A denotational semantic theory of concurrent systems

Jayadev Misra[1]

Department of Computer Science
University of Texas at Austin

http://orc.csres.utexas.edu

WG 2.3, Orlando
May 2013

[1]Thanks to Ernie Cohen.

Denotational Semantics of Concurrent Systems

- Scott's denotational semantics specialized to concurrent systems.
- Strong results for this specific domain.
- Inappropriate for other areas, such as sequential programs.
- Derive specification of a program from those of its components.
- Prove program properties (safety, progress) from the specification.

Denotational Semantics

- $f \oplus g$ is a program constructed out of

  components f and g, and

  combinator $\oplus$, a programming language construct.

- The specification of $f \oplus g$, $[\![f \oplus g]\!]$ is given by:

  $$[\![f \oplus g]\!] \triangleq [\![f]\!] \; [\![\oplus]\!] \; [\![g]\!]$$

- $[\![\oplus]\!]$ is a transformer of specifications:

  It combines two specifications, $[\![f]\!]$ and $[\![g]\!]$, to yield a specification.

Notation Overloading: use $\oplus$ instead of $[\![\oplus]\!]$.

Denotational Semantics

- $f \oplus g$ is a program constructed out of

  components f and g, and

  combinator $\oplus$, a programming language construct.

- The specification of $f \oplus g$, $[\![f \oplus g]\!]$ is given by:

  $$[\![f \oplus g]\!] \ \underline{\Delta} \ \ [\![f]\!] \ [\![\oplus]\!] \ [\![g]\!]$$

- $[\![\oplus]\!]$ is a transformer of specifications:

  It combines two specifications, $[\![f]\!]$ and $[\![g]\!]$, to yield a specification.

Notation Overloading: use $\oplus$ instead of $[\![\oplus]\!]$.

Denotational Semantics

- $f \oplus g$ is a program constructed out of

  components f and g, and

  combinator $\oplus$, a programming language construct.

- The specification of $f \oplus g$, $[\![f \oplus g]\!]$ is given by:

  $$[\![f \oplus g]\!] \ \underline{\Delta} \ \ [\![f]\!] \ [\![\oplus]\!] \ [\![g]\!]$$

- $[\![\oplus]\!]$ is a transformer of specifications:

  It combines two specifications, $[\![f]\!]$ and $[\![g]\!]$, to yield a specification.

Notation Overloading: use $\oplus$ instead of $[\![\oplus]\!]$.

Denotational Semantics

- $f \oplus g$ is a program constructed out of

  components f and g, and

  combinator $\oplus$, a programming language construct.

- The specification of $f \oplus g$, $[\![f \oplus g]\!]$ is given by:

  $$[\![f \oplus g]\!] \ \underline{\Delta} \quad [\![f]\!] \ [\![\oplus]\!] \ [\![g]\!]$$

- $[\![\oplus]\!]$ is a transformer of specifications:

  It combines two specifications, $[\![f]\!]$ and $[\![g]\!]$, to yield a specification.

Notation Overloading: use $\oplus$ instead of $[\![\oplus]\!]$.

Contributions of this work

- specifications of concurrent components.
- A theory of transformers: functions mapping specs to specs.
- Proving safety, progress, branching time properties with:

  concurrency

  non-determinacy

  recursion

  shared resource

  fairness

  divergence

  real-time

Status

- Completed the theoretical work
- Currently being applied to Orc calculus
- Need to do considerable work in verification engineering
  - logic for specification
  - applications to other process algebras
  - mechanization

Summary

Closure	Meaning	Preserving Transformer	Corresponding Function
Downward	Prefix-closed	Smooth	Monotonic
Upward	Limit-closed	Bismooth	Continuous

- A library of smooth and bismooth transformers.
- Fixed-point theorems:
  - Least upward-closed fixed point
  - Min-max fixed point (to deal with fairness)

Component Specification

- Events.
- Traces.
- A specification is a prefix-closed set of traces.

Events associated with a component

$pub(true)$	publish (output) a value
$x.read(3)$	read a value from variable x
$c.receive(\text{"}val\text{"})$	receive "val" from channel c
$Heads/Tails$	outcome of an internal coin toss
$x.add(5)$	Method call

- Events are event instances.
- They are uninterpreted, instantaneous and atomic.
- There is a universal event alphabet.

