Ruby Basics

15 January 2019 20:17

Higher Order Functions:

- Ruby (Rebecca)
- map n func : (n=items in array)
- Map n [f,g] : parallel composition
  - Don't worry about data flow direction

Component Classification:

- **$wire vs $rel**
- current = **VAR** x **.** <input> $rel **(**output**) .**
- Order of I/O
  - **Bottom-Up / Left-Right rule**
- Duplicate Input
  - **`fork`**
- **Series - ;**
- **Parallel - [,]**
- **Law - [(P;Q) , (R;S)] = [P,R] ; [Q,S]**

Compile:

rc file.rby

- compile file.rby into current.rbs

Evaluate:

re "sim-data"

- simulate current.rbs
- input sim-data with actual data

Notes

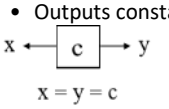
- many examples can be found in course homepage, e.g.
  - https://www.doc.ic.ac.uk/~wl/teachlocal/cuscomp/rebecca/egs.rby
- make sure you get the brackets (the types) right
  - e.g. the domain/range for [[P,Q,[R,S]],T]: <<p,q,<r,s>>,t>
- if `R` is a function, then `x R y` can be written as `y = R x`
- in Rebecca, use `` `R` x `` only when `R` is a function
  - e.g. `inc3' = VAR x . (`inc; inc; inc` x) $rel x` (there is a more concise way of defining inc3')
- `(((x))) = x` but `(x, x)` is illegal; use `<x, x>`
- LET cannot be used in `$rel` or `$wire` definitions
  - also cannot have local functions using `LET`:
    `g = LET both f = fork;f IN ...` is illegal

More Ruby

22 January 2019 12:13

Constants:

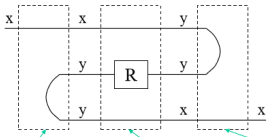
- Function takes in input
- Outputs constant



x = y = c

Converse:

- $xR^{-1}y <> yRx$
- Rebeca : $\boldsymbol{R\text{^}\sim 1}$



R-1 = x $wire <y, y, x> ; [id, R, id] ; <x, y, y> $wire x .

- **Laws of Converse**
  - $(R^{-1})^{-1} = R$
  - $(Q;R)^{-1} = R^{-1};Q^{-1}$
  - $[Q,R]^{-1} = [Q^{-1},R^{-1}]$
  - $R;R^{-1} \neq id$

Repeated Series Composition:

- Rebeca: $\boldsymbol{R\text{^}n}$
- R^n = IF (n $eq 0) THEN id
  ELSE (R ; R^(n-1)).

Wiring Patterns:

- **append** m n
- M fork n
- **apl** n
  - append 1 element to left
- **apr** n
  - to right of list
- **zip** n
- **tran** m n
  - transpose

Conjugate:

- $P^{-1};Q;P$
  - $Q \setminus P$

Repeated Parallel Composition:

- Rebecca: $\boldsymbol{map\ n\ R}$
- $map_{n+1}\ R = apl_n^{-1};[R, map_n R]\ ;apl_n$
  - input n+1
    - split
  - output n+1
- $map_{n+1}R = [R, map_n R]\ \setminus apl_n$

Types:

- Series Composition
  - $R: X\sim X, R^n: X\sim X$
- Parallel
  - $R: X\sim Y, map_n R: < X >_n \sim < Y >_n$
- Append
  - $<< X >_m, < X >_n > \sim < X >_{m+n}$

Trees:

- want $btree_n R: < X >_m \sim X,\ m = 2^n$
- $half: < X >_{2n} \sim << X >_n, < X >_n >$
  - in prelude.rby

- $\boldsymbol{btree_{n+1}R = half_m\ ;[btree_n R, btree_n R];R}$
- examples for uses
  - sum a list of numbers
  - find max in a list of numbers

Triangular Arrays:

