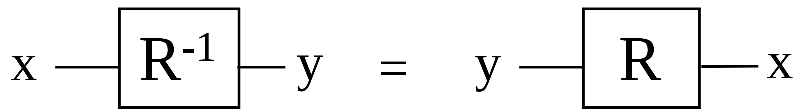
Converse, conjugate and repeated compose

- converse
  - $x\ R^{-1}\ y \;\Leftrightarrow\; y\ R\ x$
- conjugate
  - $Q \backslash P = P^{-1} ; Q ; P$
- repeated series composition
  - $R^0 \;=\; id$
  - $R^{n+1} = \; R ; R^n$
- repeated parallel composition
  - $map_0\ R \;=\; []$
  - $map_{n+1}\ R \;=\; apl_n^{-1} ; [R, map_n\ R] ; apl_n$
- binary tree (not in prelude)
  - $btree_1\ \ R \;=\; R$
  - $btree_{n+1}\ R \;=\; half_m ; [btree_n\ R, btree_n\ R] ; R,\;\; m=2^n$

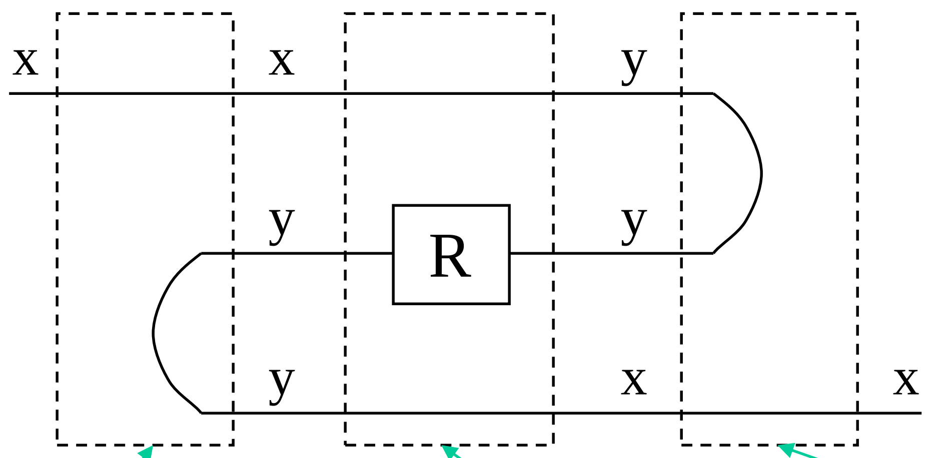
We need various combinators to describe different patterns of computation; they would all seem natural once you get used to them… be patient, just like playing Lego!

Converse

- $x\,R^{-1}\,y \iff y\,R\,x$
  - Rebecca: **`R^~1`**



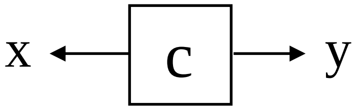
- expressed using wiring relations



R^{-1} = $\underbrace{\text{x \$wire <y, y, x>}}$; $\underbrace{\text{[id, R, id]}}$; $\underbrace{\text{<x, y, y> \$wire x}}$.

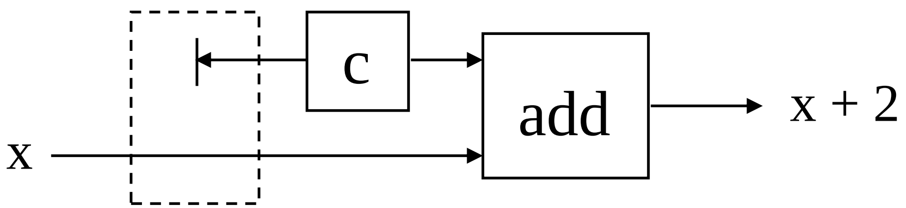
Dealing with constants

- $x \; c \; y \;\Leftrightarrow\; x = y = c$



$x = y = c$

- example: x add2 (x+2)