Events associated with a component

$pub(true)$	publish (output) a value
$x.read(3)$	read a value from variable x
$c.receive(\text{"}val\text{"})$	receive "val" from channel c
$Heads/Tails$	outcome of an internal coin toss
$x.add(5)$	Method call

- Events are event instances.
- They are uninterpreted, instantaneous and atomic.
- There is a universal event alphabet.

Execution of a component (informal notion)

An execution is a sequence of events.

Toss a coin and publish the outcome.
Two possible executions:

$[Heads, pub(\text{"}Heads\text{"})]$
$[Tails, pub(\text{"}Tails\text{"})]$

With all intermediate executions:

$[\,]$
$[Heads]$
$[Heads, pub(\text{"}Heads\text{"})]$
$[Tails]$
$[Tails, pub(\text{"}Tails\text{"})]$

Execution of a component (informal notion)

An execution is a sequence of events.

Toss a coin and publish the outcome.
Two possible executions:

$$[Heads, pub(\text{"}Heads\text{"})]$$
$$[Tails, pub(\text{"}Tails\text{"})]$$

With all intermediate executions:

$$[\,]$$
$$[Heads]$$
$$[Heads, pub(\text{"}Heads\text{"})]$$
$$[Tails]$$
$$[Tails, pub(\text{"}Tails\text{"})]$$

Another Program

Two tosses, but stop if the first toss is Heads

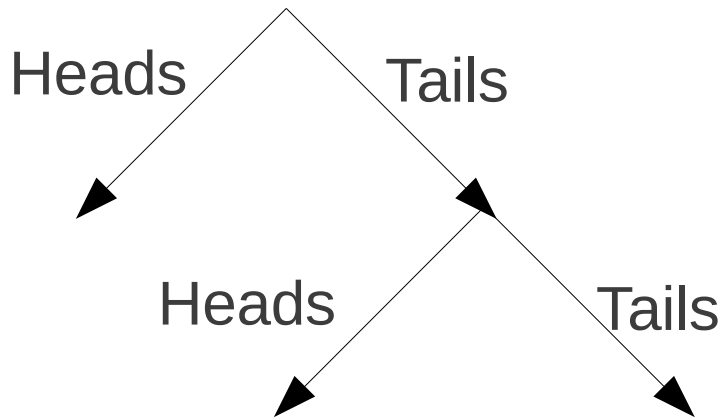
$[Heads]$
$[Tails, Heads]$
$[Tails, Tails]$

Plus all the prefixes of these sequences.

Depict Executions by a tree

Two tosses, but stop if the first toss is Heads

$[Heads]$, $[Tails, Heads]$, $[Tails, Tails]$ plus the prefixes.



- Each node is an execution.
- Label on each branch is an event.
- An ancestor is a prefix.

Infinite Executions

Toss a coin repeatedly until it lands Heads.

$[\,]$
$[\mathit{Heads}]$ $[\mathit{Tails}]$
$[\mathit{Tails}, \mathit{Heads}]$ $[\mathit{Tails}, \mathit{Tails}]$
$[\mathit{Tails}, \mathit{Tails}, \mathit{Heads}]$ $[\mathit{Tails}, \mathit{Tails}, \mathit{Tails}]$
$[\mathit{Tails}, \mathit{Tails}, \mathit{Tails}, \mathit{Heads}] \cdots$

- An unfair coin may may always land Tails.
- Admit infinite execution: $[\mathit{Tails}, \mathit{Tails}, \mathit{Tails}, \cdots]$
- Executions described by:
  $\{[\mathit{Tails}^j] \mid j \geq 0\} \cup \{[\mathit{Tails}^j, \mathit{Heads}] \mid j \geq 0\} \cup \{[\mathit{Tails}^\omega]\}$

Infinite Executions

Toss a coin repeatedly until it lands Heads.

$$
\begin{array}{ll}
[\,] & \\
[\mathit{Heads}] & [\mathit{Tails}] \\
[\mathit{Tails}, \mathit{Heads}] & [\mathit{Tails}, \mathit{Tails}] \\
[\mathit{Tails}, \mathit{Tails}, \mathit{Heads}] & [\mathit{Tails}, \mathit{Tails}, \mathit{Tails}] \\
[\mathit{Tails}, \mathit{Tails}, \mathit{Tails}, \mathit{Heads}] \cdots &
\end{array}
$$

- An unfair coin may may always land Tails.
- Admit infinite execution: $[\mathit{Tails}, \mathit{Tails}, \mathit{Tails}, \cdots]$
- Executions described by:
  $\{[\mathit{Tails}^j] \mid j \geq 0\} \cup \{[\mathit{Tails}^j, \mathit{Heads}] \mid j \geq 0\} \cup \{[\mathit{Tails}^\omega]\}$

Status of an Execution

- Status denotes the final state of an execution. From $\{W, H, D\}$.
- Infinite execution has status D.
- Finite executions typically have status H or W. Some have D.