- $\Delta_n R = [R^0, R^1, \dots, R^{n-1}]$
-

Grid & Combinators

< <LEFT>, <TOP> >~< <BOTTOM>, <RIGHT> >

Beside:

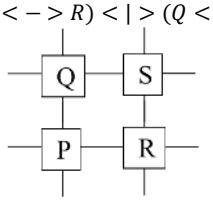
- $Q <-> R$
- fsth , sndh

Below:

- $Q <|> R = \left(Q^{-1} <-> R^{-1}\right)^{-1}$
- fstv, sndv

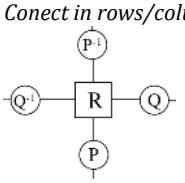
Laws:

- $(P <|> Q) <-> (R <|> S)$
- $= (P <-> R) <|> (Q <-> S)$



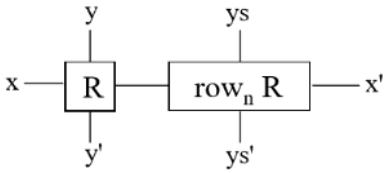
Transposed Conjugate:

- $\boldsymbol{R} \setminus [\boldsymbol{P}, \boldsymbol{Q}] =^{def} [Q, P]^{-1}; R; [P, Q]$
  - *Conect in rows/columns/grid*



Row: Repeated beside

- create list with one element
  - $x[-] < x >$
- $row_1 R =^{def} snd[-]^{-1}; R; fst[-]$
  - $= R \setminus\setminus fst[-]$
- $row_{n+1} =^{def} \boldsymbol{snd\ apl_n^{-1}}; (\boldsymbol{R} <-> \boldsymbol{row_n R})\ ; \boldsymbol{fst\ apl_n}$
  - $= \left(R <-> row_n R\right) \setminus\setminus fst\ apl_n$



Column:

- $fst\ apr_n^{-1}; <|>; snd\ apr_n$
- or conjugate of row

Reasoning & Proof

29 January 2019 09:23

Algebraic law: pointwise proof

- involves introducing I/O variables
- make use of
  - def //el
  - def ;
  - other algebra laws

Inductive Step: pointfree

- Base case: 0 or 1
- Use binary operator for **case(n+1)**
  - assuming case(n)

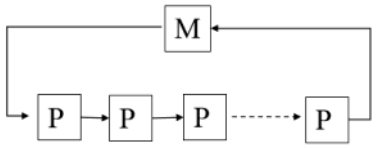
Reduction:

- $rdl\ n$
  - left reduction
  - drop left/bottom signal
- $rdr\ n$
  - right reduction
  - drop top/right signal

Sequential Design & Pipelining

29 January 2019 09:24

Systolic Array:



M: Memory
P : Pipelined Processor

Relate streams:

$< \cdots, x_{t-1}, x_t, x_{t+1}, \ldots > inc \ < \cdots, y_{t-1}, y_t, y_{t+1}, \ldots >$
$< \cdots, < \boldsymbol{x_t}, \boldsymbol{y_t} >, \ldots > add \ < \cdots, x_t + \ y_t, \ldots >$

Delay:

- $< \cdots, x_{t-1}, x_t, x_{t+1}, \ldots > D \ < \cdots, x_{t-2}, x_{t-1}, x_t, \ldots >$
- Rebecca
  - D

- initialised delay
  - $DI\ C$
  - $y_0 = C$

- anti-delay
  - $AD = D\hat{}{\sim}1$

Pipelining:

- increase throughput
- reduce power consumption
  - *less glitches*
- may increase
  - area
  - clock power consumption
  - latency

***Graphical* Method:**

- domain connection - put a D
- range connection - put a D^{-1}
- make sure R is **timeless**
  - $D; R = R; D \ or \ R = D^{-1}RD$
- internal D implementable
- I/O D's can be non-implementable
- contours
  - row
    - horizontal contour
  - column
    - vertical contour
  - grid
    - diagonal contour
- **Final design**
  - cancel out $D^{-1}{}'s$
  - stick $\max\left(\#D_O^{-1}\right)$
    - onto every output