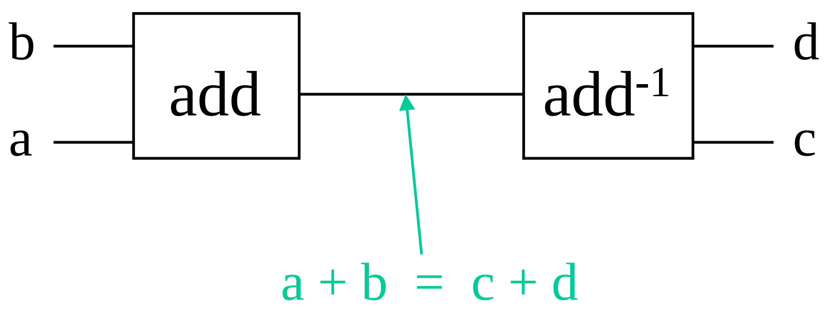


$add2 \;=\; \pi_1^{-1}$; snd 2 ; add

- Rebecca:
  - **`add2 = VAR x . x $rel (`add` <x,2>).`**
  - or **`add2 = pi1^~1 ; snd 2 ; add.`**

Laws of converse

- $(R^{-1})^{-1} = R$
- $(Q \,;\, R)^{-1} = R^{-1} \,;\, Q^{-1}$
- $[Q, R]^{-1} = [Q^{-1}, R^{-1}]$
- $R \,;\, R^{-1} \neq id$ in general, it usually denotes a constraint (e.g. when R = add)



Note: add ; add^{-1} is not supported by the rc compiler

Conjugate

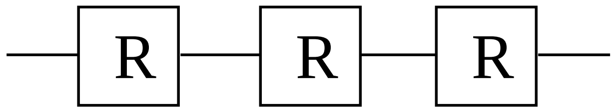
- the pattern P^{-1} ; Q ; P occurs often:
  abbreviates to Q \ P

- "Q conjugated by P"

- examples:
  - Q is word-level design, P maps word-level data to bit-level, Q\P is bit-level design
  - P converts rectangular to polar co-ordinates
  - P describes interface constraints

- P \ (Q ; R) = ?
  [P, Q] \ [R, S] = ?

Repeated series composition

- 

R^3

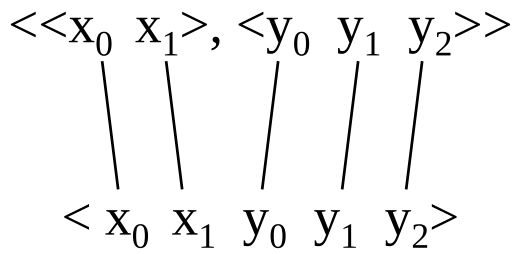
- recursive definition:
  - $R^0 = id, \quad R^{n+1} = R \,;\, R^n$