  W is Waiting:
  more autonomous computation to do or waiting for external input.

  H is Halted: nothing more to do.

  D is Divergent: An infinite computation.

- Example of Divergent Execution

  def loop( ) = loop( )

Trace

A trace is $s[m]$ where

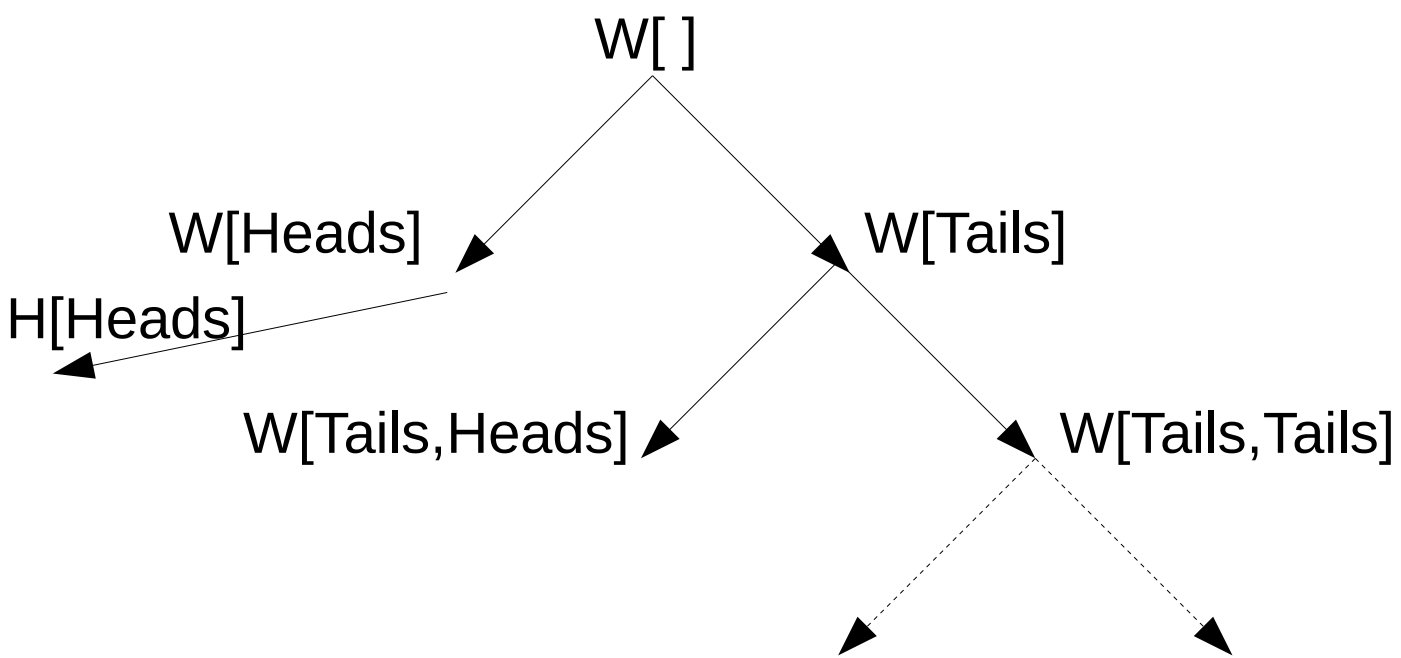
- s, status, is from $\{W, H, D\}$.
- m finite or infinite event sequence.

Trace (formal notion)

Trace: A sequence of events plus the final state of computation.

