Concurrency and coordination in a streaming network

A.Shafarenko

Compiler Technology and Computer Architecture Group (CTCA)

http://ctca.feis.herts.ac.uk/

University of Hertfordshire

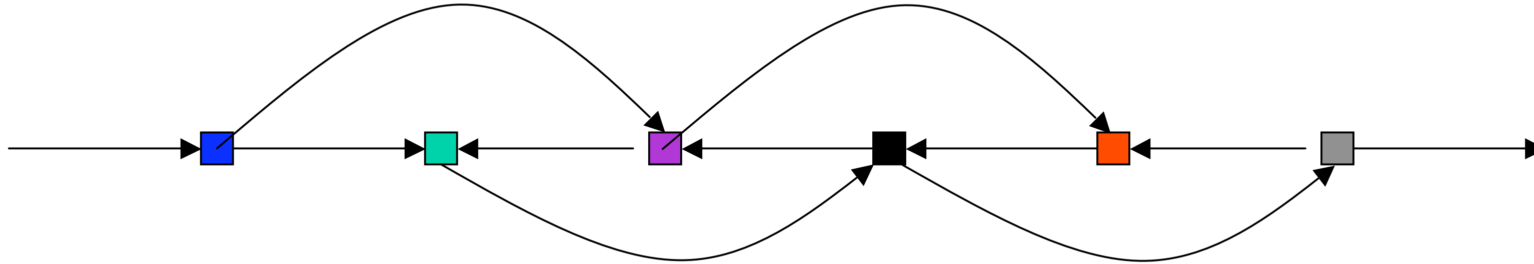
U.K.

Credits

Dr Sven-Bodo Scholz, CTCA	co-investigator
Dr Clemens Grelck, CTCA	implementation
Prof Walter Dosch, Uni Lübeck	semantics
Dr Juha Pärsinen, VTT Finland	graphic tools

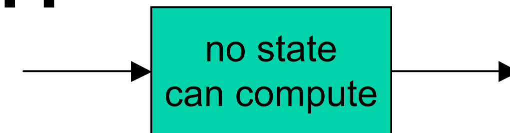
S.Herhut, F.Penzcek
(PhD students)

What is stream processing?

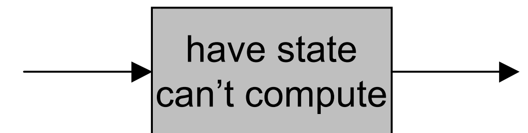


- Many definitions and no single point of view
- The idea of processing nodes connected by channels
- **Static** heterogeneous connections
  - Nodes are different (data parallelism inside nodes if necessary)
  - Connections established before run-time
- Nodes are causal
  - the output at any time depends only on the parts of the input streams read so far, and not on any subsequent stream elements
  - available memory at a node does not depend on the length of the input streams: processing “on the fly”

Our approach



- Coordination
  - Single-Input-Single-Output nodes
    - Treat nodes as user-defined application boxes and non-computing special boxes.
      - Application boxes are written in any language
    - Unified interface, a **single** synchronised **input** channel.
      - A function argument list
    - Triggered by a single input item, producing a (short) stream of items
    - A **single** (targeted) **output** channel.
      - A special procedure call for the output
    - User-defined boxes have no persistent internal state
  - Network defined in a special coordination language SNet
    - Describes how boxes are connected using an algebraic formula
    - Splits and merges streams
      - Deterministic and nondeterministic mergers
      - Structurally uniform: SISO to SISO
    - Takes care of synchronisation
    - Takes care of storage (state)
      - The only type of storage is synchronising memory
    - Does not compute



Our approach (ii)

- Type-directed
  - Streams of nonrecursive (flat) variant records
  - Records are treated as sets of labelled values
  - Box type signatures in terms of labels:

