\begin{matrix} s \subseteq 2\,..\,n \wedge (i+1)^2 \le n \wedge \\ \forall j \in 2\,..\,i \cdot s \cap c_j = \emptyset \end{matrix},\ \begin{matrix} i < i' \wedge \\ \forall j \in 2\,..\,i \cdot s' \cap c_j = \emptyset \end{matrix}\right] \\ \sqsubseteq\ & \quad \textcolor{blue}{\text{introduce sequential composition}} \\ & i := i + 1; \\ & s\colon \left[\begin{matrix} s \subseteq 2\,..\,n \wedge i^2 \le n \wedge \\ \forall j \in 2\,..\,i-1 \cdot s \cap c_j = \emptyset \end{matrix},\ \forall j \in 2\,..\,i \cdot s' \cap c_j = \emptyset\right] \end{aligned}$$

Refining the specification:

$$\begin{aligned} & s\colon \left[\begin{matrix} s \subseteq 2\,..\,n \wedge i^2 \le n \wedge \\ \forall j \in 2\,..\,i-1 \cdot s \cap c_j = \emptyset \end{matrix},\ \forall j \in 2\,..\,i \cdot s' \cap c_j = \emptyset\right] \\ \sqsubseteq\ & \quad \textcolor{blue}{\text{to achieve the post condition the elements in } c_i \text{ need to be removed}} \\ & s\colon \left[s \subseteq 2\,..\,n \wedge i^2 \le n,\ s' \cap c_i = \emptyset\right] \\ \sqsubseteq\ & \quad \textcolor{blue}{\text{recall that } c_i \text{ contains all the multiples of } i\text{, excluding } i} \\ & s\colon \left[s \subseteq 2\,..\,n \wedge i^2 \le n,\ \forall j \cdot 2 * i \le j * i \le n \Rightarrow j * i \notin s'\right] \end{aligned}$$

Reminder: this is all in the context of $(\mathbf{guar}\ s' \subseteq s \wedge s - s' \subseteq C)$

Introduce inner loop

$s\colon\left[s \subseteq 2\,..\,n \wedge i^2 \le n,\ \forall j \cdot 2*i \le j*i \le n \Rightarrow j*i \notin s'\right]$

$\sqsubseteq$ introduce variable k to be used as a loop index

var $k := 2$;

$$k, s\colon\left[\begin{array}{l} s \subseteq 2\,..\,n \wedge i^2 \le n \wedge \\ \forall j \cdot 2*i \le j*i < k*i \Rightarrow \\ \quad j*i \notin s \end{array},\ \begin{array}{l} n < k*i \wedge \\ \forall j \cdot 2*i \le j*i < k*i \Rightarrow \\ \quad j*i \notin s' \end{array}\right]$$

$\sqsubseteq$ introduce inner loop

while $k*i \le n$ **do**

$$k, s\colon\left[\begin{array}{l} s \subseteq 2\,..\,n \wedge 2*i \le k*i \le n \wedge \\ \forall j \cdot 2*i \le j*i < k*i \Rightarrow \\ \quad j*i \notin s \end{array},\ \begin{array}{l} k < k' \wedge \\ \forall j \cdot 2*i \le j*i < k*i \Rightarrow \\ \quad j*i \notin s' \end{array}\right]$$

Refine the inner loop body

$$k, s\colon\left[\begin{array}{l} s \subseteq 2\,..\,n \wedge 2*i \le k*i \le n \wedge \\ \forall j \cdot 2*i \le j*i < k*i \Rightarrow \\ \quad j*i \notin s \end{array},\ \begin{array}{l} k < k' \wedge \\ \forall j \cdot 2*i \le j*i < k*i \Rightarrow \\ \quad j*i \notin s' \end{array}\right]$$

$\sqsubseteq$ introduce sequential composition

$$s\colon\left[\begin{array}{l} s \subseteq 2\,..\,n \wedge 2*i \le k*i \le n \wedge \\ \forall j \cdot 2*i \le j*i < k*i \Rightarrow \\ \quad j*i \notin s \end{array},\ \begin{array}{l} \forall j \cdot 2*i \le j*i < (k+1)*i \Rightarrow \\ \quad j*i \notin s' \end{array}\right];$$

$k := k + 1$

Now refine the specification

$$s\colon\left[\begin{array}{l} s \subseteq 2\,..\,n \wedge 2*i \le k*i \le n \wedge \\ \forall j \cdot 2*i \le j*i < k*i \Rightarrow \\ \quad j*i \notin s \end{array},\ \begin{array}{l} \forall j \cdot 2*i \le j*i < (k+1)*i \Rightarrow \\ \quad j*i \notin s' \end{array}\right]$$

$\sqsubseteq$ to achieve the post condition the element $k*i$ must be removed

$s\colon\left[s \subseteq 2\,..\,n \wedge 2*i \le k*i \le n,\ k*i \notin s'\right]$

Bring back the guarantee

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

$(\textbf{guar}\ s' \subseteq s \wedge s - s' \subseteq C) \Cap s\colon\left[s \subseteq 2\,..\,n \wedge 2*i \le k*i \le n,\ k*i \notin s'\right]$

$\sqsubseteq$ strengthen guarantee and weaken precondition

$(\textbf{guar}\ s' \subseteq s \wedge s - s' \subseteq \{k*i\}) \Cap s\colon\left[s \subseteq 0\,..\,n \wedge k*i \in 0\,..\,n,\ k*i \notin s'\right]$

Remove an element from the set

Define

$$Rem(m) \mathrel{\widehat{=}} (\textbf{guar}\ s' \subseteq s \wedge s - s' \subseteq \{m\}) \Cap s\colon\left[s \subseteq 0\,..\,n \wedge m \in 0\,..\,n,\ m \notin s'\right]$$

The code so far is

var $i := 1$;
while $(i+1)^2 \le n$ **do**
  $i := i + 1$;
  var $k := 2$;
  while $k*i \le n$ **do**
    $Rem(k*i)$;
    $k := k + 1$

Data refinement: representing the set as an array of words

- A finite set contained in $0 \mathinner{..} n$ 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 $0 \mathinner{..} ws - 1$
- For a large set one needs a vector v of $\left\lceil \frac{n+1}{ws} \right\rceil$ words
- The function $retr(v)$ retrieves the set represented by v

$$retr(v) \mathrel{\widehat{=}} \{j \in 0 \mathinner{..} n \mid (j \bmod ws) \in v(j \operatorname{div} ws)\}$$

Remove an element from the set

Define

$$Rem(m) \mathrel{\widehat{=}} (\mathbf{guar}\ s' \subseteq s \wedge s - s' \subseteq \{m\}) \Cap s\colon \left[s \subseteq 0 \mathinner{..} n \wedge m \in 0 \mathinner{..} n,\ m \notin s' \right]$$

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

$$\begin{array}{l} (\mathbf{guar}\ retr(v') \subseteq retr(v) \wedge retr(v) - retr(v') \subseteq \{m\}) \Cap \\ v\colon \left[retr(v) \subseteq 0 \mathinner{..} n \wedge m \in 0 \mathinner{..} n,\ m \notin retr(v') \right] \end{array}$$

From the definition of *retr*

$$m \notin retr(v') \Leftrightarrow (m \bmod ws) \notin v'(m \operatorname{div} ws)$$

Hence the specification can be written as

$$\begin{array}{ll} & v\colon \left[retr(v) \subseteq 0 \mathinner{..} n \wedge m \in 0 \mathinner{..} n,\ (m \bmod ws) \notin v'(m \operatorname{div} ws) \right] \\ \sqsubseteq & \\ & v(m \operatorname{div} ws)\colon \left[m \in 0 \mathinner{..} n,\ (m \bmod ws) \notin v'(m \operatorname{div} ws) \right] \end{array}$$

Removing an element from a set represented as a single word

$$\begin{array}{l} RemW(\mathbf{var}\ w : \mathbb{F}(0 \mathinner{..} ws - 1), i : 0 \mathinner{..} ws - 1) \mathrel{\widehat{=}} \\ \quad (\mathbf{guar}\ w' \subseteq w \wedge w - w' \subseteq \{i\}) \Cap \\ \quad w\colon \left[w \subseteq 0 \mathinner{..} ws - 1 \wedge i \in 0 \mathinner{..} ws - 1,\ i \notin w' \right] \end{array}$$

Therefore

$$\begin{array}{ll} & Rem(m) \\ \sqsubseteq & \\ & RemW(v(m \operatorname{div} ws), m \bmod ws) \end{array}$$

RemW can be implemented using bit-wise operations on a word (exercise)

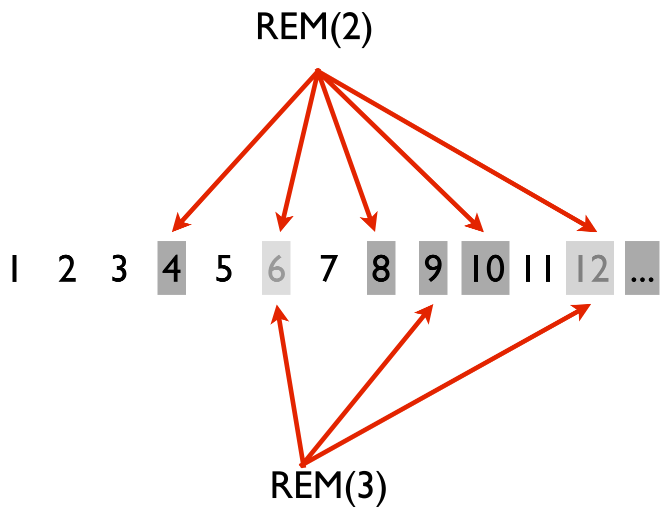
Conclusions

- 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 $RemMults(i)$ iteration is easy $RemMults(i) \triangleq s : [s' = s - c_i]$
- for $Sieve \triangleq ||_i \, RemMults(i)$
  - need the rely of $RemMults(i)$ to be $s' \subseteq s$
  - relax the equality in the postcondition of $RemMults(i)$ to $s' \cap c_i = \emptyset$
  - avoid removing too much with a guarantee of $RemMults(i)$ 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)

$$\underbrace{\overbrace{\sigma_0}^{pre} \quad \cdots \quad \overbrace{\epsilon(\sigma_i\ \sigma_{i+1})}^{rely} \quad \cdots \quad \underbrace{\pi(\sigma_j\ \sigma_{j+1})}_{guar} \quad \cdots \quad \sigma_f}_{post}$$

- 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. $j' = j \vee j' = j + 1$
  - 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

R/G (old)

RemMults(i)
ext wr $s : \mathbb{F}\,\mathbb{N}_1$
pre $s \subseteq 0\,..\,n$
rely $s' \subseteq s$
guar $s' \subseteq s \wedge \cdots$
post $s' = s - c_i$

Proof rules (also used a 5-tuple form)

$$\frac{\begin{array}{l} \{P, R_l\}\ s_l\ \{G_l, Q_l\} \\ \{P, R_r\}\ s_r\ \{G_r, Q_r\} \\ R \vee G_r \Rightarrow R_l \\ R \vee G_l \Rightarrow R_r \\ G_l \vee G_r \Rightarrow G \\ P \wedge Q_l \wedge Q_r \wedge (R \vee G_l \vee G_r)^* \Rightarrow Q \end{array}}{\{P, R\}\ s_l \mid\mid s_r\ \{G, Q\}}$$

R/G rethought

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

R/G (old)

RemMults(i)
ext wr $s : \mathbb{F}\,\mathbb{N}_1$
pre $s \subseteq 0\,..\,n$
rely $s' \subseteq s$
guar $s' \subseteq s \wedge \cdots$
post $s' = s - c_i$

R/G decomposed [?, ?]