- Rebecca:
  - ```
    R^n  =  IF (n $eq 0) THEN id
                         ELSE (R ; R^(n-1)).
    ```
  - defined in prelude.rby, which contains useful definitions

- recursion unfolds at compile time
  - IF THEN ELSE, \$eq, +, **-** : compile time operators (LET or WHERE are not supported)
    ```

Wiring patterns

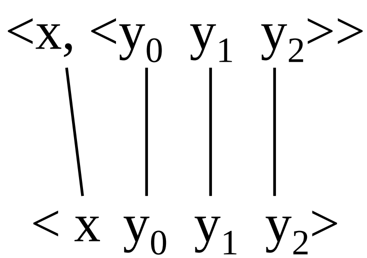
- $\text{append}_{m\ n}$



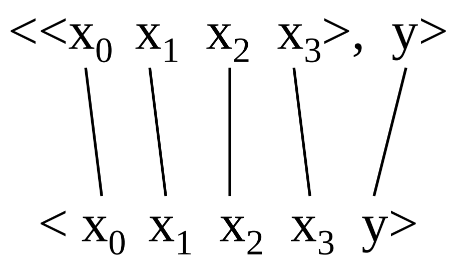
(like ++ in Haskell or Miranda)

$\text{append}_{2\ 3}$

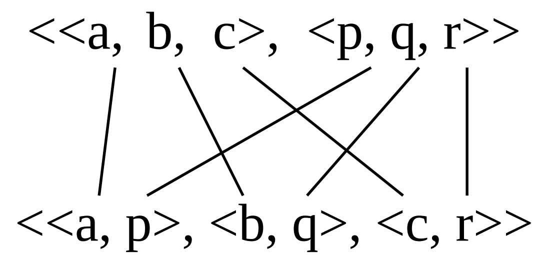
- other examples (some defined recursively: see prelude)



apl_3
(append left)

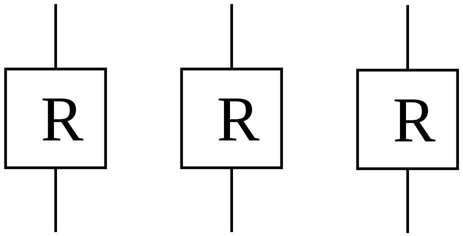


apr_4
(append right)



zip_3 or $\text{tran}_{2\ 3}$
(transpose)

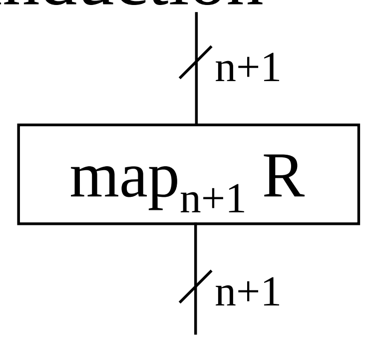
Repeated parallel composition

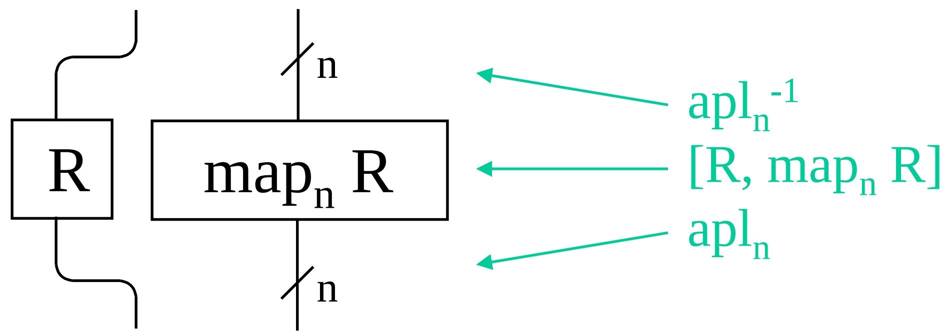
- 

  $$\mathrm{map}_3\ R \;=\; [R, R, R]$$

- recursive description:
  - base case: $\mathrm{map}_0\ R = []$

- induction





$$\begin{aligned} \mathrm{map}_{n+1}\ R \quad &= \quad \mathrm{apl}_n^{-1} ; [R, \mathrm{map}_n\ R] ; \mathrm{apl}_n \\ &= \quad [R, \mathrm{map}_n\ R] \setminus \mathrm{apl}_n \end{aligned}$$

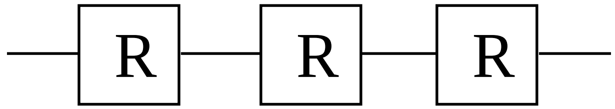
where $\quad Q \setminus P \;=\; P^{-1} ; Q ; P$

Types

- key to getting descriptions correct: shape of variables
- primitive types: bit, uint /sint (unsigned/signed integer), ureal / sreal (unsigned/signed real number)
- composite types: tuple or list
  - <<1,2>,3>: <<uint,uint>,int>
  - <2.31, -3.56, 7.08>: <sreal>$_3$
- types for primitive blocks
  - uadd: <uint,uint> ~ uint
  - Rebecca simulator: **`add`** can work out the type from data
- types for composite blocks
  - given Q: (A ~ B) and R: (B ~ C), (Q ; R): (A ~ C)
  - given Q: (A ~ B) and R: (C ~ D), [Q, R]: <A,C> ~ <B,D>

Types: repeated compositions

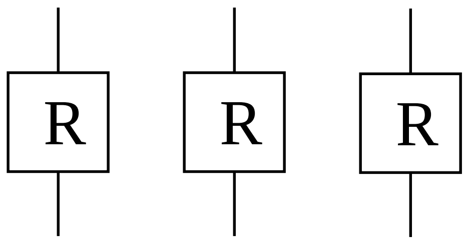
- in series:



R^3

given $R : X \sim X$, $R^n : X \sim X$

- in parallel:



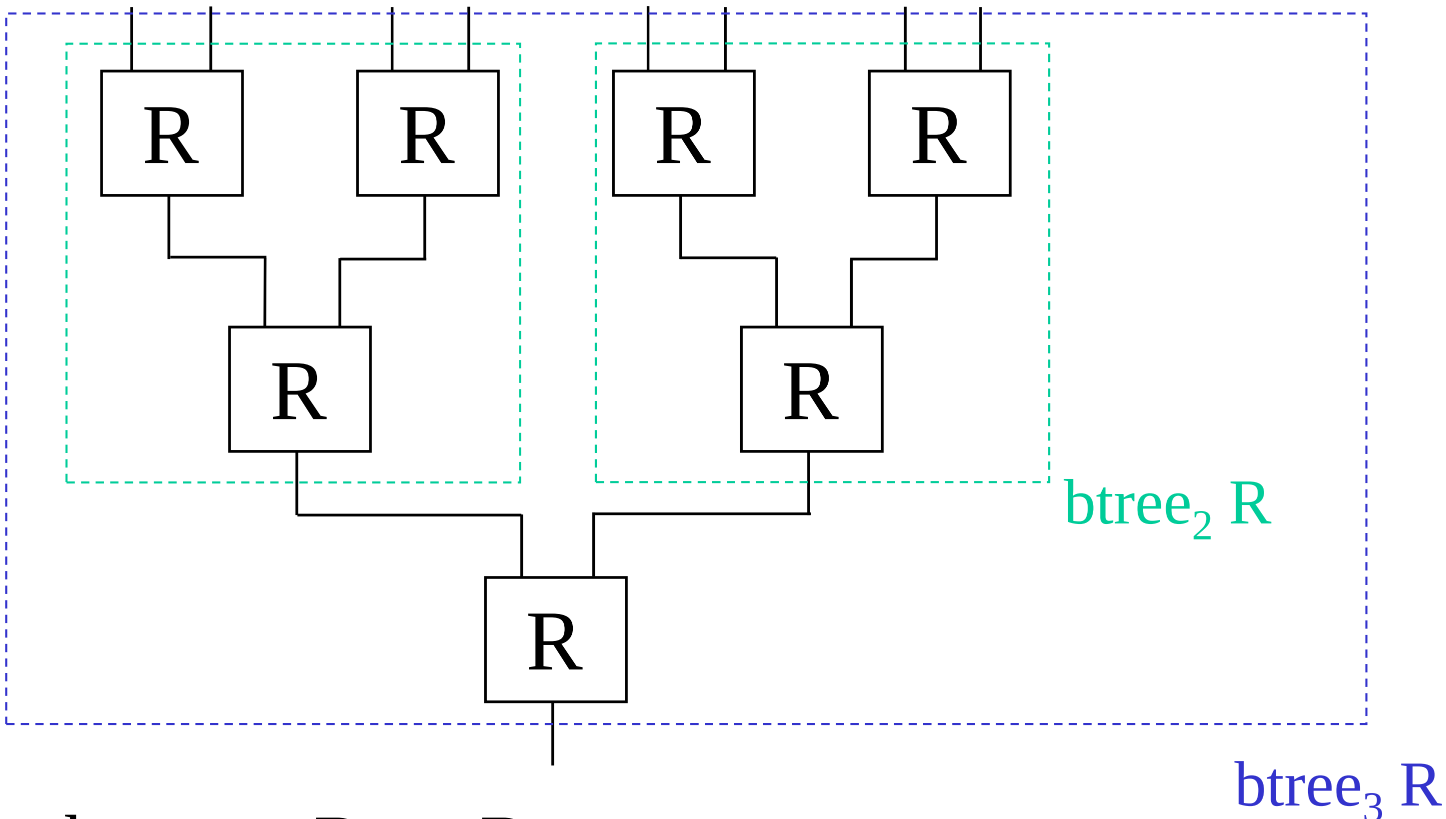
$map_3\ R = [R, R, R]$

given $R : X \sim Y$, $map_n\ R : <X>_n \sim <Y>_n$

- wiring patterns: $append_{m\ n} : <<X>_m, <X>_n> \sim <X>_{m+n}$

- Rebecca: no compile-time typechecker yet…

Tree-shaped array



- btree$_1$ R = R