{a,b} –> {c,d,e}, {g}, {x,y}
{a,b,c} –> {g,w}
{v} –> {w}

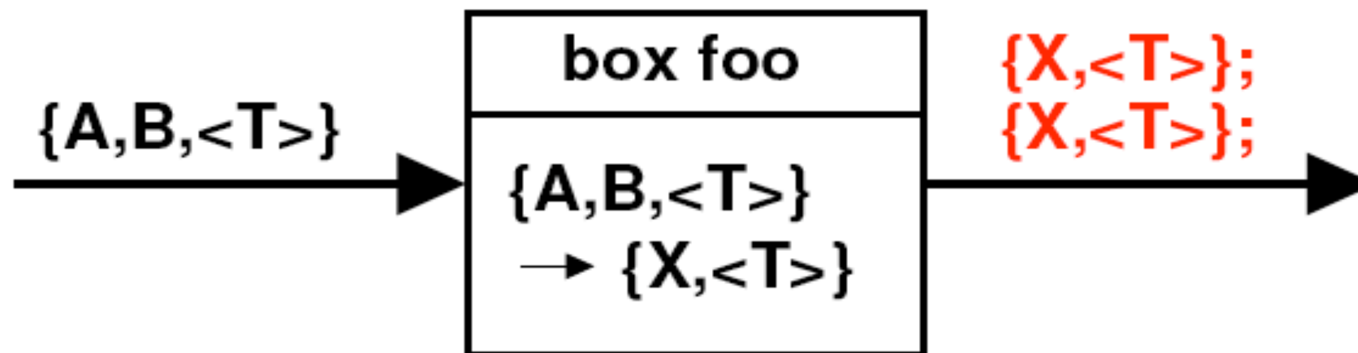
Input type: {{a,b},{a,b,c},{v}}, output type {{c,d,e},{g},{g,w},{w},{x,y}}
3 variants for input, 5 variants of output
Box can use any of output variants zero or more times
Box signature declared **in addition** to box code
box interface translates records to argument tuples
Value types can be captured, too, but are outside the scope of this talk

Our approach (iii)

- Subtyping
  - SNet has no awareness of the field values
  - A record type is a label set: {x,y,z}
  - Subtyping by subsetting
    - {a,x} can be used in place of {a}
    - Since no order of fields, any unwanted ones can be ignored
  - Variants are anonymous and not disjoint:
    - A network can expect {x,y} or {x,y,z}, and {x,y,z} < {x,y}: best match
    - Optional tags: {<a>,x,y} and {<b>,x,y,z} no longer subtypes
    - Tags may carry an integer value, which is **readable/writeable by SNet**
    - Tags are supported by box language interface
  - Subtyping rules for records: a subtype has
    - same or fewer variants
    - for each variant, same or more fields.

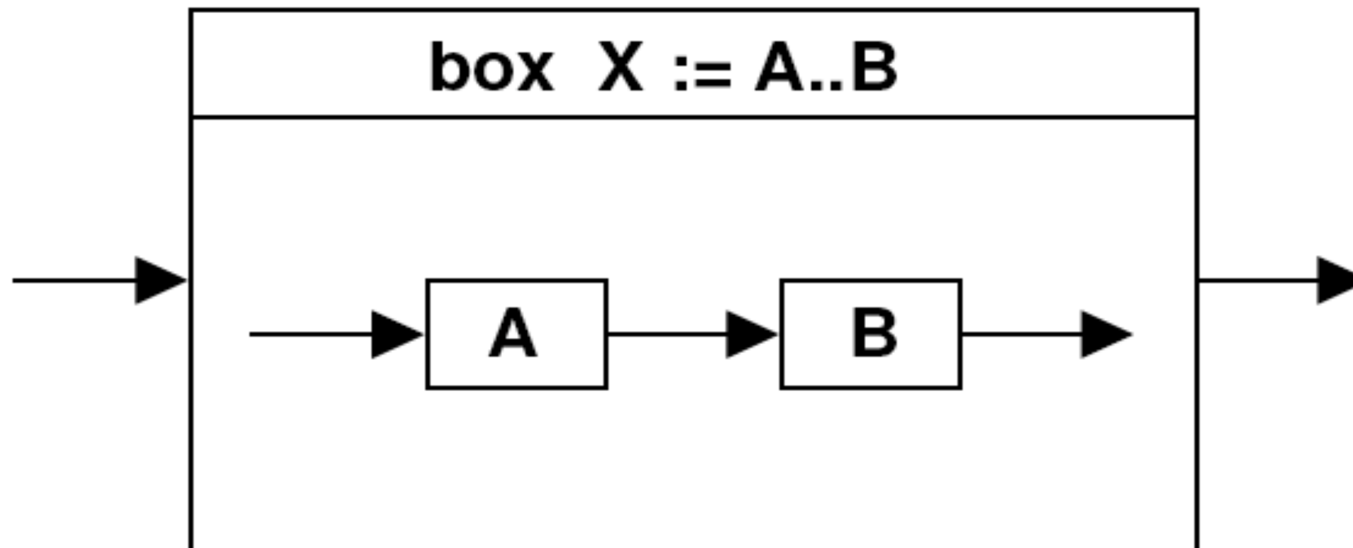
S-Net at a Glance

- central construct: box
- connected by
  - single input stream
  - single output stream
- streams transport records: sets of named fields
  - opaque value fields
  - integer-valued tag fields
- box behaviour declared by type signature
- behaviour defined in box language, not **S-Net**
- box maps single input record to stream of output records