Toss a coin and publish the outcome. Possible traces are:

$W[\,]$ $\quad W[Heads]$ $\quad W[Tails]$
$W[Heads, pub(\text{"}Heads\text{"})]$ $\quad W[Tails, pub(\text{"}Tails\text{"})]$
$H[Heads, pub(\text{"}Heads\text{"})]$ $\quad H[Tails, pub(\text{"}Tails\text{"})]$



Trace prefix

In the trace tree, prefix of a node is an ancestor.

Formally, $s[m] \leq s'[m']$, means

$s[m] = s'[m']$, or

$(s = W)$ and (m prefix of m')

Applies to infinite traces.

- $\leq$ is a partial order.
- $<$ is a well-founded order.
- $W[\,]$ is the bottom trace.

Trace prefix

In the trace tree, prefix of a node is an ancestor.

Formally, $s[m] \leq s'[m']$, means

$s[m] = s'[m']$, or

$(s = W)$ and (m prefix of m')

Applies to infinite traces.

- $\leq$ is a partial order.
- $<$ is a well-founded order.
- $W[\,]$ is the bottom trace.

Prefix Closure (downward closure)

Prefix closure of trace t is the set of all its prefixes:

$$t_* = \{s \mid s \leq t\}$$

For traceset (non-empty set of traces) p define downward closure by:

$p_* = \cup_{t \in p}(t_*)$, for non-empty p

$(p \times q \times \cdots \times r)_* = p_* \times q_* \cdots \times r_*$ Cartesian Product

Spec

- A specification (spec) is a prefix-closed set of traces.
- Definition: Traceset p is a spec iff $p = p_*$.
- Note: A spec is always non-empty.

Meaning of a component spec

- Each trace in a spec is a possible execution in some environment.
- So, a spec is prefix-closed.
- $W[m]$ without extension denotes deadlock.
- Eventual halting:
  - Every waiting trace has an extension by an autonomous event.
  - There is no divergent trace.

Example spec: successor

- $suc(x)$ publishes the successor of the argument integer x.
- spec: $\{H[read(i), pub(i+1)] \mid i \text{ integer}\}_*$

Example spec: conditional

- $Ift(b)$ for boolean b
  - publishes signal for $b = true$
  - halts silently for $b = false$
- spec: $\{H[read(false)], H[read(true), pub(signal)]\}_*$

Tree depiction of a spec is insufficient

Toss a coin sequentially until it lands Heads.

unfair coin: $\{H[Tails^j, Heads] \mid j \geq 0\}_* \cup \{D[Tails^\omega]\}$

fair coin: $\{H[Tails^j, Heads] \mid j \geq 0\}_*$

Explicit inclusion/exclusion of infinite traces in a spec.

Transformers

Denotational Semantics (repeated)

- $f \oplus g$ is a program constructed out of

  components f and g, and

  combinator $\oplus$, a programming language construct.

- The specification of $f \oplus g$, $[\![f \oplus g]\!]$ is given by:

  $$[\![f \oplus g]\!] \triangleq [\![f]\!][\![\oplus]\!][\![g]\!]$$

- $[\![\oplus]\!]$ is a transformer:

  It combines two specifications, $[\![f]\!]$ and $[\![g]\!]$, to yield a specification.

  Notation Overloading: use $\oplus$ instead of $[\![\oplus]\!]$.

A Motivating Example

- Programming language construct, $\oplus$: $\oplus(A, B, C)$
- Execute A, B concurrently.
- If A engages in e and B in $\overline{e}$, they rendezvous. Then start C to run concurrently with A and B.

A Motivating Example: $\oplus(A, B, C)$

- Let specifications of A, B, C be p, q, r, respectively.
- C' starts with event a and then behaves as C:
  spec is $cons(a, r)$.
- spec of A, B, C' running concurrently: $p \mid q \mid cons(a, r)$.
- Retain those traces in which $\{e, \overline{e}, a\}$ are contiguous.
  Replace these 3 events by event τ:
  $rendezvous(\{e, \overline{e}, a\}, \tau, (p \mid q \mid cons(a, r)))$
- Drop the τ symbol from each trace:
  $\oplus'(p, q, r) =$
  $drop(\tau, rendezvous(\{e, \overline{e}, a\}, \tau, (p \mid q \mid cons(a, r))))$