  btree$_{n+1}$ R = [btree$_n$ R, btree$_n$ R] ; R
- type of btree$_n$ R, given R: <X,X> ~ X?

Tree: definition and examples

- given $R: \langle X,X\rangle \sim X$, want $\text{btree}_n\ R: \langle X\rangle_m \sim X$, $m=2^n$
- need wiring component
  - half: $\langle x\rangle_{2n} \sim \langle\langle x\rangle_n, \langle x\rangle_n\rangle$ in prelude.rby
- $\text{btree}_1\ R = R$

  $\text{btree}_{n+1}\ R = \text{half}_m ; [\text{btree}_n\ R, \text{btree}_n\ R] ; R$
- examples: find the sum or maximum of a list of numbers, use btree_n add or btree_n max
- example: inner product $z = \sum_{i<m} x_i \times y_i$
  - innerprod: $\langle \langle \text{uint},\text{uint}\rangle_m \rangle \sim$ uint
  - innerprod $= \text{map}_m$ mult ; btree_n add where $m=2^n$

Triangular array

- $\Delta_n\ R \ = \ [R^0, R^1, R^2, ..., R^{n-1}]$
  $= \text{if } n=0 \text{ then } []$
  $\text{else } [\Delta_{n-1}\ R,\ R^{n-1}] \setminus apr_{n-1}$
- flipped triangle
  - $\Delta^{\sim}_n\ R \ = \ [R^{n-1}, R^{n-2}, ..., R^0]$
- $\Delta_1\ R \ = \ ?$
  $\Delta_2\ R \ = \ ?$
- example: polynomial evaluation $\quad y = \sum_{i<m} a_i \times x^i$
  - polyeval: $\langle sreal \rangle_m \sim sreal$
  - polyeval $= \Delta_m\ (\text{mult1 } x)\ ;\ \text{btree}_n\ \text{add}$ $\quad$ where $m=2^n$

take x from external? See coursework…

Hints

- create directory, go there, follow the link <u>Rebecca setup</u> in *https://www.doc.ic.ac.uk/~wl/teachlocal/cuscomp*
- remember: include *prelude*, and *full stop* at end of definition
  - understand the idea behind each definition
- get brackets and domain/range variables right:
  - check domain and range data structures by compiling components in turn and inspect "current.rbs"
- note: **`R^~1`** and not **`R^-1`**,
  `` `add`<x,y> `` and not **`'add'<x,y>`**
- distinguish
  - compile-time operators: **`IF`**, **`$eq`**, ... (begin with capital or **`$`**)
  - run-time operators: **`xor`**, **`add`**, ... (begin with lower case)
- error location (line no.) produced by rc may be incorrect
- recursive definition: compilation may not terminate

Answer to Unassessed Coursework 1

1.
```
FAM = VAR x y . <x,y>
                 $rel
                <`add` <x,y>, `mult` <x,y>>.
```