$\{s \subseteq 0\,..\,n\}$
guar$(s' \subseteq s \wedge \cdots) \bullet$
rely $s' \subseteq s \bullet$
$s : [s' = s - c_i]$

Now [?]

RemMults$(i : \mathbb{N})$
$\{s \subseteq 0\,..\,n\}$
$($**rely** $s' \subseteq s) \Cap$
$($**guar** $s' \subseteq s \wedge s - s' \subseteq c_i) \Cap$
$s\!: \left[s' \cap c_i = \emptyset\right]$

R/G rethought

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

Distribute-G-seq

$$(\textbf{guar}\ g) \Cap (c\,;d) = ((\textbf{guar}\ g) \Cap c)\,;((\textbf{guar}\ g) \Cap d)$$

Distribute-G-par

$$(\textbf{guar}\ g) \Cap (c \parallel d) = ((\textbf{guar}\ g) \Cap c) \parallel ((\textbf{guar}\ g) \Cap d)$$

Conjunction-mono

$$c_0 \sqsubseteq c_1 \wedge d_0 \sqsubseteq d_1 \quad \Rightarrow \quad c_0 \Cap d_0 \sqsubseteq c_1 \Cap d_1$$

(Some) Laws of the new algebraic R/G

... a few of many!

Conjoin-G: $(\textbf{guar}\ g_1) \Cap (\textbf{guar}\ g_2) = (\textbf{guar}\ g_1 \wedge g_2)$

Strengthen-G: $(\textbf{guar}\ g_1) \sqsubseteq (\textbf{guar}\ g_2)$
if $g_2 \Rightarrow g_1$

Distribute-G: $((\textbf{guar}\ g) \Cap ||_i\, c_i = ||_i(\textbf{guar}\ g) \Cap c_i$

Trading rely, guarantee and post

Trading-R-G-Post: $(\textbf{rely}\ r) \Cap \left[(r \vee g)^* \wedge q\right] \sqsubseteq (\textbf{rely}\ r) \Cap (\textbf{guar}\ g) \Cap \left[q\right]$

Introducing a parallel composition

Intro-multi-Par: $(\textbf{rely}\ r) \Cap \left[\bigwedge_i q_i\right] \quad \sqsubseteq \quad ||_i(\textbf{guar}\ \rho) \Cap (\textbf{rely}\ \rho) \Cap \left[q_i\right]$
if $r \Rightarrow \rho$

(asymmetric version later)

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

$$
\begin{array}{ll}
& (\mathbf{rely}\ s' = s) \Cap s\colon [s' = s - C] \\
= & \text{set theory} \\
& (\mathbf{rely}\ s' = s) \Cap s\colon [s' \subseteq s \wedge s - s' \subseteq C \wedge s' \cap C = \emptyset] \\
\sqsubseteq & \text{by Trading-R-G-Post as } s' \subseteq s \wedge s - s' \subseteq C \text{ is reflexive and transitive} \\
& (\mathbf{guar}\ s' \subseteq s \wedge s - s' \subseteq C) \Cap (\mathbf{rely}\ s' = s) \Cap s\colon [s' \cap C = \emptyset] \\
= & \text{as } s' \cap C = \emptyset \equiv s' \cap \bigcup_i c_i = \emptyset \equiv \bigcup_i (s' \cap c_i) = \emptyset \equiv \forall i \,.\, s' \cap c_i = \emptyset \\
& (\mathbf{guar}\ s' \subseteq s \wedge s - s' \subseteq C) \Cap (\mathbf{rely}\ s' = s) \Cap s\colon [\forall i \cdot s' \cap c_i = \emptyset] \\
\sqsubseteq & \text{by Intro-multi-Par} \\
& (\mathbf{guar}\ s' \subseteq s \wedge s - s' \subseteq C) \Cap (\|_i (\mathbf{guar}\ s' \subseteq s) \Cap (\mathbf{rely}\ s' \subseteq s) \Cap s\colon [s' \cap c_i = \emptyset]) \\
= & \text{Distribute-G and Conjoin-G} \\
& \|_i (\mathbf{guar}\ s' \subseteq s \wedge s - s' \subseteq C) \Cap (\mathbf{rely}\ s' \subseteq s) \Cap s\colon [s' \cap c_i = \emptyset] \\
\sqsubseteq & \text{Strengthen-G} \\
& \|_i (\mathbf{guar}\ s' \subseteq s \wedge s - s' \subseteq c_i) \Cap (\mathbf{rely}\ s' \subseteq s) \Cap s\colon [s' \cap c_i = \emptyset]
\end{array}
$$

Onwards to code

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

$$
\begin{array}{l}
RemMults(i : \mathbb{N}) \\
\quad \{s \subseteq 0..n\} \\
\quad (\mathbf{guar}\ s' \subseteq s \wedge s - s' \subseteq c_i) \Cap (\mathbf{rely}\ s' \subseteq s) \Cap s\colon [s' \cap c_i = \emptyset]
\end{array}
$$

Can be implemented by successively removing each multiple

$$
\begin{array}{l}
\mathbf{var}\ k := 2; \\
\mathbf{while}\ \ k * i \leq n\ \ \mathbf{do} \\
\quad Rem(k * i); \\
\quad k := k + 1
\end{array}
$$

The interesting part is *Rem*. Its specification allows interference that removes elements from s. It guarantees to remove element m, only.

$$
\begin{array}{l}
Rem(m : \mathbb{N}) \\
\quad \{s \subseteq 0\,..\,n \wedge m \in 0\,..\,n\} \\
\quad (\mathbf{guar}\ s' \subseteq s \wedge s - s' \subseteq \{m\}) \Cap (\mathbf{rely}\ s' \subseteq s) \Cap s\colon [m \notin s']
\end{array}
$$

Removing an element from a set atomically

The specification of *Rem* allows interference that removes elements from s. It guarantees to remove element m, only.

$$
\begin{array}{l}
Rem(m : \mathbb{N}) \\
\quad \{s \subseteq 0\,..\,n \wedge m \in 0\,..\,n\} \\
\quad (\mathbf{guar}\ s' \subseteq s \wedge s - s' \subseteq \{m\}) \Cap (\mathbf{rely}\ s' \subseteq s) \Cap s\colon [m \notin s']
\end{array}
$$

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

$$v : \mathbf{array}\ 0..\lceil \tfrac{n+1}{ws} \rceil - 1\ \mathbf{of}\ \mathbb{F}(0\,..\,ws - 1)$$

Representation relation

$$retr(v) = \{j \in 0\,..\,n \mid j \bmod ws \in v(j \operatorname{div} ws)\}$$

Implementation using *RemW* which removes an element from set (as a word)

$$RemW(v(m \operatorname{div} ws), m \bmod ws)$$

Specification

$$
\begin{array}{l}
RemW(\mathbf{var}\ w : \mathbb{F}(0..ws - 1), i : 0..ws - 1) \\
\quad (\mathbf{guar}\ w' \subseteq w \wedge w - w' \subseteq \{i\}) \Cap (\mathbf{rely}\ w' \subseteq w) \Cap w\colon [i \notin w']
\end{array}
$$

Compare and swap

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

$$
\begin{array}{l}
CAS(\mathbf{var}\ w, lw, nw, \mathbf{var}\ done) \mathrel{\widehat{=}} \\
\quad (\mathbf{rely}\ lw' = lw \wedge nw' = nw \wedge done' = done) \Cap \\
\quad w, done : \left\langle \begin{array}{l} (w = lw \Rightarrow w' = nw \wedge done') \wedge \\ (w \neq lw \Rightarrow w' = w \wedge \neg\, done') \end{array} \right\rangle
\end{array}
$$

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

$$
\begin{array}{ll}
& w, done : \left\langle \begin{array}{l} w \subseteq lw \wedge nw = lw - \{i\}, \\ (w = lw \Rightarrow w' = w - \{i\}) \wedge (w \neq lw \Rightarrow w' \subset lw) \end{array} \right\rangle \\
\sqsubseteq & \\
& CAS(w, lw, nw, \_);
\end{array}
$$

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

Introduce loop

$\{w \subseteq 0..ws-1 \wedge i \in 0..ws-1\}$
$(\textbf{guar } w' \subseteq w \wedge w - w' \subseteq \{i\}) \Cap (\textbf{rely } w' \subseteq w) \Cap w\colon [i \notin w']$
$\sqsubseteq$ invariant *true* and variant $w \supset w'$
while $i \in w$ **do**
$(\textbf{guar } w' \subseteq w \wedge w - w' \subseteq \{i\}) \Cap (\textbf{rely } w' \subseteq w) \Cap$
$w\colon [w \supset w' \vee i \notin w']$

Refine loop body

$(\textbf{guar } w' \subseteq w \wedge w - w' \subseteq \{i\}) \Cap (\textbf{rely } w' \subseteq w) \Cap w\colon [w \supset w' \vee i \notin w']$
$\sqsubseteq$ strengthen guarantee, introduce local variable *lw*
var lw $\cdot$
$(\textbf{guar } w' = w \vee w' = w - \{i\}) \Cap (\textbf{rely } w \supseteq w') \Cap$
$(lw\colon [w \supseteq lw' \supseteq w']\,; w\colon [lw \supseteq w\,,\ lw \supset w' \vee i \notin w'])$

Refining the first specification

$(\textbf{guar } w' = w \vee w' = w - \{i\}) \Cap (\textbf{rely } w \supseteq w') \Cap lw\colon [w \supseteq lw' \supseteq w']$
$\sqsubseteq$
$lw := w$

Refining the second specification

$(\textbf{guar } w' = w \vee w' = w - \{i\}) \Cap (\textbf{rely } w \supseteq w') \Cap w\colon [lw \supseteq w\,,\ lw \supset w' \vee i \notin w']$
$\sqsubseteq$ introduce variable *nw* to contain the updated value
var $nw := lw - \{i\};$
$(\textbf{guar } w' = w \vee w' = w - \{i\}) \Cap (\textbf{rely } w \supseteq w') \Cap$
$w\colon [lw \supseteq w \wedge nw = lw - \{i\}\,,\ lw \supset w' \vee i \notin w']$

Introducing a Compare-and-Swap (CAS)

$CAS(\textbf{var } w, lw, nw, \textbf{var } done) \mathrel{\widehat{=}}$
$(\textbf{rely } lw' = lw \wedge nw' = nw \wedge done' = done) \Cap$
$$w, done : \left\langle \begin{array}{l} (w = lw \Rightarrow w' = nw \wedge done') \wedge \\ (w \neq lw \Rightarrow w' = w \wedge \neg\, done') \end{array} \right\rangle$$