A Motivating Example: $\oplus(A, B, C)$

- Let specifications of A, B, C be p, q, r, respectively.
- C' starts with event a and then behaves as C:
  spec is $cons(a, r)$.
- spec of A, B, C' running concurrently: $p \mid q \mid cons(a, r)$.
- Retain those traces in which $\{e, \overline{e}, a\}$ are contiguous.
  Replace these 3 events by event τ:
  $rendezvous(\{e, \overline{e}, a\}, \tau, (p \mid q \mid cons(a, r)))$
- Drop the τ symbol from each trace:
  $\oplus'(p, q, r) =$
  $drop(\tau, rendezvous(\{e, \overline{e}, a\}, \tau, (p \mid q \mid cons(a, r))))$

A Motivating Example: $\oplus(A, B, C)$

- Let specifications of A, B, C be p, q, r, respectively.
- C' starts with event a and then behaves as C:
  spec is $cons(a, r)$.
- spec of A, B, C' running concurrently: $p \mid q \mid cons(a, r)$.
- Retain those traces in which $\{e, \overline{e}, a\}$ are contiguous.
  Replace these 3 events by event τ:
  $rendezvous(\{e, \overline{e}, a\}, \tau, (p \mid q \mid cons(a, r)))$
- Drop the τ symbol from each trace:
  $\oplus'(p, q, r) =$
  $drop(\tau, rendezvous(\{e, \overline{e}, a\}, \tau, (p \mid q \mid cons(a, r))))$

A Motivating Example: $\oplus\,(A, B, C)$

- Let specifications of A, B, C be p, q, r, respectively.
- C' starts with event a and then behaves as C:
  spec is $cons(a, r)$.
- spec of A, B, C' running concurrently: $p \mid q \mid cons(a, r)$.
- Retain those traces in which $\{e, \overline{e}, a\}$ are contiguous.
  Replace these 3 events by event τ:
  $rendezvous(\{e, \overline{e}, a\}, \tau, (p \mid q \mid cons(a, r)))$
- Drop the τ symbol from each trace:
  $\oplus'\,(p, q, r) =$
  $drop(\tau, rendezvous(\{e, \overline{e}, a\}, \tau, (p \mid q \mid cons(a, r))))$

A Motivating Example: $\oplus\,(A, B, C)$

- Let specifications of A, B, C be p, q, r, respectively.
- C' starts with event a and then behaves as C:
  spec is $cons(a, r)$.
- spec of A, B, C' running concurrently: $p \mid q \mid cons(a, r)$.
- Retain those traces in which $\{e, \overline{e}, a\}$ are contiguous.
  Replace these 3 events by event τ:
  $rendezvous(\{e, \overline{e}, a\}, \tau, (p \mid q \mid cons(a, r)))$
- Drop the τ symbol from each trace:
  $\oplus'\,(p, q, r) =$
  $drop(\tau, rendezvous(\{e, \overline{e}, a\}, \tau, (p \mid q \mid cons(a, r))))$

Example Transformer: Sequential Composition, $f;g$

- g starts executing when and only when f halts.
- A trace of $f;g$ is of the form:
  - $s[m]$ where $s[m]$ is a trace of f and s is W or D, or
  - $s[m\ n]$ where

    $H[m]$ is a trace of f

    $s[n]$ is a trace of g

Example Transformer: Sequential Composition, $f;g$

- g starts executing when and only when f halts.
- A trace of $f;g$ is of the form:
  - $s[m]$ where $s[m]$ is a trace of f and s is W or D, or
  - $s[m\ n]$ where

    $H[m]$ is a trace of f

    $s[n]$ is a trace of g

Example Transformer: parallel composition, $f \mid g$

- f and g execute independently.
- Let $s[m]$ be a trace of f, $t[n]$ of g, s and t from $\{H, W\}$.

  Then, $f \mid g$ includes traces $(s \cap t)(m \otimes n)$ where:
  - $\cap$ symmetric. $H \cap s = s$, $W \cap W = W$.
  - $m \otimes n$ is all interleavings (merge) of m and n.
- Merging with infinite sequence: fair and unfair merge.

Example Transformer: parallel composition, $f \mid g$

- f and g execute independently.
- Let $s[m]$ be a trace of f, $t[n]$ of g, s and t from $\{H, W\}$.

  Then, $f \mid g$ includes traces $(s \cap t)(m \otimes n)$ where:
  - $\cap$ symmetric. $H \cap s = s$, $W \cap W = W$.
  - $m \otimes n$ is all interleavings (merge) of m and n.
- Merging with infinite sequence: fair and unfair merge.