Pipeline Control

31 January 2019 09:11

$dstl$

Distribute L

Add x to left of each element in a list

Design Tree:

- relate designs by transformation
- root
  - obvious, but inefficient
- leaves
  - efficient, not obvious

Characterise Designs

- features and components
- cells and registers
  - size, power, latency
- latency
  - longest path from any input to any output
- critical path
  - slowest combinatorial circuit
- problems
  - ignoring wire length and word-size

Control Pipelining:

- clustering
  - if $R\ is\ timeless$
  - $R^{KN} = \left(R^K; D\right)^N; D^{-N}$
  - fully-pipelined
    - $K = 1 \quad , N = KN$
    - **1 'K'-luster**
  - non-pipelined
    - $K = KN, N = 1$
- connect cluster in rows
  - $row_{KN}R = row_K\left(row_N R\right) \backslash\backslash \left(group_{KN}^{-1}\right)$
  - $< x >_{mn} group_{mn} \ll x >_n >_m$

Slowdown:

- replace each D with D^N
- sampler
  - $x\ ev_n y \Rightarrow \forall t, y_n = x_{nt}$
  - properties
    - $ev_n^{-1}; ev_n = id$
    - $x\left(ev_n; ev_n^{-1}\right)y \Rightarrow \forall t, x_{nt} = y_{nt}$
    - $ev_n; D = D^n; ev_n$
- slow
  - $slow_n R = ev_n; R; ev_n^{-1} = R \backslash \mathrm{ev}_n^{-1}$
  - if R combinatorial
    - $slow_n R = R$
  - $slow_n D = D^n$
  - $slow_n(Q; R) = slow_n Q; slow_n R$
  - $slow_n[Q, R] = \left[slow_n Q, slow_n R\right]$
  - *can use before adding pipeline*

Pipeline Strategies

05 February 2019 17:22

Design:

- *increase regularity at **bit-level***
- able to repeat components
- leads to grid design for **word level**

Cluster a grid:

- $grid_{mp\ nq}R = \left(grid_{mn}\left(grid_{nq}R\right)\right) \backslash\backslash \left[group_{mn}, group_{nq}\right]^{-1}$

- pipeline
  - around the sub-grids
  - *through* the sub-grids
    - lead grid

Lead and trail arrays:

- lead - **main diagonal different** block
  - $lead\ n\ R\ Q$

Summary: Clustering and Pipelining

- given $R = R \setminus D^{-1}$
- general form:
  - CC1 R = θ1 ; CC2 (CC3 P) ; θ2 clustering
  - CC2 (CC3 P) = τ1 ; CC2 Q ; τ2 pipelining

CC1 R	CC2 (CC3 p)	Q	θ1	θ2	τ1	τ2
R^{kn}	$(R^{k})^{n}$	R^{k} ; D	id	id	id	D^{-n}
row_{kn} R	row_{k} (row_{n} R)	row_{n} R ; snd D	snd $group_{k\,n}$	fst $group_{k\,n}^{-1}$	Δ (ΔD)	$[\Delta D^{-1}, D^{-n}]$
$grid_{mp\ nq}$ R	$grid_{m\,n}$ ($grid_{p\,q}$ R)	$grid_{p\,q}$ R ; D	[$group_{n\,q}$, $group_{m\,p}$]	[$group_{m\,p}^{-1}$, $group_{n\,q}^{-1}$]	[$\tilde{\Delta}$D, ΔD]	[ΔD;D^{n}, $\tilde{\Delta}$D;D^{m}]$^{-1}$
$grid_{mk\ nk}$ R	$grid_{m\,n}$ ($trail_{k}$ R R)	$trail_{k}$ R (R;D)	[$group_{n\,k}$, $group_{m\,k}$]	[$group_{m\,k}^{-1}$, $group_{n\,k}^{-1}$]	[$\tilde{\Delta}$D, ΔD]	[ΔD;D^{n}, $\tilde{\Delta}$D;D^{m}]$^{-1}$