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

$(\textbf{guar } w' = w \vee w' = w - \{i\}) \Cap (\textbf{rely } w \supseteq w') \Cap$
$w\colon [lw \supseteq w \wedge nw = lw - \{i\}\,,\ lw \supset w' \vee i \notin w']$
$\sqsubseteq$
$(\textbf{guar } w' = w \vee w' = w - \{i\}) \Cap (\textbf{rely } w \supseteq w') \Cap$
$w\colon [lw \supseteq w \wedge nw = lw - \{i\}\,,\ (lw = w \Rightarrow w' = nw) \wedge (lw \neq w \Rightarrow w' = w)]$
$\sqsubseteq$
$CAS(w, lw, nw, \_)$

Removing an element from a (small) set atomically

Specification

$$RemW(\textbf{var } w : \mathbb{F}(0..ws-1), i : 0..ws-1)$$
$$(\textbf{guar } w' \subseteq w \wedge w - w' \subseteq \{i\}) \Cap (\textbf{rely } w' \subseteq w) \Cap w\!:\!\left[i \notin w'\right]$$

Code

$$\begin{array}{l}
\textbf{while } i \in w \textbf{ do invariant } \mathit{true} \\
\quad \textbf{var } lw := w; \\
\quad \textbf{var } nw := lw - \{i\}; \quad \text{– stable because variables local} \\
\quad CAS(w, lw, nw, _); \quad \text{– refines } w : \left\langle \begin{array}{l} w \subseteq lw \wedge nw = lw - \{i\}, \\ (w = lw \Rightarrow w' \subseteq w - \{i\}) \wedge \\ (w \neq lw \Rightarrow w' \subset lw) \end{array} \right\rangle \\
\{i \notin w\}
\end{array}$$

Termination

Code

$$\begin{array}{l}
\textbf{while } i \in w \textbf{ do invariant } \mathit{true} \textbf{ wf-relation } (w' \subset w) \; \mathit{OR} \; (\#w' < \#w) \\
\quad \textbf{var } lw := w; \\
\quad \textbf{var } nw := lw - \{i\}; \quad \text{– stable because variables local} \\
\quad CAS(w, lw, nw, _); \quad \text{– refines } w : \left\langle \begin{array}{l} lw \subseteq w \wedge nw = lw - \{i\}, \\ (w = lw \Rightarrow w' \subseteq w - \{i\}) \wedge \\ (w \neq lw \Rightarrow w' \subset lw) \end{array} \right\rangle \\
\{i \notin w\}
\end{array}$$

Termination

- If the CAS succeeds, $i \notin w$ and the loop terminates
- If the CAS fails, $w' \subset w$ and the hence the loop variant decreases

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.

$$\textbf{with } x \textbf{ do } c \ \widehat{=}\ \textbf{idle} \,;((\textbf{demand } x' = x) \Cap c)\,; \textbf{idle}$$
$$\textbf{with } x \textbf{ do } c \ \widehat{=}\ \langle \mathrm{id} \rangle^{\omega} \,;((\textbf{demand } x' = x) \Cap c)\,; \textbf{idle}$$

$\boxed{\text{This allows id steps forever, even when } x \text{ isn't in use elsewhere.}}$

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

$$\textbf{await } b \ \widehat{=}\ [[\neg\, b]]^{\omega} \,; [[b]]$$

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

$$\textbf{await } b = \textbf{while} \neg\, b \textbf{ do nil}$$

Invariant under interference

For a rely relation r and predicate p, r maintains p if

$$r \ \Rrightarrow\ (p \Rightarrow p')$$

Examples: for integer x, sets s, and sequence buf

$$\begin{array}{rcl} x \leq x' & \Rrightarrow & (0 \leq x \Rightarrow 0 \leq x') \\ x = x' & \Rrightarrow & (0 \leq x \Rightarrow 0 \leq x') \\ s \supseteq s' & \Rrightarrow & (s \subseteq 0..n \Rightarrow s' \subseteq 0..n) \\ s = s' & \Rrightarrow & (s \subseteq 0..n \Rightarrow s' \subseteq 0..n) \\ buf' \text{ suffix } buf & \Rrightarrow & (\#buf < N \Rightarrow \#buf' < N) \\ buf \text{ prefix } buf' & \Rrightarrow & (\#buf \neq 0 \Rightarrow \#buf' \neq 0) \end{array}$$

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. $r \Rrightarrow (p \Rightarrow p')$, then

$$(\textbf{rely } r) \Cap \left[p\,,\ r^* \wedge p'\right] \ \sqsubseteq\ \textbf{idle}$$

For example, the rely condition $(buf' \text{ suffix } buf)$ maintains $\#buf < N$, and hence

$$(\textbf{rely } r) \Cap \left[\#buf < N\,,\ buf' \text{ suffix } buf \wedge \#buf' < N\right] \ \sqsubseteq\ \textbf{idle}$$

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

$$(\textbf{rely } r) \Cap \left[p\,,\ r^* \wedge p' \wedge b'\right] \ \sqsubseteq\ \textbf{await } b$$

Multi-place buffer of size N

module *Buffer*
var $buf : \mathrm{seq}\ Value$
invariant $\#buf \leq N$
initially $buf = [\,]$

$write(v : Value)$
rely buf' suffix $buf \Cap$ – single writer
guar buf prefix $buf' \Cap$
with buf **await** $\#buf < N$ **do**
$buf : \left[buf' = buf \frown [v]\right]$

$read()res : Value$
rely buf prefix $buf' \Cap$ – single reader
guar buf' suffix $buf \Cap$
with buf **await** $\#buf \neq 0$ **do**
$res, buf : \left[buf = [res] \frown buf'\right]$

Initial refinement of write

$write(v : Value)$
rely buf' suffix buf ⋒ – single writer
guar buf prefix buf' ⋒
with buf **await** $\#buf < N$ **do**
  $buf : \left[buf' = buf \frown [v] \right]$
$\sqsubseteq$
rely buf' suffix buf ⋒
guar buf prefix buf' ⋒
 await $\#buf < N$; – await buffer not full – stable under rely
 with buf **do**
  $buf : \left[\#buf < N\,,\ buf' = buf \frown [v] \right]$

Initial refinement of read

$read()res : Value$
rely buf prefix buf' ⋒ – single reader
guar buf' suffix buf ⋒
with buf **await** $\#buf \neq 0$ **do**
  $res, buf : \left[buf = [res] \frown buf' \right]$
$\sqsubseteq$
rely buf prefix buf' ⋒
guar buf' suffix buf ⋒
 await $\#buf \neq 0$; – await buffer not empty – stable under rely
 $res : \left[res' = hd(buf) \right]$
 with buf **do**
  $buf : \left[\#buf \neq 0\,,\ buf' = tl(buf) \right]$

Multi-place buffer implementation

The buffer b has $N + 1$ slots but one is always unused. We define the notation $a \oplus b = (a + b) \bmod (N + 1)$. 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

$$retr(b, r, w) = \textbf{if } r = w \textbf{ then } [\,] \textbf{ else } [b[r]] \frown retr(b, r \oplus 1, w)$$

module *BufferI* **implements** *Buffer*
var $b : (0 \,..\, N) \to Value$;
 $r, w : 0 \,..\, N$;
initially $r = 0 \wedge w = 0$;
representation $buf = retr(b, r, w)$

Write in a circular buffer

$write(v : Value)$
rely buf' suffix buf ⋒
guar buf prefix buf' ⋒
await $\#buf < N$; – await buffer not full – stable under rely
with buf **do** $buf : \left[\#buf < N\,,\ buf' = buf \frown [v] \right]$ – atomic update of buf

is data refined by

rely $w' = w \wedge b' = b \wedge (r = w \Rightarrow r' = r)$ ⋒
guar $r' = r \wedge (r = w \oplus 1 \Rightarrow w' = w) \wedge retr(b, r, w)$ prefix $retr(b', r', w')$ ⋒
var $nw := w \oplus 1$;
 await $\langle r \rangle \neq nw$; – await buffer not full – stable under rely
 $b[w] := v$;
  – Ensure $b[w]$ is flushed before updating w
 with w **do** $w := nw$ – atomic update of w

Read in circular buffer

$read()res : Value$
rely buf prefix $buf' \Cap$
guar buf' suffix $buf \Cap$
await $\#buf \neq 0$; – await buffer not empty – stable under rely
$res\colon [res' = hd(buf)]$
with buf **do** $buf\colon [\#buf \neq 0\,,\ buf' = tl(buf)]$ – atomic update of *buf*

is data refined by

rely $r' = r \wedge (r = w \oplus 1 \Rightarrow w' = w) \wedge retr(b, r, w)$ prefix $retr(b', r', w') \Cap$
guar $w' = w \wedge b' = b \wedge (r = w \Rightarrow r' = r) \Cap$
await $r \neq \langle w \rangle$; – await buffer non-empty – stable under rely
$res := b[r]$;
var $nr := r \oplus 1$;
 – Ensure $b[r]$ has been fully read before updating r
 with r **do** $r := nr$ – atomic update of r

Multi-place buffer implementation with size

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

- if $s = 0$ the buffer is empty and
- if $s = N$ 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.

$$retr\_r(b, r, s) = (\lambda\, i \in 0 \,..\, s - 1 \cdot b[(r + i) \bmod N])$$
$$retr\_w(b, w, s) = (\lambda\, i \in 0 \,..\, s - 1 \cdot b[(w + i + n - s) \bmod N])$$

module *BufferI* **implements** *Buffer*
var $b : (0 \,..\, N - 1) \rightarrow Value$;
 $r, w : 0 \,..\, N - 1$;
 $s : 0 \,..\, N$;
initially $s = 0 \wedge r = 0 \wedge w = 0$;
representation $buf = retr\_r(b, r, s) = retr\_w(b, w, s)$

Write in a circular buffer with size

$write(v : Value)$
rely buf' suffix $buf \Cap$
await $\#buf < N$; – stable under rely
with buf **do** $buf\colon [\#buf < N\,,\ buf' = buf \frown [v]]$

is data refined using representation $buf = retr\_w(b, w, s)$ by

rely $w' = w \wedge b' = b \wedge 0 \leq s' \leq s \Cap$
guar $r' = r \wedge s \leq s' \leq N \wedge retr\_r(b, r, s)$ prefix $retr\_r(b', r', s') \Cap$
await $\langle s \rangle < N$; – await buffer not full – stable under rely
$b[w] := v$;
 – Ensure $b[w]$ is flushed before updating s
$(w := (w + 1) \bmod N \parallel$ **with** s **do** $s := s + 1)$ – atomic update of s

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

Read in circular buffer with size

$read()res : Value$
rely buf prefix $buf' \Cap$
await $\#buf \neq 0$; – stable under rely
$res\colon [res' = hd(buf)]$
with buf **do** $buf\colon [\#buf \neq 0\,,\ buf' = tl(buf)]$

is data refined using representation $buf = retr\_r(b, r, s)$ by

rely $r' = r \wedge s \leq s' \leq N \wedge retr\_r(b, r, s)$ prefix $retr\_r(b', r', s') \Cap$
guar $w' = w \wedge b' = b \wedge 0 \leq s' \leq s \Cap$
await $\langle s \rangle \neq 0$; – await buffer non-empty – stable under rely
$res := b[r]$;
 – Ensure $b[r]$ has been fully read before updating s or r
$(r := (r + 1) \bmod N \parallel$ **with** s **do** $s := s - 1)$ – atomic update of s

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

Find least first element of an array that satisfies P

The objective is, given an array v with indices in the range $0 \mathrel{..} N-1$, 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.

$$findp \mathrel{\widehat{=}} t\colon \left[(t' = N \lor satp(v,t')) \land notp(v, 0 \mathrel{..} N-1, t')\right] \lhd$$

where

$$\begin{array}{rcl} satp(v,t) & \mathrel{\widehat{=}} & t \in 0 \mathrel{..} N-1 \land P(v(t)) \\ notp(v,s,t) & \mathrel{\widehat{=}} & (\forall i \in s \cdot i < t \Rightarrow \neg P(v(i))) \end{array}$$

[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).