Example Transformer: parallel composition, $f \mid g$

- f and g execute independently.
- Let $s[m]$ be a trace of f, $t[n]$ of g, s and t from $\{H, W\}$.

  Then, $f \mid g$ includes traces $(s \cap t)(m \otimes n)$ where:
  - $\cap$ symmetric. $H \cap s = s$, $W \cap W = W$.
  - $m \otimes n$ is all interleavings (merge) of m and n.
- Merging with infinite sequence: fair and unfair merge.

Example Transformer: parallel composition, $f \mid g$

- f and g execute independently.
- Let $s[m]$ be a trace of f, $t[n]$ of g, s and t from $\{H, W\}$.

  Then, $f \mid g$ includes traces $(s \cap t)(m \otimes n)$ where:
  - $\cap$ symmetric. $H \cap s = s$, $W \cap W = W$.
  - $m \otimes n$ is all interleavings (merge) of m and n.
- Merging with infinite sequence: fair and unfair merge.

Definition: Transformer, Trace-wise Transformer

- A transformer is a function that maps a tuple of specs to a spec: $f(p, q, \cdots, r)$

  Notation: Infix $p \oplus q$ for 2-tuple transformer .

- Tracewise-transformer: Maps a tuple of traces to a traceset. Then,

$$f(p) = \cup\{f(t) \mid t \in p\}$$

$$p \oplus q = \cup\{s \oplus t \mid s \in p,\ t \in q\}$$

- Henceforth all transformers are tracewise.

When is $f(p)$ a spec given that p is a spec?

Smooth Transformer

- A smooth transformer preserves prefix closure.
- Smooth Transformer: For any trace s,

$$f_*(s) = f(s_*) \qquad \text{(Notation: } f_*(s) \text{ is } (f(s))_*\text{)}$$
$$(s \oplus t)_* = s_* \oplus t_*$$

Properties of smooth transformers

- For smooth f and spec p, $f_*(p) = f(p_*)$.
- Follows: A smooth transformer transforms specs to specs.
- Composition of smooth transformers is smooth.
- f is smooth iff
  - f transforms specs to specs, and
  - f is monotonic: $s \leq t \Rightarrow f_*(s) \subseteq f_*(t)$.

Example of Smooth Transformer: choice

- f *or* g: choose to execute either f or g

  transformer: s *or* $t = \{s\} \cup \{t\}$

- *or* is smooth.

Example of Smooth Transformer: cons

- Append a specific event a as the first event of every trace.
- $$cons(a, W[\,]) = \{W[\,], W[a]\}$$
  $$cons(a, s[m]) = \{s[a\ m]\}$$

Example of Smooth Transformer: Filter

- A filter transformer accepts or rejects each trace.
- A *filter* is defined by a predicate b on traces, where
  1. $b(W[\,])$ holds, and
  2. If $b(t)$ holds then $b(s)$ holds for all prefixes s of t.
- A filter transformer accepts all prefixes for which b holds.

$$f(t) = \{s \mid b(s) \wedge s \leq t\}$$

Examples of Smooth transformers

- unfair merge: $f \mid g$
- fair merge: $f \mid' g$
- rendezvous: merge traces so that events e and e' are contiguous.
- sequential composition: $f \,;\, g$

$$H[m] \,;\, t[n] = \{t[m\ n]\},$$

$$s \,;\, t[n] = \{s\}, \text{ otherwise}$$

Fairness

- Coin tosses are fair.
- Fair scheduler: In a multiprocess implementation every process gets to execute eventually.
- A semaphore is granted fairly.
- Any finite interval in time can contain only a finite number of events.

Fairness is a filter transformer

- The transformer accepts all finite traces,
  accepts the fair infinite traces and rejects the unfair ones.

- Fits the definition of a filter, a smooth transformer.

Example: coin toss forever until Heads appears.

- unfair coin:

$$\{H[Tails^j, Heads] \mid j \geq 0\}_* \cup \{D[Tails^\omega]\}$$

- fair coin: Apply the filter that rejects the infinite sequence of Tails.

$$\{H[Tails^j, Heads] \mid j \geq 0\}_*$$

Fairness is a filter transformer

- The transformer accepts all finite traces,
  accepts the fair infinite traces and rejects the unfair ones.