State Machines

11 February 2019 20:58

Loop:

- $x(loopR)y : < x, \boldsymbol{s} > R < \boldsymbol{s}, y >$

Simples State Machines:

- counter: $loop(add; DI\ 0; fork)$

Looping a Row:

- $loop\ \left(row_k R; fst(map_k Q)\right)$

Decomposing a Loop:

- $loop(P <-> Q) = loopP\ ;\ loopQ$
- $loop(row_n R) = (loopR)^n$

Steps:

- Start with S1
- Transform S1 to S2 so that
  - $loop\ (S2; fstD)$
  - computes desired function
- Initialise registers
- Decompose into smaller state machines
- Pipeline the design

Intro

15 February 2019 10:23

SIMD vs vector:

- Array - one instruction to many FUs
- Vector - instruction applied to vector registers
  - e.g. $VR3 = VR2\ VOP\ VR1$

SoC Interconnect:

- Traditional bus
- Network on chip

Product cost:

- $K = K_f + 0.1K_f\sqrt[3]{n} + K_v n$
- Fixed costs
- Support costs
- Variable costs

Die Cost

15 February 2019 10:49

Yield:

Wafer of diameter d, single chip area of A:

$$N = \frac{\pi d^2}{4A} \quad [Gross\ Yield]$$

$N_G - good\ dice$
$N_D - number\ of\ defects\ on\ a\ \boldsymbol{wafer}$

$$[Yield] = \frac{N_G}{N} = e^{-\frac{N_D}{N}} = e^{-\rho_D A}$$

$N_D = \rho_D AN \ : \ \rho_D = defects/cm^2$

Area:

- Min feature size $f = 2\lambda\ (e.g. 2 * 7nm)$
- Smallest transistor
  - $4\lambda^2\ within\ a\ \boldsymbol{25\lambda^2}\ region$

Min Feature size	f	2λ
Register Bit Equiv.	rbe	$2700\lambda^2 = 675f^2$
A	A	$f^2 x 10^6 = 1481\ rbe$

Add padding to net area - giving Gross Die Area

Power

15 February 2019 11:31

$$P_{total} = \frac{CV^2 freq}{2} + I_{leakage} V$$

- Smaller C enables higher frequency
- Frequency linearly proportional to voltage

$$\frac{freq_1}{freq_2} = \sqrt[3]{\frac{P_2}{P_1}}$$

- Halving voltage
  - Halves frequency
  - Reduces power by **1/8**

Buffer Design

15 February 2019 11:53

Maximum-Rate:

- Mask the service latency
- Often full (expected high request rates / fixed data rate)
- Sized to avoid *runout*
- $BF = 1 + \left\lceil s.\frac{access\ time(cycles)}{p} \right\rceil$
  - s: $items\ consumed\ per\ cycle$
  - p: $items\ fetched\ in\ an\ acces$

Mean-Rate:

- Minimize probability of overflowing
- $Q = \ mean\ occupancy = mean\ rate * mean\ time\ to\ service$
- $BF = \min(Q/p\ , Q + \sigma/\sqrt{p})$
- $p_{overflow} = \min(Q/BF\ , \sigma^2/(BF - Q)^2\ )$

Intro & Streaming

03 March 2019 23:07

Host Code
- C
- calls functions generated by tools

Manager
- Java
- links kernel to IO

Kernel
- Design

<u>Kernel:</u>
- DFEVar
- io.input("label", type)
- io.output("label", DFEVar, type)
- Conditional choice
  - ternary operator
- Constants
  - constant.var(value, type)
- Java loops
  - unroll into series
  - or use to make parallel system

More advanced Designs

09 March 2019 23:04

Loop Counters:

- control.count.**simpleCounter**(N, bits)
- Complex counter
  - stride
  - wrap point
  - triggers
- Chained Counters
  - use for nested-loop behaviour
  - CounterChain chain = control.count.makeCounterChain();
    - DFEVar i = chain.addCounter(M, 1)
    - DFEVar j = chain.addCounter(N, 1)

Scalar Inputs:

- io.scalarInput("label", type)
- loaded once per stream
  - when calling function in C

Mapped ROMs/RAMs:

- Memory<DFEVar> mappedROM = mem.alloc(size, type)
- mappedROM.mapToCPU("mappedROM")

Stream Offsets:

- buffer stream on-chip for random access
- offset works on available stream inputs
  - DFEVar prev = **stream.offset(x,-1)**
- **Boundary** cases
  - using a counter, set conditional variables
  - use ternary to deal with edge cases
    - use above variables to select correct case

Vectors:

- DFEVectorType myVec = **new** DFEVectorType(size, Type)

Scheduling

09 March 2019 23:07

Stream scheduling algorithm:

- transform an abstract dataflow graph into one that produces the correct results given the latencies of the operations
- Can try to optimise for
  - Latency
  - Amount of buffering
  - Area

ASAP:

- Start from input
- Add buffering as soon as latency mismatch

ALAP:

- Start from output
  - Negative latencies
- Can change cycle inputs come in

Stream offsets:

- Wires with negative or positive latency
- Scheduler adds buffers to preserve correct functionality

Performance

09 March 2019 23:05

High-level Model:

- $Speedup = \frac{T_{old}}{T_{new}}$
- $T_{new} = a.\frac{T_{old}}{S} + (1 - a).T_{old}$
- $\boldsymbol{T_{accel} = T_{init} + T_{stream}}$
- T_{stream} dominates for **large data** streams
- $\boldsymbol{T_{init}}$
  - SW time
  - Config
  - Scalar inputs, ROMs
  - Prepare streams

$$T_{stream} = \mathbf{max}\left(\boldsymbol{T_{compute}}, \boldsymbol{T_{memory}}, \boldsymbol{T_{bus}}, \boldsymbol{T_{net}}\right)$$

- $T_{compute} = \frac{Cycles}{frequency}$
- $T_{bus} = \frac{size}{bus\ BW}$
  - Speed depends on transfer size
    - **Small size lower speed**
- $T_{memory} = \frac{reads + writes}{mem\ BW}$
  - $\boldsymbol{mem_{BW} = \frac{Bytes}{DIMM} * No.DIMMs * Freq * effc}$
    - $effc$ depends on
      - **Transfer size**
      - **Access pattern**

How to Improve:

- Compute - **papalism** - $\boldsymbol{Area_{chip} = Area_{Manager} + (Area_{kernel}.P)}$
- Bus
  - Use on-card memory
  - Compression, data representation - **trades compute area for BW area**
- Memory
  - Compression, data representation

Loops

09 March 2019 23:05

Loop Attributes:

- array access pattern
- loop-carried dependencies

Loop performance metrics:

- ratio of computation:memory
- bottleneck - CPU/mem/IO

Outer loop is implied in MaxJ kernels

Nested Loops:

- use a CounterChain
  - outer loop is first counter
- unrolling inner loops
  - if dependence
    - generate HW in series

Variable Length Loops:

- while or variable for
- find **max** number of executions
- HW for loop with max N
  - use conditional execution within loop

Loops with dependence:

- need to deal with pipeline initialisation
- Use simpleCounter
  - DFEVar count = control.count.simpleCounter(32, C)
  - DFEVar **carry = scalarType.newInstance(this);**
  - **use counter** to sum += **(count<C)?** 0 : carry
  - **carry.connect(stream.offset(sum, -C))**

- **Depth** and Loop-Carry dependence
  - need to avoid mixing dependence when previous still in pipeline
  - loop **interchange** to keep pipeline full