Specification

$$findp \mathrel{\widehat{=}} (\textbf{rely}\ v' = v \land t' = t) \Cap t\colon \left[(t' = N \lor satp(v,t')) \land notp(v, 0 \mathrel{..} N-1, t')\right]$$

Representing the result using two variables

Two variables ot and et are introduced with the intention that on termination the minimum of ot and et will be the least index satisfying p.

$$\begin{array}{l} (\textbf{rely}\ v' = v \land t' = t) \Cap t\colon \left[(t' = N \lor satp(v,t')) \land notp(v, 0 \mathrel{..} N-1, t')\right] \\ \sqsubseteq \quad \text{by Law variable-rely-guarantee for } ot \text{ and } et \\ \quad \textbf{var}\ ot, et \cdot \\ \qquad (\textbf{rely}\ v' = v \land t' = t \land ot' = ot \land et' = et) \Cap \\ \qquad ot, et, t\colon \left[\begin{array}{l} (min(ot',et') = N \lor satp(v, min(ot',et'))) \land \\ notp(v, 0 \mathrel{..} N-1, min(ot',et')) \end{array}\right]; \lhd \\ \qquad t := min(ot', et') \end{array}$$

Using a guarantee invariant

A guarantee invariant is a guarantee that states a predicate p is invariant.

$$(\textbf{guar-inv}\ p) \mathrel{\widehat{=}} (\textbf{guar}\ p \Rightarrow p')$$

A guarantee invariant of

$$min(ot, et) = N \lor satp(v, min(ot, et)) \tag{1}$$

can be employed; the invariant is established by setting both ot and et to N.

$$\begin{array}{l} (\textbf{rely}\ v' = v \land ot' = ot \land et' = et) \Cap \\ ot, et\colon \left[\begin{array}{l} (min(ot',et') = N \lor satp(v, min(ot',et'))) \land \\ notp(v, 0 \mathrel{..} N-1, min(ot',et')) \end{array}\right]; \lhd \\ \sqsubseteq \quad \text{by Law trade-rely-guarantee-invariant; Law rely-sequential} \\ ot := N; et := N; \\ ((\textbf{guar-inv}\ min(ot,et) = N \lor satp(v, min(ot,et)))) \Cap \\ (\textbf{rely}\ v' = v \land ot' = ot \land et' = et) \Cap \\ ot, et\colon \left[notp(v, 0 \mathrel{..} N-1, min(ot',et'))\right] \lhd \end{array}$$

Concurrency

The motivation for the parallel algorithm comes from the observation that the set of indices to be searched, $0\,..\,N-1$, can be partitioned into the odd and even indices, namely $evens(N)$ and $odds(N)$, respectively, which can be searched in parallel.

$$\begin{array}{l} notp(v, odds(N), min(ot', et')) \wedge notp(v, evens(N), min(ot', et')) \Rrightarrow \\ \quad notp(v, 0\,..\,N-1, min(ot', et')) \end{array}$$

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

$$\begin{array}{ll} & (\textbf{rely}\ v' = v \wedge ot' = ot \wedge et' = et) \Cap \\ & ot, et\colon \left[notp(v, 0\,..\,N-1, min(ot', et'))\right] \\ \sqsubseteq & \text{by Law introduce-parallel-spec-weaken-rely} \\ & (\textbf{guar}\ ot' \leq ot \wedge et' = et) \Cap (\textbf{rely}\ et' \leq et \wedge ot' = ot \wedge v' = v) \Cap \\ & ot, et\colon \left[notp(v, odds(N), min(ot', et'))\right] \lhd \\ & \parallel \\ & (\textbf{guar}\ et' \leq et \wedge ot' = ot) \Cap (\textbf{rely}\ ot' \leq ot \wedge et' = et \wedge v' = v) \Cap \\ & ot, et\colon \left[notp(v, evens(N), min(ot', et'))\right] \end{array}$$

Refining the branches to code

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

$$\begin{array}{ll} & (\textbf{guar}\ ot' \leq ot \wedge et' = et) \Cap (\textbf{rely}\ et' \leq et \wedge ot' = ot \wedge v' = v) \Cap \\ & ot, et\colon \left[notp(v, odds(N), min(ot', et'))\right] \\ = & \\ & (\textbf{guar}\ ot' \leq ot) \Cap (\textbf{rely}\ et' \leq et \wedge ot' = ot \wedge v' = v) \Cap \\ & ot\colon \left[notp(v, odds(N), min(ot', et'))\right] \end{array}$$

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.

$$\begin{array}{ll} & (\textbf{guar}\ ot' \leq ot) \Cap (\textbf{rely}\ et' \leq et \wedge ot' = ot \wedge v' = v) \Cap \\ & ot\colon \left[notp(v, odds(N), min(ot', et'))\right] \lhd \\ \sqsubseteq & \text{by Law variable-rely-guarantee for } oc \\ & \textbf{var}\ oc \cdot \\ & \quad (\textbf{rely}\ et' \leq et \wedge oc' = oc \wedge ot' = ot \wedge v' = v) \Cap \\ & \quad oc, ot\colon \left[notp(v, odds(N), min(ot', et'))\right] \lhd \end{array}$$

At this point a guarantee invariant

$$notp(v, odds(N), oc) \wedge bnd(oc, N) \tag{2}$$

is introduced where the bounding conditions on oc follow.

$$bnd(oc, N) \mathrel{\widehat{=}} 1 \leq oc \leq N + 1$$

This guarantee invariant is established by setting oc to one.

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.

$$\begin{array}{ll} & (\textbf{rely}\ et' \leq et \wedge oc' = oc \wedge ot' = ot \wedge v' = v) \Cap \\ & \quad oc, ot\colon \left[notp(v, odds(N), min(ot', et'))\right] \\ \sqsubseteq & \text{Laws rely-sequential, trade-rely-guarantee-invariant, assignment-rely-guarantee} \\ & oc := 1; \\ & (\textbf{guar-inv}\ notp(v, odds(N), oc) \wedge bnd(oc, N)) \Cap \\ & \quad (\textbf{rely}\ et' \leq et \wedge oc' = oc \wedge ot' = ot \wedge v' = v) \Cap \\ & \quad oc, ot\colon \left[oc' \geq min(ot', et')\right] \lhd \end{array}$$

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 $\_ \succ \_$ on T
- a boolean expression b and predicates b_0 and b_1

if

- p is r-stable, i.e. $r \Rrightarrow (p \Rightarrow p')$
- $\_ \succ \_$ is well-founded on p, i.e. $p \lhd (\_ \succ \_)$ is well-founded
- v is non-increasing under r on p, i.e. $p \wedge r \Rrightarrow v' \preceq v$
- b is single reference, i.e. it has only a single reference to a non-stable variable
- $p \wedge b \Rrightarrow b_0$ and $p \wedge r \Rrightarrow (b_0 \Rightarrow b_0')$
- $p \wedge \neg\, b \Rrightarrow b_1$ and $p \wedge r \Rrightarrow (b_1 \Rightarrow b_1')$

then

$$
\begin{array}{l}
\quad (\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge b_1' \wedge v' \preceq v\right] \\
\sqsubseteq \textbf{while}\ b\ \textbf{do}((\textbf{rely}\ r) \Cap \left[p \wedge b_0\,,\ p' \wedge v' \prec v\right]
\end{array}
$$

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 $oc \geq ot \vee oc \geq et$ is preserved by the rely condition as decreasing et will not falsify it. Hence b_1 is $oc \geq ot \vee oc \geq et$, which ensures $oc \geq min(ot, et)$ as required. For loop termination a well-founded relation reducing the variant $ot - oc$ is used.

$$
\begin{array}{l}
\quad (\textbf{rely}\ et' \leq et \wedge oc' = oc \wedge ot' = ot \wedge v' = v) \Cap \\
\qquad oc, ot\colon \left[oc' \geq min(ot', et')\right] \\
\sqsubseteq \quad \text{by Law rely-loop} \\
\quad \textbf{while}\ oc < ot \wedge oc < et\ \textbf{do} \\
\qquad (\textbf{rely}\ et' \leq et \wedge oc' = oc \wedge ot' = ot \wedge v' = v) \Cap \\
\qquad\quad oc, ot\colon \left[oc < ot\,,\ -1 \leq ot' - oc' < ot - oc\right] \lhd
\end{array}
$$

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

$$
\begin{array}{l}
\quad (\textbf{rely}\ et' \leq et \wedge oc' = oc \wedge ot' = ot \wedge v' = v) \Cap \\
\qquad oc, ot\colon \left[oc < ot\,,\ -1 \leq ot' - oc' < ot - oc\right] \\
\sqsubseteq \quad \text{by Law weaken-rely} \\
\quad (\textbf{rely}\ oc' = oc \wedge ot' = ot \wedge v' = v) \Cap \\
\qquad oc, ot\colon \left[oc < ot\,,\ -1 \leq ot' - oc' < ot - oc\right] \lhd
\end{array}
$$

At this stage we bring back in the guarantee invariants introduced above. The refinement is now uses Law rely-conditional.

$$
\begin{array}{l}
\quad (\textbf{guar-inv}\ min(ot, et) = N \vee satp(v, min(ot, et))) \Cap \\
\quad (\textbf{guar-inv}\ notp(v, odds(N), oc) \wedge bnd(oc, N)) \Cap \\
\quad (\textbf{rely}\ oc' = oc \wedge ot' = ot \wedge v' = v) \Cap \\
\qquad oc, ot\colon \left[oc < ot\,,\ -1 \leq ot' - oc' < ot - oc\right] \\
\sqsubseteq \\
\quad \textbf{if}\ P(v(oc))\textbf{then} \\
\qquad (\textbf{guar-inv}\ min(ot, et) = N \vee satp(v, min(ot, et))) \Cap \\
\qquad (\textbf{guar-inv}\ notp(v, odds(N), oc) \wedge bnd(oc, N)) \Cap \\
\qquad (\textbf{rely}\ oc' = oc \wedge ot' = ot \wedge v' = v) \Cap \\
\qquad\quad oc, ot\colon \left[P(v(oc)) \wedge oc < ot\,,\ -1 \leq ot' - oc' < ot - oc\right] \\
\quad \textbf{else} \\
\qquad (\textbf{guar-inv}\ min(ot, et) = N \vee satp(v, min(ot, et))) \Cap \\
\qquad (\textbf{guar-inv}\ notp(v, odds(N), oc) \wedge bnd(oc, N)) \Cap \\
\qquad (\textbf{rely}\ oc' = oc \wedge ot' = ot \wedge v' = v) \Cap \\
\qquad\quad oc, ot\colon \left[\neg\, P(v(oc)) \wedge oc < ot\,,\ -1 \leq ot' - oc' < ot - oc\right]
\end{array}
$$