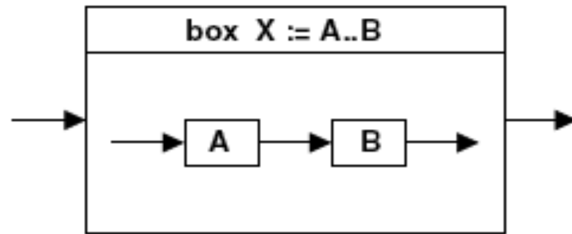


Network Combinators: Serial

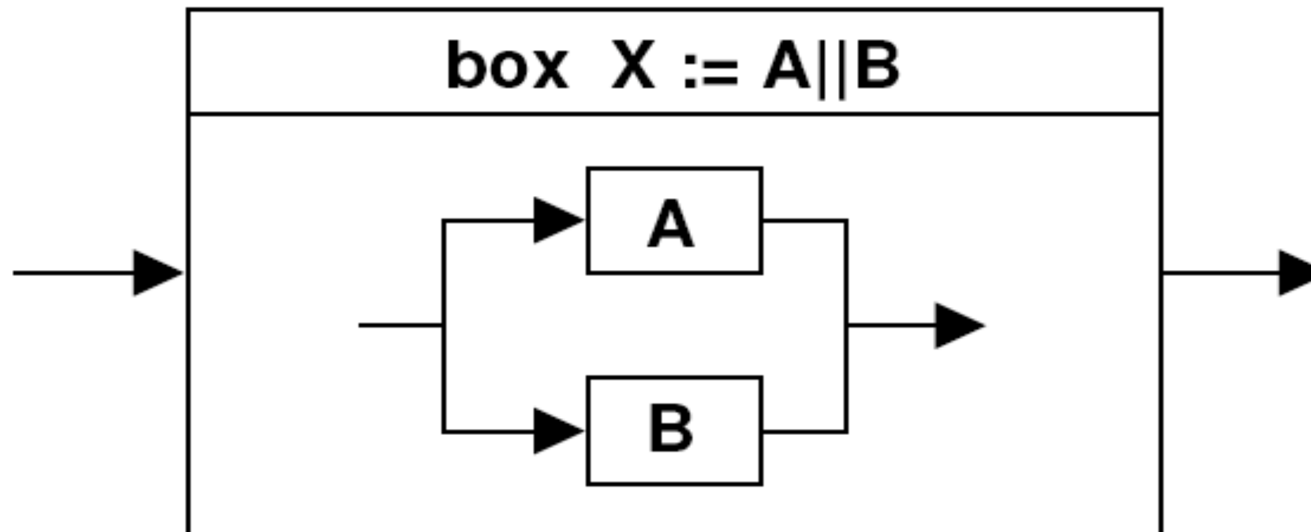
- A and B operate concurrently
- record -> A; out(A) -> B, one at a time



Network Combinators: Choice



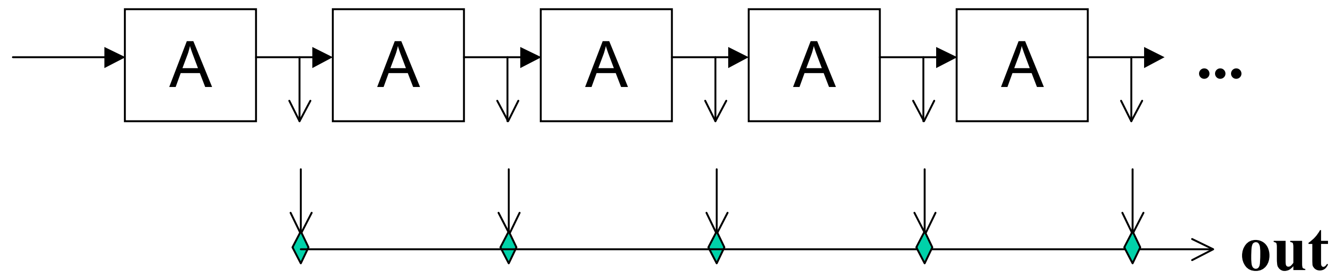
- A and B operate concurrently
- input matches A -> A; else matches B -> B
- Matches both -> **best match**
- Both matches are best -> n/d choice
- out(A) and out(B) merged out of order