- Fits the definition of a filter, a smooth transformer.

Example: coin toss forever until Heads appears.

- unfair coin:

$$\{H[Tails^j, Heads] \mid j \geq 0\}_* \cup \{D[Tails^\omega]\}$$

- fair coin: Apply the filter that rejects the infinite sequence of Tails.

$$\{H[Tails^j, Heads] \mid j \geq 0\}_*$$

Shared Resource

- Consider $x.read() \mid x.write(3)$,
  where local variable x is initialized to 0.

- spec of $x.read()$ includes the trace $H[read(5)]$.

  spec of program $x.write(3)$ is $H[write(3)]_*$

- Applying merge: a trace of $x.read() \mid x.write(3)$ is

  $H[read(5), write(3)]$, an invalid trace.

Parallel executions may not be independent

- The complete program is

  int $x = 0$

  $x.read() \mid x.write(3)$

- The declaration "int $x = 0$" induces a filter transformer, $x.int$.

  It rejects all traces that are not possible with the resource.

- Given specs p and q of $x.read()$ and $x.write(3)$, spec of

  int $x = 0$

  $x.read() \mid x.write(3)$

  is $x.int(p \mid q)$

Research Area

- Each shared resource is defined by a filter.
- Each filter is an acceptor of strings, i.e., a formal language.
- So, a shared resource can be specified as a language.
- The language may include infinite strings, say, for strong semaphore.
- I have defined filters for
  read/write shared variables,
  write-once variables,
  channel,
  weak and strong semaphore

Recursion: Procedure $stut()$

- Toss an unfair coin
  if it lands Heads halt, otherwise call $stut()$.

- Let the spec of $stut()$ be x.

- $stut()$ chooses between
  - halting the computation (when toss lands Heads), with spec $H[Heads]_*$, and
  - event Tails followed by $stut()$, with spec $cons(Tails, x)$
  - The transformer for choice is set union.

- $x = H[Heads]_* \cup cons(Tails, x)$

Recursion: Procedure $stut()$

- Toss an unfair coin
  if it lands Heads halt, otherwise call $stut()$.

- Let the spec of $stut()$ be x.

- $stut()$ chooses between
  - halting the computation (when toss lands Heads), with spec $H[Heads]_*$, and
  - event Tails followed by $stut()$, with spec $cons(Tails, x)$
  - The transformer for choice is set union.

- $x = H[Heads]_* \cup cons(Tails, x)$

Recursion: Procedure $stut()$

- Toss an unfair coin
  if it lands Heads halt, otherwise call $stut()$.

- Let the spec of $stut()$ be x.

- $stut()$ chooses between
  - halting the computation (when toss lands Heads), with spec $H[Heads]_*$, and
  - event Tails followed by $stut()$, with spec $cons(Tails, x)$
  - The transformer for choice is set union.

- $x = H[Heads]_* \cup cons(Tails, x)$

Solutions of recursive equation: $x = F(x)$

- Extensively studied in denotational semantics where x, called a point, is from a complete partial order (CPO).
  - There is a partial order $\subseteq$ in the cpo.
  - There is a bottom element, $\perp$.
  - Every chain $x_0 \leq x_1 \ldots$ has a least upper bound (lub) y:

    $x_i \subseteq y$ upper bound
    $y \subseteq z$ for any upper bound z.

- A solution of $x = F(x)$ is a fixed point of F.

  Wanted: the least fixed point, $lfp(F)$, according to $\subseteq$.

Kleene-Scott Theorem

- F is *continuous* means:
  For every chain C, $F(lub(C)) = lub(F(C))$.

- Theorem: Given $x = F(x)$ where F is *continuous*:

$$lfp(F) = lub(F^i(W[\,]))$$

- That is, with

$$x_0 = \bot,\ x_{i+1} = F(x_i),$$

$$lfp(F) = lub(x_0, ..., x_i, ...)$$

In the current work

Specs form a complete partial order, where

- the order relation is subset order over specs, lub is set union,
- $\perp$ is the $W[\,]$,
- F, a smooth transformer is always continuous.
- Proposition: $lfp(F)$ is the expected outcome in an execution.

Example: $stut()$

- Recursive equation: $x = H[Heads]_* \cup cons(Tails, x)$
- $lfp(stut) = \{H[Tails^j, Heads] \mid j \geq 0\}_*$
- This is not the correct solution.
  Does not include the infinite trace.