Finally, Law assignment-rely-guarantee can be applied to each of the branches. Each assignment ensures the guarantee invariant $(min(ot, et) = N \vee satp(v, min(ot, et)) \wedge notp(v, odds(N), oc) \wedge bnd(oc, N)$ is maintained.

$$\sqsubseteq \textbf{if } P(v(oc)) \textbf{ then } ot := oc \textbf{ else } oc := oc + 2$$

Collected code

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.

$$\begin{array}{l}
\textbf{var } ot, et \cdot \\
ot := N; \\
et := N; \\
\left(\begin{array}{l}
\textbf{var } oc \cdot \\
oc := 1; \\
\textbf{while } oc < ot \wedge oc < et \textbf{ do} \\
\quad \textbf{if } P(v(oc)) \textbf{ then } ot := oc \\
\quad\quad \textbf{else } oc := oc + 2
\end{array} \;\middle\|\; \begin{array}{l}
\textbf{var } ec \cdot \\
ec := 0; \\
\textbf{while } ec < ot \wedge ec < et \textbf{ do} \\
\quad \textbf{if } P(v(ec)) \textbf{ then } et := ec \\
\quad\quad \textbf{else } ec := ec + 2
\end{array} \right); \\
t := min(ot, et)
\end{array}$$

Treiber stack

Abstract state is a sequence of values

$$\textbf{var } A : \mathrm{seq}\ Val$$

Specification uses atomic step style

$$\begin{array}{l}
Push(v : Val) \\
\langle \mathrm{id} \rangle^{\omega} ; A : \left\langle A' = [v] \frown A \right\rangle ; \langle \mathrm{id} \rangle^{*} \\
\sqcup \\
\textbf{rely } A' = A \Cap (\langle \mathrm{id} \rangle^{*} ; A : \left\langle A' = [v] \frown A \right\rangle ; \langle \mathrm{id} \rangle^{*})
\end{array}$$

$$\begin{array}{l}
Pop() r : [Val] \\
\langle \mathrm{id} \rangle^{\omega} ; A, r : \left\langle A = [r'] \frown A' \vee (A = [\,] = A' \wedge r' = null) \right\rangle ; \langle \mathrm{id} \rangle^{*} \\
\sqcup \\
\textbf{rely } A' = A \Cap (\langle \mathrm{id} \rangle^{*} ; A, r : \left\langle A = [r'] \frown A' \vee (A = [\,] = A' \wedge r' = null) \right\rangle ; \langle \mathrm{id} \rangle^{*})
\end{array}$$

Treiber stack representation

Representation as a linked list

$$\begin{array}{l}
\textbf{type } Node = \{data : Val;\ next : *Node\} \\
\textbf{var } s : *Node
\end{array}$$

Abstraction relation

$$\begin{array}{l}
stack(s : *Node, A : \mathrm{seq}\ Val) = \\
\quad (s = \textbf{null} \wedge A = [\,]) \vee \\
\quad (\exists\, v, n \cdot s \mapsto Node(v, n) \wedge head(A) = v \wedge stack(n, tail(A)))
\end{array}$$

Repeat

Repeat statement semantics

$$\textbf{repeat}\ c\ \textbf{until}\ b = (\langle \mathrm{id} \rangle^{*} ; c ; [[\neg\, b]])^{\omega} ; \langle \mathrm{id} \rangle^{*} ; c ; [[b]]$$

Push specification (possibly nonterminating)

$$\begin{aligned}&\langle \mathrm{id} \rangle^{\omega} ; A : \left\langle A' = [v] \frown A \right\rangle ; \langle \mathrm{id} \rangle^{*} \\ &= (\langle \mathrm{id} \rangle^{*})^{\omega} ; \langle \mathrm{id} \rangle^{*} ; A : \left\langle A' = [v] \frown A \right\rangle ; \langle \mathrm{id} \rangle^{*}\end{aligned}$$

To implement this specification as a repeat statement, we want

$$\begin{aligned}\langle \mathrm{id} \rangle^{*} &\sqsubseteq \langle \mathrm{id} \rangle^{*} ; c ; [[\neg\, b]] \\ \langle \mathrm{id} \rangle^{*} ; A : \left\langle A' = [v] \frown A \right\rangle ; \langle \mathrm{id} \rangle^{*} &\sqsubseteq \langle \mathrm{id} \rangle^{*} ; c ; [[b]]\end{aligned}$$

but needs change of representation as well

Implementation

$Push(v : Val)$
var $x : *Node;$
$x := new\ Node();$
$x \to data := v;$
$\{stack(s, A) * x \mapsto Node(v, \_)\};$
var $done : \mathbb{B};$
repeat
  var $t : *Node;$
  $\langle t := s \rangle ;$
  $x \to next := t;$
  $\{stack(s, A) * (x \mapsto Node(v, t))\}$
  $CAS(s, t, x, done)$
until $done$

Matching the refinement conditions

$$\begin{aligned}&x, done : \langle \mathrm{true} \rangle^{*} \\ &\sqsubseteq \\ &\langle \mathrm{id} \rangle^{*} ; \mathbf{var}\ t : *Node;\ \langle t := s \rangle ;\ x.next := t;\ CAS(s, t, x, done) ; [[\neg\, done]]\end{aligned}$$

$$\begin{aligned}&x, done : \langle \mathrm{true} \rangle^{*} ; x, s, done : \left\langle \exists A, A' \cdot \begin{array}{l} stack(s, A) \land \\ A' = [v] \frown A \land \\ stack(s', A') \end{array} \right\rangle ; \langle \mathrm{id} \rangle^{*} \\ &\sqsubseteq \\ &\langle \mathrm{id} \rangle^{*} ; \mathbf{var}\ t : *Node;\ \langle t := s \rangle ;\ x.next := t;\ CAS(s, t, x, done) ; [[done]]\end{aligned}$$

Overview

Theory for rely/guarantee concurrency motivated by

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

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 $\oplus : S \times S \to S$
  - 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
  - $(\mathbb{N}, +, 0)$
  - $(\mathbb{N}, *, 1)$
  - $(\mathit{Programs}, \mathbin{;}, \mathbf{nil})$
  - $(\mathit{Programs}, \parallel, \mathbf{skip})$
  - $(\mathit{Programs}, \Cap, \mathbf{chaos})$
- All except $(\mathit{Programs}, \mathbin{;}, \mathbf{nil})$ are commutative monoids
  - Commutative: $x_0 \oplus x_1 = x_1 \oplus x_0$

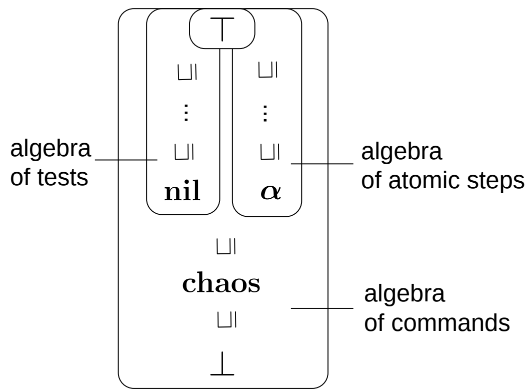
Kleene algebra - the algebra of regular expressions

	Regular expressions	Relations	Programs
Alternatives	$e_0 \mid e_1$	$r_0 \cup r_1$	$c_0 \sqcap c_1$
Sequence	$e_0\, e_1$	$r_0 \fatsemi r_1$	$c_0 \mathbin{;} c_1$
Kleene star	e^*	r^*	c^*
Identity of sequence	ϵ	id	$\mathbf{nil}$
Identity of alternation	$\emptyset$	$\emptyset$	$\top$
Basic elements	a	(x, y)	$\Pi(\sigma_0, \sigma_1)$
			$\mathcal{E}(\sigma_0, \sigma_1)$

where a is a symbol; x and y are elements of the base type of the relation; and σ_0 and σ_1 are program states.

Structure of concurrent program algebra

- Concurrent refinement algebra $(\sqcap, \sqcup, \mathbin{;}, \parallel, \Cap)$
- 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

$c \sqcap d$ non-deterministic choice (lattice infimum or meet)

$$(c_0 \sqcap c_1) \sqcap c_2 = c_0 \sqcap (c_1 \sqcap c_2) \quad \text{– associative}$$
$$c_0 \sqcap c_1 = c_1 \sqcap c_0 \quad \text{– commutative}$$
$$c \sqcap c = c \quad \text{– idempotent}$$
$$c \sqcap \top = c = \top \sqcap c \quad \text{– identity } \top$$

$c \sqcup d$ lattice supremum or join

- associative, commutative, idempotent, identity $\bot$

$c \parallel d$ parallel composition

- associative, commutative, identity **skip**

$c \Cap d$ weak conjunction

- associative, commutative, idempotent, identity **chaos**

$c \,;\, d$ sequential composition (sometimes elided to $c\,d$ below)

- associative, identity **nil**

$\sqcap$ and $\sqcup$ have the same precedence, which is lower than $\parallel$ and $\Cap$, which are lower than ;

Complete lattice

For any set of commands C

- $\bigsqcap C$ is the infimum (greatest lower bound) of the set of commands
- $\bigsqcup C$ is the supremum (least upper bound) of the set of commands

Aczel traces

Represent

- a program doing a step from σ_0 to σ_1 by $\Pi(\sigma_0, \sigma_1)$ and
- its environment doing a step from σ_0 to σ_1 by $\mathcal{E}(\sigma_0, \sigma_1)$.

Every step of parallel synchronises steps of the two processes

$$\begin{array}{rl}\mathcal{E}(\sigma_0,\sigma_1),\Pi(\sigma_1,\sigma_2),\mathcal{E}(\sigma_2,\sigma_3),\mathcal{E}(\sigma_3,\sigma_4),\mathcal{E}(\sigma_4,\sigma_5) & \parallel \\ \mathcal{E}(\sigma_0,\sigma_1),\mathcal{E}(\sigma_1,\sigma_2),\Pi(\sigma_2,\sigma_3),\mathcal{E}(\sigma_3,\sigma_4),\Pi(\sigma_4,\sigma_5) & = \\ \mathcal{E}(\sigma_0,\sigma_1),\Pi(\sigma_1,\sigma_2),\Pi(\sigma_2,\sigma_3),\mathcal{E}(\sigma_3,\sigma_4),\Pi(\sigma_4,\sigma_5) & \end{array}$$

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

$$\begin{array}{rl}\mathcal{E}(\sigma_0,\sigma_1),\Pi(\sigma_1,\sigma_2),\mathcal{E}(\sigma_2,\sigma_3),\mathcal{E}(\sigma_3,\sigma_4),\Pi(\sigma_4,\sigma_5) & \Cap \\ \mathcal{E}(\sigma_0,\sigma_1),\Pi(\sigma_1,\sigma_2),\mathcal{E}(\sigma_2,\sigma_3),\mathcal{E}(\sigma_3,\sigma_4),\Pi(\sigma_4,\sigma_5) & = \\ \mathcal{E}(\sigma_0,\sigma_1),\Pi(\sigma_1,\sigma_2),\mathcal{E}(\sigma_2,\sigma_3),\mathcal{E}(\sigma_3,\sigma_4),\Pi(\sigma_4,\sigma_5) & \end{array}$$