```
or FAM = fork ; [add, mult].
```

2.
```
AM = VAR u v x y . << u, v>, <x, y>> $rel
                <`add` <u,v>, `mult` <x,y>>.
```

```
or AM = [add, mult].

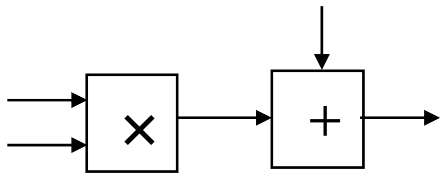
FAM4 =  AM ; FAM ; FAM ; FAM = AM ; FAM^3.
```

3.
```
fork3  =  x  $wire  <<x,x,x>, x>.
rsh    =  <x, <y,z>>  $wire  <<x,y>, z>.
twoadd =  VAR x y z . <x,<y,z>>
                 $rel (`add` <`add` <x,y>,z>).
```

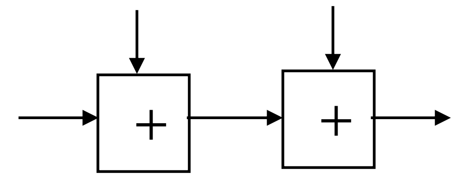
Unassessed Coursework 2

1. Describe the following without $rel. What are the types?

(a)



(b)



2. What are the types for conjugate, append, and triangle?

3. Develop a wiring block such that: $\langle X, \langle A \rangle_n \rangle \sim \langle\langle X, A \rangle\rangle_n$

then develop a design for computing polynomial evaluation:

$$y = \sum_{i<n} a_i \times x^i$$

simulate symbolically the design for n=8.