The fixed point should include the limit of all trace chains.

Upward Closure

- Two notions of chains:
  - specs: $p_0 \subseteq p_1 \ldots$. lub is the union of the p_is.
  - traces: Chain $C = t_0 \leq t_1 \ldots$.
- Limit of the trace chain, $lim(C)$, is a trace.
  Shortest trace that includes every t_i as a prefix.
- Define upward closure of spec p as
  $p^* = p \cup \{lim(C) \mid C \text{ a chain in } p\}$
- Follows: for specs, $(p \times q \cdots \times r)^* = p^* \times q^* \cdots \times r^*$

least upward-closed fixed point (*lufp*)

- For recursive equation $x = F(x)$,
  the least upward-closed fixed point p is a spec such that:

$p = F(p)$	fixed point
$p = p^*$	upward-closed
$p \subseteq q$	for any upward-closed fixed point q

  Note: p is a spec, so downward-closed.

- $lufp(F)$ may not exist for smooth F.

least upward-closed fixed point ($lufp$)

- For recursive equation $x = F(x)$,
  the least upward-closed fixed point p is a spec such that:

  $p = F(p)$ fixed point

  $p = p^*$ upward-closed

  $p \subseteq q$ for any upward-closed fixed point q

  Note: p is a spec, so downward-closed.

- $lufp(F)$ may not exist for smooth F.

Bismooth Transformer

- Smooth: $f(p_*) = f_*(p)$, for any traceset p
- Bismooth:

  Smooth (preserve downward-closure)

  Spec p: $f(p^*) = f^*(p)$ (preserve upward-closure)

Fairness is smooth but not bismooth.

Unfair merge is bismooth, fair merge only smooth.

Continuous filter is bismooth, discontinuous filter only smooth.

All other transformers seen so far are bismooth.

Bismooth Transformer

- Smooth: $f(p_*) = f_*(p)$, for any traceset p
- Bismooth:

  Smooth (preserve downward-closure)

  Spec p: $f(p^*) = f^*(p)$ (preserve upward-closure)

Fairness is smooth but not bismooth.

Unfair merge is bismooth, fair merge only smooth.

Continuous filter is bismooth, discontinuous filter only smooth.

All other transformers seen so far are bismooth.

lufp of bismooth transformer

Theorem: For bismooth F, $lufp(F) = lfp^*(F)$

Revisit $stut()$

- Recursive equation: $x = H[Heads]_* \cup cons(Tails, x)$
- $lfp(stut) = \{H[Tails^j, Heads] \mid j \geq 0\}_*$
- $lufp(stut())$
  $=$ {From theorem}
  $lfp^*(stut())$
  $=$ { $lfp(stut) = \{H[Tails^j, Heads] \mid j \geq 0\}_*$ }
  $(\{H[Tails^j, Heads] \mid j \geq 0\}_*)^*$
  $=$ {computing}
  $\{H[Tails^j, Heads] \mid j \geq 0\}_* \cup \{D[Tails^\omega]\}$
- Toss of a fair coin:
  $fair\_filter(\{H[Tails^j, Heads] \mid j \geq 0\}_* \cup \{D[Tails^\omega]\})$
  $=$ {Definition of $fair\_filter$ }
  $\{H[Tails^j, Heads] \mid j \geq 0\}_*$

Revisit $stut()$

- Recursive equation: $x = H[Heads]_* \cup cons(Tails, x)$
- $lfp(stut) = \{H[Tails^j, Heads] \mid j \geq 0\}_*$
- $lufp(stut())$
  = {From theorem}
  $lfp^*(stut())$
  = { $lfp(stut) = \{H[Tails^j, Heads] \mid j \geq 0\}_*$ }
  $(\{H[Tails^j, Heads] \mid j \geq 0\}_*)^*$
  = {computing}
  $\{H[Tails^j, Heads] \mid j \geq 0\}_* \cup \{D[Tails^\omega]\}$
- Toss of a fair coin:
  $fair\_filter(\{H[Tails^j, Heads] \mid j \geq 0\}_* \cup \{D[Tails^\omega]\})$
  = {Definition of $fair\_filter$ }
  $\{H[Tails^j, Heads] \mid j \geq 0\}_*$