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,

- $\pi(\mathrm{id})$ is a single stuttering program step (id is the identity relation)
- $\pi = \pi(\text{univ})$ can perform any single program step (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

$$\begin{aligned} \pi(r_0) \sqcap \pi(r_1) &= \pi(r_0 \cup r_1) \\ \pi(r_0) \sqcup \pi(r_1) &= \pi(r_0 \cap r_1) \\ !\,\pi(r) &= \pi(\overline{r}) \sqcap \epsilon \end{aligned}$$

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,

- $\tau(\Sigma) = \textbf{nil}$
- $\tau(\emptyset) = \top$
- $\tau(p_1) \sqcap \tau(p_2) = \tau(p_1 \cup p_2)$
- $\tau(p_1) \sqcup \tau(p_2) = \tau(p_1) ; \tau(p_2) = \tau(p_1) \parallel \tau(p_2) = \tau(p_1 \cap p_2)$
- $\neg\, \tau(p) = \tau(\overline{p})$

Assertions/preconditions: for a test t

- $\textbf{pre}\ t = t \sqcap \neg\, t ; \bot$
- $\{p\} = \textbf{pre}\ \tau(p) = \tau(p) \sqcap \tau(\overline{p}) ; \bot$

Assumptions

For a an atomic step command

- $\textbf{assume}\ a = a \sqcap (!\, a) ; \bot$
- $!(\pi(r_0) \sqcap \epsilon(r_1)) = \pi(\overline{r_0}) \sqcap \epsilon(\overline{r_1})$
- $!(\pi \sqcap \epsilon(r)) = \pi(\emptyset) \sqcap \epsilon(\overline{r}) = \top \sqcap \epsilon(\overline{r}) = \epsilon(\overline{r})$
- $\textbf{assume}\ \pi \sqcap \epsilon(r) = \pi \sqcap \epsilon(r) \sqcap \epsilon(\overline{r}) ; \bot$

Note that program and environment steps partition atomic steps

Synchonise atomic steps

For atomic commands a and b (think π and ϵ commands) and arbitrary commands c and d

$$
\begin{aligned}
(a ; c) \Cap (b ; d) &= (a \Cap b) ; (c \Cap d) \\
(a ; c) \Cap \textbf{nil} &= \top \\
a \Cap \bot &= \bot
\end{aligned}
$$

Laws

$$
\begin{aligned}
a^* \Cap b^* &= (a \Cap b)^* \\
a^* ; c \Cap b^* ; d &= (a \Cap b)^* ((c \Cap d) \sqcap (a ; a^* ; c \Cap d) \sqcap (c \Cap b ; b^* ; d))
\end{aligned}
$$

For program and environment steps

$$
\begin{aligned}
\pi(r_1) \parallel \pi(r_2) &= \top \\
\pi(r_1) \parallel \epsilon(r_2) &= \pi(r_1 \cap r_2) \\
\epsilon(r_1) \parallel \epsilon(r_2) &= \epsilon(r_1 \cap r_2) \\
\pi(r) \parallel \bot &= \bot \\
\epsilon(r) \parallel \bot &= \bot
\end{aligned}
$$

$$
\begin{aligned}
\pi(r_1) \Cap \pi(r_2) &= \pi(r_1 \cap r_2) \\
\pi(r_1) \Cap \epsilon(r_2) &= \top \\
\epsilon(r_1) \Cap \epsilon(r_2) &= \epsilon(r_1 \cap r_2) \\
\pi(r) \Cap \bot &= \bot \\
\epsilon(r) \Cap \bot &= \bot
\end{aligned}
$$

Interchange laws

Weak conjunction interchange sequential

$$(c_0 ; c_1) \Cap (d_0 ; d_1) \quad \sqsubseteq \quad (c_0 \Cap d_0) ; (c_1 \Cap d_1)$$

Weak conjunction interchange parallel

$$(c_0 \parallel c_1) \Cap (d_0 \parallel d_1) \quad \sqsubseteq \quad (c_0 \Cap d_0) \parallel (c_1 \Cap d_1)$$

Iteration

Iteration zero or more times, c^ω, allows finite iteration, c^*, or infinite iteration, c^∞

$$c^\omega = c^* \sqcap c^\infty \tag{3}$$

Examples

π^* performs a finite number of program steps
$(\pi \sqcap \epsilon)^*$ performs a finite number of steps
ϵ^∞ performs an infinite sequence of environment steps

skip is the identity of parallel and **chaos** is the identity of weak conjunction

$$\begin{aligned} \textbf{skip} &= \epsilon^\omega \\ \textbf{chaos} &= (\pi \sqcap \epsilon)^\omega \end{aligned}$$

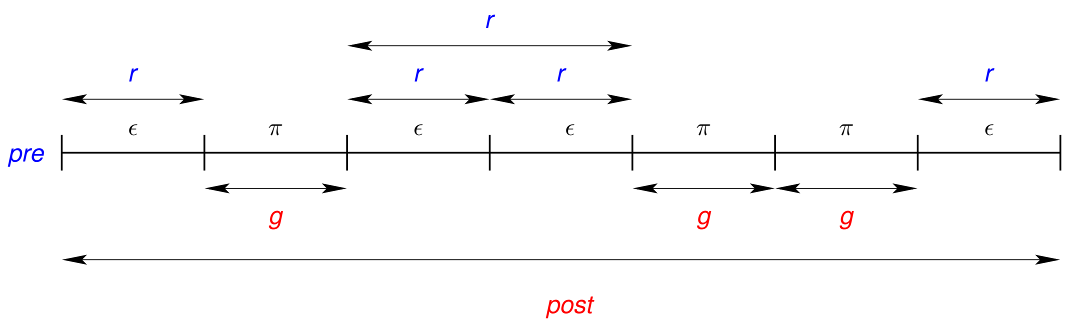
Asynchronised atomic step

$$\langle r \rangle \quad = \quad \epsilon^\omega ; \pi(r) ; \epsilon^\omega$$

For example

$$\begin{aligned} \langle r_1 \rangle \parallel \langle r_2 \rangle &= \epsilon^\omega ; (\pi(r_1) \parallel \pi(r_2)) ; \epsilon^\omega \sqcap \langle r_1 \rangle ; \langle r_2 \rangle \sqcap \langle r_2 \rangle ; \langle r_1 \rangle \\ &= \langle r_1 \rangle ; \langle r_2 \rangle \sqcap \langle r_2 \rangle ; \langle r_1 \rangle \end{aligned}$$

Rely/guarantee



Guarantee and rely

For relations g and r

$$\begin{aligned}\textbf{guar}\ g &= (\pi(g) \sqcap \epsilon)^{\omega} \\ \textbf{rely}\ r &= (\pi \sqcap \epsilon(r) \sqcap \epsilon(\overline{r})\bot)^{\omega} \\ &= (\textbf{assume}\,!\,\epsilon(\overline{r}))^{\omega}\end{aligned}$$

recalling **assume** $a = a \sqcap !\,a ; \bot$ and $!\,\epsilon(\overline{r}) = \pi \sqcap \epsilon(r)$
For example, $c \Cap \textbf{guar}\ g \Cap \textbf{rely}\ r$ imposes a guarantee of g on c and assumes the environment steps satisfy r.

Termination

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

$$\textbf{term} = (\epsilon^{\omega} ; \pi)^{*} ; \epsilon^{\omega} \tag{4}$$

The refinement

$$\textbf{term} \sqsubseteq c$$

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

$$\textbf{term} \sqsubseteq x := 1$$

Specification commands

- Frames on commands
  $x : c = (\textbf{guar}\ \mathrm{id}(\overline{x})) \Cap c$
- Atomic operation
  $\langle q \rangle = \epsilon^{\omega} ; \pi(q) ; \epsilon^{\omega}$
- Non-atomic specification (relational post)
  $\left[q\right] = \bigsqcap_{\sigma \in \Sigma} \tau(\{\sigma\}) ; \textbf{term} ; \tau(\{\sigma' \in \Sigma \mid (\sigma, \sigma') \in q\})$

Lemmas for specifications

$$\begin{aligned}\left[\mathrm{univ}\right] &= \textbf{term} \\ \left[q_1\right] \Cap \left[q_2\right] &= \left[q_1 \wedge q_2\right] \\ \left[q\right] \Cap \textbf{term} &= \left[q\right] \\ \left[q\right] \parallel \textbf{term} &= \left[q\right] \\ q_2 \subseteq q_1 &\Rightarrow \left[q_1\right] \sqsubseteq \left[q_2\right]\end{aligned}$$

Parallel introduction

$$(\textbf{rely}\ r) \Cap \left[q_1 \wedge q_2\right] \sqsubseteq \begin{array}{l}((\textbf{rely}\ r \cup r_1) \Cap \left[q_1\right] \Cap (\textbf{guar}\ r \cup r_2)) \parallel \\ ((\textbf{rely}\ r \cup r_2) \Cap \left[q_2\right] \Cap (\textbf{guar}\ r \cup r_1))\end{array}$$

Proof

$$\begin{array}{ll} & (\textbf{rely}\ r) \Cap \left[q_1 \wedge q_2\right] \\ \sqsubseteq & \text{as } c \Cap c = c \text{ and } \left[q_1 \wedge q_2\right] = \left[q_1\right] \Cap \left[q_2\right] \text{ and weaken relies} \\ & (\textbf{rely}\ r \cup r_1) \Cap \left[q_1\right] \Cap (\textbf{rely}\ r \cup r_2) \Cap \left[q_2\right] \\ \sqsubseteq & \text{by Lemma Y (twice)} \\ & ((\textbf{rely}\ r \cup r_1) \Cap \left[q_1\right]) \parallel ((\textbf{guar}\ r \cup r_1) \Cap \textbf{term}) \Cap \\ & ((\textbf{guar}\ r \cup r_2) \Cap \textbf{term}) \parallel ((\textbf{rely}\ r \cup r_2) \Cap \left[q_2\right]) \\ \sqsubseteq & \text{conjunction-interchange-parallel } (c_1 \parallel c_2) \Cap (d_1 \parallel d_1) \sqsubseteq (c_1 \Cap d_1) \parallel (c_2 \Cap d_2) \\ & ((\textbf{rely}\ r \cup r_1) \Cap \left[q_1\right] \Cap (\textbf{guar}\ r \cup r_2) \Cap \textbf{term}) \parallel \\ & ((\textbf{guar}\ r \cup r_1) \Cap \textbf{term} \Cap (\textbf{rely}\ r \cup r_2) \Cap \left[q_2\right]) \\ \sqsubseteq & \text{by Lemma Q1} \\ & ((\textbf{rely}\ r \cup r_1) \Cap \left[q_1\right] \Cap (\textbf{guar}\ r \cup r_2)) \parallel ((\textbf{guar}\ r \cup r_1) \Cap (\textbf{rely}\ r \cup r_2) \Cap \left[q_2\right])\end{array}$$

Lemma Y

$$(\textbf{rely } r) \Cap [q] \;\sqsubseteq\; ((\textbf{rely } r) \Cap [q]) \parallel ((\textbf{guar } r) \Cap \textbf{term})$$

Proof

$$\begin{array}{ll} & (\textbf{rely } r) \Cap [q] \\ \sqsubseteq & \text{by Lemmas X and Q2} \\ & ((\textbf{rely } r) \parallel (\textbf{guar } r)) \Cap ([q] \parallel \textbf{term}) \\ \sqsubseteq & \text{conjunction-interchange-parallel } (c_1 \parallel c_2) \Cap (d_1 \parallel d_1) \sqsubseteq (c_1 \Cap d_1) \parallel (c_2 \Cap d_2) \\ & ((\textbf{rely } r) \Cap [q]) \parallel ((\textbf{guar } r) \Cap \textbf{term}) \end{array}$$

Lemmas X and Q

Lemma X

$$(\textbf{rely } r) \;\sqsubseteq\; (\textbf{rely } r) \parallel (\textbf{guar } r)$$

Lemma Q1

$$[q] \Cap \textbf{term} \;=\; [q]$$

Lemma Q2

$$[q] \parallel \textbf{term} \;=\; [q]$$

Applying to process algebras

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

- $\Pi(a)$ is interpreted as an atomic event or action a
- $\mathcal{E}(a)$ is a corresponding environment event
- $\pi(A)$ allows any event $\Pi(a)$ 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 $\overline{A}$ independently

- $A : c = c \sqcup \epsilon(\overline{A})^\omega$

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

- $c \|_A d = A : c \parallel A : d$

Iteration

Finite iteration zero or more times, c^*, possibly infinite iteration zero or more times, c^ω, and infinite iteration, c^∞, are defined via their usual recursive equations and have the following unfolding and induction properties.

$$
\begin{array}{rcl}
c^* & \mathrel{\widehat=} & \nu x \cdot \mathbf{nil} \sqcap c\,;x \\
c^* & = & \mathbf{nil} \sqcap c\,;c^* \\
x \sqsubseteq d \sqcap c\,;x & \Rightarrow & x \sqsubseteq c^*\,;d \\
c^* & = & \mathbf{nil} \sqcap c^*\,;c \\
x \sqsubseteq d \sqcap x\,;c & \Rightarrow & x \sqsubseteq d\,;c^* \\
\\
c^\omega & \mathrel{\widehat=} & \mu x \cdot \mathbf{nil} \sqcap c\,;x \\
c^\omega & = & \mathbf{nil} \sqcap c\,;c^\omega \\
d \sqcap c\,;x \sqsubseteq x & \Rightarrow & c^\omega\,;d \sqsubseteq x \\
\\
c^\infty & \mathrel{\widehat=} & \mu x \cdot c\,;x \\
c^\infty & = & c\,;c^\infty \\
c\,;x \sqsubseteq x & \Rightarrow & c^\infty \sqsubseteq x
\end{array}
\tag{5}
$$

Some basic commands

$$\mathit{update}(x, v) \mathrel{\widehat=} \pi(x' = v \wedge \mathrm{id}(\overline{x})) \tag{6}$$

$$\mathbf{skip} \mathrel{\widehat=} \epsilon^\omega \tag{7}$$

$$\mathbf{chaos} \mathrel{\widehat=} (\pi \sqcap \epsilon)^\omega \tag{8}$$

$$\mathbf{term} \mathrel{\widehat=} (\pi \sqcap \epsilon)^*\,;\mathbf{skip} \tag{9}$$

$$\mathbf{idle} \mathrel{\widehat=} (\pi(\mathrm{id}) \sqcap \epsilon)^*\,;\mathbf{skip} \tag{10}$$

$$\langle r \rangle \mathrel{\widehat=} \mathbf{skip}\,;\pi(r)\,;\mathbf{skip} \tag{11}$$

Relies and guarantees

$$\mathbf{guar}\, g \mathrel{\widehat=} (\pi(g) \sqcap \epsilon)^\omega \tag{12}$$

$$\mathbf{rely}\, r \mathrel{\widehat=} (\pi \sqcap \epsilon(r))^\omega\,;(\mathbf{nil} \sqcap \epsilon(\overline{r})\,;\bot) \tag{13}$$

$$x : c \mathrel{\widehat=} (\mathbf{guar}\, \mathrm{id}(\overline{x})) \Cap c \tag{14}$$

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

Tests and steps

$$\tau(p_1)\,;\tau(p_2) = \tau(p_1 \wedge p_2) \tag{15}$$

$$\tau(p)\,;\pi(r) = \pi(p \wedge r) \tag{16}$$

$$\tau(p)\,;\epsilon(r) = \epsilon(p \wedge r) \tag{17}$$

$$\pi(r \wedge p')\,;\tau(p) = \pi(r \wedge p') \tag{18}$$

$$\epsilon(r \wedge p')\,;\tau(p) = \epsilon(r \wedge p') \tag{19}$$

Invariance

If $c\,;\tau(p) \sqsubseteq \tau(p)\,;c$, then

$$\tau(p)\,;c\,;\tau(p) = \tau(p)\,;c\,.$$

Proof.

$$\begin{aligned}&\tau(p)\,;c\,;\tau(p) \sqsubseteq \tau(p)\,;\tau(p)\,;c = \tau(p)\,;c = \\ &\quad \tau(p)\,;c\,;\mathbf{nil} \sqsubseteq \tau(p)\,;c\,;\tau(p)\,.\end{aligned}$$

□

Invariance over steps

If $r \Rrightarrow (p \Rightarrow p')$, then both the following hold.

$$\pi(r)\,;\tau(p) \quad \sqsubseteq \quad \tau(p)\,;\pi(r) \qquad (20)$$

$$\epsilon(r)\,;\tau(p) \quad \sqsubseteq \quad \tau(p)\,;\epsilon(r) \qquad (21)$$

Proof.
The assumption ensures $p \wedge r \wedge p' = p \wedge r$. We give the proof for (**??**) which uses (**??**). The proof for (**??**) is similar but uses (**??**).

$$\begin{aligned}&\pi(r)\,;\tau(p) = \mathbf{nil}\,;\pi(r)\,;\tau(p) \sqsubseteq \tau(p)\,;\pi(r)\,;\tau(p) = \pi(p \wedge r)\,;\tau(p) \\ &= \pi(p \wedge r \wedge p')\,;\tau(p) = \pi(p \wedge r \wedge p') = \pi(p \wedge r) = \tau(p)\,;\pi(r)\end{aligned}$$

□

Invariance over iterations

If $c\,;\tau(p) \sqsubseteq \tau(p)\,;c$, then both

$$c^\omega\,;\tau(p) \sqsubseteq \tau(p)\,;c^\omega \qquad (22)$$

$$c^*\,;\tau(p) \sqsubseteq \tau(p)\,;c^* \qquad (23)$$

Proof.
Property (**??**) holds by ω-induction (**??**) if $\tau(p) \sqcap c\,;\tau(p)\,;c^\omega \;\sqsubseteq\; \tau(p)\,;c^\omega$, which can be proven using the assumption and ω-folding (**??**).

$$\tau(p) \sqcap c\,;\tau(p)\,;c^\omega \;\sqsubseteq\; \tau(p) \sqcap \tau(p)\,;c\,;c^\omega \;=\; \tau(p)\,;(\mathbf{nil} \sqcap c\,;c^\omega) \;=\; \tau(p)\,;c^\omega$$

Property (**??**) holds by *-induction (**??**) if $c^*\,;\tau(p) \;\sqsubseteq\; \tau(p) \sqcap c^*\,;\tau(p)\,;c$, which can be proven using the assumption and *-folding (**??**).

$$\tau(p) \sqcap c^*\,;\tau(p)\,;c \;\sqsupseteq\; \tau(p) \sqcap c^*\,;c\,;\tau(p) \;=\; (\mathbf{nil} \sqcap c^*\,;c)\,;\tau(p) \;=\; c^*\,;\tau(p)$$

□

Rely-invariant

If $r \Rrightarrow (p \Rightarrow p')$, then

$$((\mathbf{rely}\;r) \Cap \mathbf{idle})\,;\tau(p) \;\sqsubseteq\; \tau(p)\,;((\mathbf{rely}\;r) \Cap \mathbf{idle})$$

Proof.
The proof uses the definitions of **rely** r (**??**) and **idle** (**??**) and then pushes the test $\tau(p)$ left using applications of Lemma invariance-iteration. Note that the identity relation id maintains any invariant p.

$$\begin{aligned}&((\mathbf{rely}\;r) \Cap \mathbf{idle})\,;\tau(p) \\ &= ((\pi \sqcap \epsilon(r))^\omega\,;(\mathbf{nil} \sqcap \epsilon(\bar{r})\,;\bot) \Cap (\pi(\mathrm{id}) \sqcap \epsilon)^*\,;\epsilon^\omega)\,;\tau(p) \\ &= (\pi(\mathrm{id}) \sqcap \epsilon(r))^*\,;\epsilon(r)^\omega\,;(\mathbf{nil} \sqcap \epsilon(\bar{r})\,;\bot)\,;\tau(p) \\ &= (\pi(\mathrm{id}) \sqcap \epsilon(r))^*\,;\epsilon(r)^\omega\,;(\tau(p) \sqcap \epsilon(\bar{r})\,;\bot\,;\tau(p)) \\ &\sqsubseteq (\pi(\mathrm{id}) \sqcap \epsilon(r))^*\,;\epsilon(r)^\omega\,;\tau(p)\,;(\mathbf{nil} \sqcap \epsilon(\bar{r})\,;\bot) \\ &\sqsubseteq (\pi(\mathrm{id}) \sqcap \epsilon(r))^*\,;\tau(p)\,;\epsilon(r)^\omega\,;(\mathbf{nil} \sqcap \epsilon(\bar{r})\,;\bot) \\ &\sqsubseteq \tau(p)\,;(\pi(\mathrm{id}) \sqcap \epsilon(r))^*\,;\epsilon(r)^\omega\,;(\mathbf{nil} \sqcap \epsilon(\bar{r})\,;\bot) \\ &= \tau(p)\,;((\mathbf{rely}\;r) \Cap \mathbf{idle})\end{aligned}$$

□

Defining expressions

$$
\begin{aligned}
[\![\kappa]\!]_v &\mathrel{\widehat{=}} \mathbf{idle}\,;\tau(\kappa = v)\,;\mathbf{idle} && (24)\\
[\![x]\!]_v &\mathrel{\widehat{=}} \mathbf{idle}\,;\tau(x = v)\,;\mathbf{idle} && (25)\\
[\![\ominus e]\!]_v &\mathrel{\widehat{=}} \textstyle\bigsqcap\{v_1 \mid v = \mathit{eval}(\ominus, v_1) \cdot [\![e]\!]_{v_1}\} && (26)\\
[\![e_1 \oplus e_2]\!]_v &\mathrel{\widehat{=}} \textstyle\bigsqcap\{v_1, v_2 \mid v = \mathit{eval}(\oplus, v_1, v_2) \cdot [\![e_1]\!]_{v_1} \parallel [\![e_2]\!]_{v_2}\} && (27)
\end{aligned}
$$

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 $(x \bmod N)$ 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,

$$r \Rrightarrow (e = v \Rightarrow e' = v)\,.$$

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 κ is trivially stable.
- A variable x is stable under r if for fresh v, $r \Rrightarrow (x = v \Rightarrow x' = v)$.
- 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,

$$(\mathbf{rely}\ r) \Cap (\mathbf{idle}\,;\tau(e = v)) \ \sqsubseteq\ (\mathbf{rely}\ r) \Cap (\tau(e = v)\,;\mathbf{idle})$$

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 [**?**, **?**]. If x is subject to modification then $x + x$ is not single-reference but $2 * x$ 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.

Single-reference expressions

Definition (single-reference-expression)

The definition is based on the syntactic form of e.

- A constant κ 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 σ in which the single-reference variable (x) is accessed.

Defining commands

$$x := e \;\widehat{=}\; \textstyle\bigsqcap_{v \in Val} [\![e]\!]_v \,;\, update(x, v)\} \,;\, \textbf{idle} \quad (28)$$

$$\textbf{if } b \textbf{ then } c \textbf{ else } d \;\widehat{=}\; (([\![b]\!]_{\text{true}} \,;\, c) \sqcap ([\![\neg\, b]\!]_{\text{true}} \,;\, d)) \,;\, \textbf{idle} \quad (29)$$

$$\textbf{while } b \textbf{ do } c \;\widehat{=}\; ([\![b]\!]_{\text{true}} \,;\, c)^{\omega} \,;\, [\![\neg\, b]\!]_{\text{true}} \quad (30)$$

$$[q] \;\widehat{=}\; \textstyle\bigsqcap_{\sigma \in \Sigma} \tau(\{\sigma\}) \,;\, \textbf{term} \,;\, \tau(\{\sigma' \mid (\sigma, \sigma') \in q\}) \quad (31)$$

$$[p\,,\ q] \;\widehat{=}\; \{p\} \,;\, [q] \quad (32)$$

Rely sequential

A specification with a post condition which is the composition of two relations q_1 and q_2 may be refined by by a sequential composition of one command satisfying q_1 and a second satisfying q_2.
For rely condition r, predicates p_0, p_1 and p_2, and relations q_1 and q_2.

$$(\textbf{rely } r) \Cap [p_0\,,\ (q_1 \mathbin{⨟} q_2) \wedge p_2'] \sqsubseteq ((\textbf{rely } r) \Cap [p_0\,,\ q_1 \wedge p_1']) \,;\, ((\textbf{rely } r) \Cap [p_1\,,\ q_2 \wedge p_2'])$$

Single-reference test

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,

$$(\mathbf{rely}\ r) \Cap (\mathbf{idle}\,;\boldsymbol{\tau}(e = \kappa)\,;\mathbf{idle}) \quad \sqsubseteq \quad [[e]]_\kappa\,.$$

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

Rely idle

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

$$(\mathbf{rely}\ r) \Cap \left[p\,,\ r^* \wedge p'\right] \sqsubseteq (\mathbf{rely}\ r) \Cap \mathbf{idle}\,.$$

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^*. □

Rely test

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

$$(\mathbf{rely}\ r) \Cap \left[p\,,\ r^* \wedge p' \wedge b_0'\right] \ \sqsubseteq\ [[b]]_{\mathrm{true}}\,.$$

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

$$\begin{aligned}
&(\mathbf{rely}\ r) \Cap \left[p\,,\ r^* \wedge p' \wedge b_0'\right] \\
\sqsubseteq\ &((\mathbf{rely}\ r) \Cap \left[p\,,\ r^* \wedge p'\right])\,;((\mathbf{rely}\ r) \Cap \left[p\,,\ \mathrm{id} \wedge p' \wedge b\right]); \\
&\qquad ((\mathbf{rely}\ r) \Cap \left[p \wedge b_0\,,\ r^* \wedge p' \wedge b_0'\right]) \\
\sqsubseteq\ &\mathbf{idle}\,;\boldsymbol{\tau}(b)\,;\mathbf{idle} \\
\sqsubseteq\ &[[b]]_{\mathrm{true}}
\end{aligned}$$

□

Rely assignment

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

$$(\mathbf{rely}\ r) \Cap x\colon\left[e \approx x' \approx e'\right] \ \sqsubseteq\ x := e$$

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 (**??**).

$$
\begin{array}{ll}
 & (\textbf{rely}\ r) \Cap x \colon \left[e \approx x' \approx e'\right] \\
\sqsubseteq & (\textbf{rely}\ r) \Cap x \colon \left[\exists\, v \cdot e \approx v \approx e' \wedge x' = v\right] \\
\sqsubseteq & (\textbf{rely}\ r) \Cap \bigsqcap_{v \in Val} x \colon \left[e \approx v \approx e' \wedge x' = v\right] \\
\sqsubseteq & (\textbf{rely}\ r) \Cap \bigsqcap_{v \in Val} \left[e \approx v \approx e'\right] ; x \colon \left[e \approx e' \wedge x' = v\right] \\
\sqsubseteq & (\textbf{rely}\ r) \Cap \bigsqcap_{v \in Val} \left[e \approx e'\right] ; \left[v = e = e'\right] ; \left[e \approx e'\right] ; x \colon \left[e = e' \wedge x' = v\right] ; \left[e \approx e'\right] \\
\sqsubseteq & (\textbf{rely}\ r) \Cap \bigsqcap_{v \in Val} \textbf{idle} ; \tau(v = e) ; \textbf{idle} ; update(x, v) ; \textbf{idle} \\
\sqsubseteq & (\textbf{rely}\ r) \Cap \bigsqcap_{v \in Val} [\![e]\!]_v ; update(x, v) ; \textbf{idle} \\
\sqsubseteq & x := e
\end{array}
$$

□

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. $b \Rrightarrow b_0$ and $r \Rrightarrow (b_0 \Rightarrow b_0')$. 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.

$$p \wedge b \Rrightarrow b_0 \qquad p \wedge r \Rrightarrow (b_0 \Rightarrow b_0')$$

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

$$p \wedge \neg\, b \Rrightarrow b_1 \qquad p \wedge r \Rrightarrow (b_1 \Rightarrow b_1')$$

For example, the negation of the earlier example is $oc \geq ot \vee oc \geq et$ and that is maintained by interference that may only decrease et. Note that

$$p \Rrightarrow (p \wedge b) \vee (p \wedge \neg\, b) \Rrightarrow b_0 \vee b_1$$

but there may be states in which both b_0 and b_1 hold. For the above example, taking b_0 as $oc < ot$ and b_1 as $oc \geq ot \vee oc \geq et$, both conditions hold in states satisfying $oc < ot \wedge oc \geq et$.

Loops

The Hoare logic rule for reasoning about a loop, **while** b **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 $(\_ \succ \_)$ whenever b holds initially.

The invariant and the variant

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.

$$r \Rrightarrow (p \Rightarrow p') \tag{33}$$
$$p \wedge r \Rrightarrow v \succeq v' \tag{34}$$

Rely finite iteration

For predicate p, and relation q, if r maintains p,

$$(\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge q^*\right] \ \sqsubseteq\ ((\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge q\right])^*$$

Proof.
The proof is via finite iteration induction (**??**) and the refinement holds if,

$$(\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge q^*\right] \sqsubseteq \textbf{nil} \sqcap ((\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge q\right]) \,;((\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge q^*\right])$$

which holds by Law rely-sequential because $q \mathbin{⨟} q^* \Rrightarrow q^*$. □

Rely well-founded iteration

For predicate p, variant expression v of type T, and a relation $(\_ \succ \_) \in T \times T$ that is well-founded on p, if r maintains p, and v is non-increasing under r,

$$(\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge v \succeq v'\right] \ \sqsubseteq\ ((\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge v \succ v'\right])^\omega$$

Proof.

$$\begin{array}{ll}
& ((\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge v \succ v'\right])^\omega \\
= & \text{isolation, i.e. } c^\omega = c^* \sqcap c^\infty \\
& ((\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge v \succ v'\right])^* \sqcap ((\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge v \succ v'\right])^\infty \\
= & \text{well-founded infinite iteration is infeasible} \\
& ((\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge v \succ v'\right])^* \\
\sqsupseteq & \text{by Law rely-finite-iteration} \\
& (\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge v \succeq v'\right]
\end{array}$$

□

Rely loop

Given predicates p, b_0 and b_1, 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

$$\begin{array}{lll}
p \wedge r \Rrightarrow p' & p \wedge b \Rrightarrow b_0 & p \wedge r \wedge b_0 \Rrightarrow b_0' \\
p \wedge r \Rrightarrow v \succeq v' & p \wedge \neg\, b \Rrightarrow b_1 & p \wedge r \wedge b_1 \Rrightarrow b_1'
\end{array}$$

then

$$(\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge b_1' \wedge v \succeq v'\right] \ \sqsubseteq\ \textbf{while}\ b\ \textbf{do}((\textbf{rely}\ r) \Cap \left[p \wedge b_0\,,\ p' \wedge v \succ v'\right])$$

Proof.

$$\begin{array}{ll}
& (\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge b_1' \wedge v \succeq v'\right] \\
\sqsubseteq & \text{by Law rely-sequential} \\
& ((\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge v \succeq v'\right]) \,;((\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge b_1' \wedge v \succeq v'\right]) \\
\sqsubseteq & \text{by Law rely-test using the assumptions on } b_1 \\
& ((\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge v \succeq v'\right]) \,; [[\neg\, b]]_{\mathrm{true}} \\
\sqsubseteq & \text{by Law rely-well-founded-iteration} \\
& ((\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge v \succ v'\right])^\omega \,; [[\neg\, b]]_{\mathrm{true}} \\
\sqsubseteq & \text{by Law rely-sequential as } (v \succeq v') \mathbin{⨟} (v \succ v') \Rrightarrow v \succ v' \\
& (((\textbf{rely}\ r) \Cap \left[p\,,\ p' \wedge b_0' \wedge v \succeq v'\right]) \,;((\textbf{rely}\ r) \Cap \left[p \wedge b_0\,,\ p' \wedge v \succ v'\right]))^\omega \,; [[\neg\, b]]_{\mathrm{true}} \\
\sqsubseteq & \text{by Law rely-test using the assumptions on } b_0 \\
& ([[b]] \,;((\textbf{rely}\ r) \Cap \left[p \wedge b_0\,,\ p' \wedge v \succ v'\right]))^\omega \,; [[\neg\, b]]_{\mathrm{true}} \\
= & \text{definition of loop } (\textbf{??}) \\
& \textbf{while}\ b\ \textbf{do}((\textbf{rely}\ r) \Cap \left[p \wedge b_0\,,\ p' \wedge v \succ v'\right])
\end{array}$$

□

Rely loop early

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

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl} p \wedge r & \Rrightarrow & p' & p \wedge b & \Rrightarrow & b_0 & p \wedge r \wedge b_0 & \Rrightarrow & b_0' \\ p \wedge r & \Rrightarrow & v \succeq v' & p \wedge \neg b & \Rrightarrow & b_1 & p \wedge r \wedge b_1 & \Rrightarrow & b_1' \\ & & & p \wedge b_2 & \Rrightarrow & \neg b & p \wedge r \wedge b_2 & \Rrightarrow & b_2' \end{array}$$

then

$$(\mathbf{rely}\, r) \Cap [p\, ,\; p' \wedge b_1'] \sqsubseteq \mathbf{while}\, b\, \mathbf{do}((\mathbf{rely}\, r) \Cap [p \wedge b_0\, ,\; p' \wedge (v \succ v' \vee b_2')])$$

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*.

Some things I haven't covered

- Local variables
- Modules

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 $\|$, $\Cap$, ;
- 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