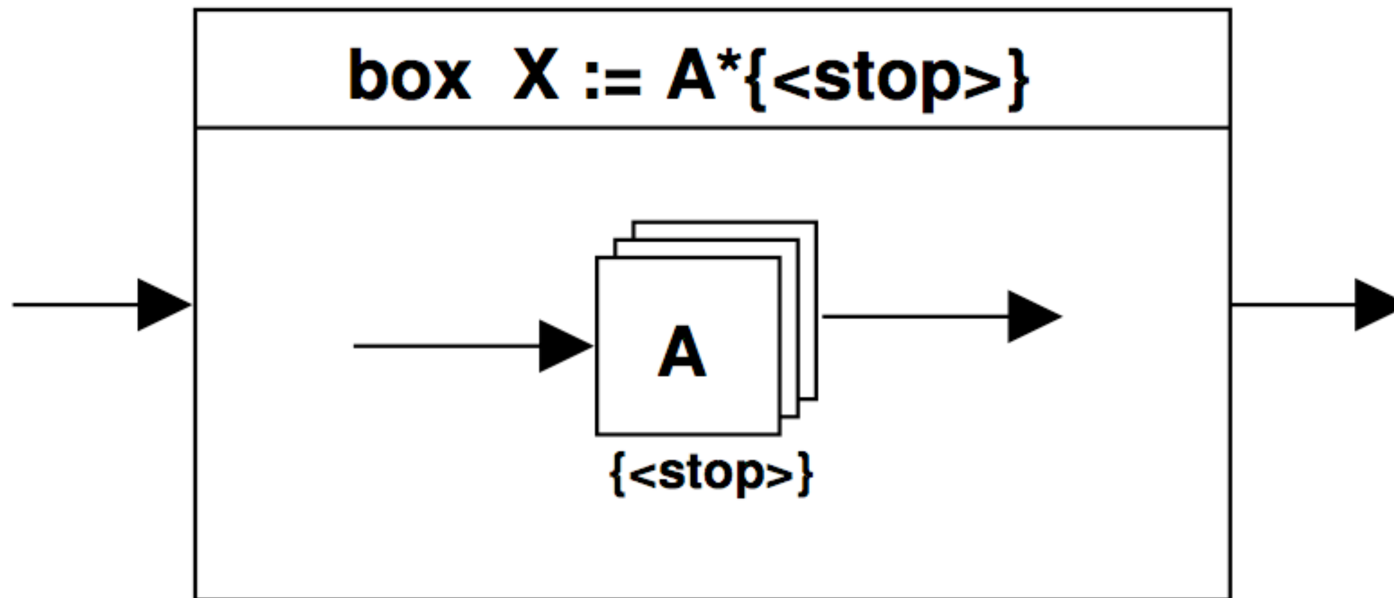


Single stream concurrency

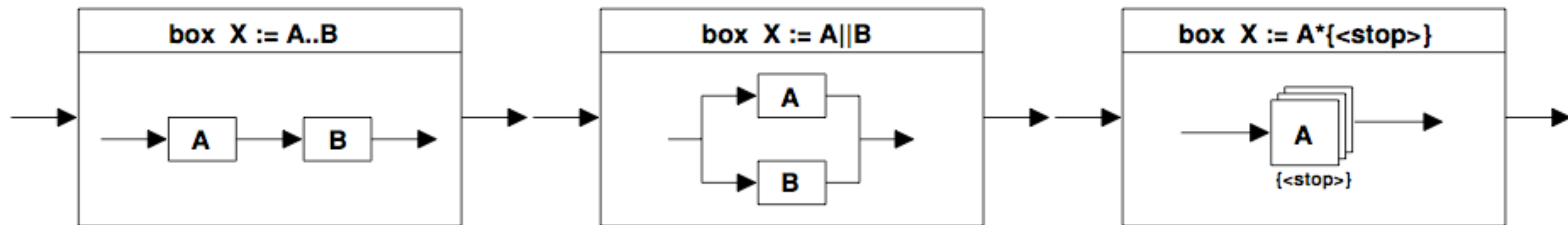
- Left: MIMO
  - Each channel blockable independently, state transitions inside
- Right: SISO
  - Merger nondeterministic, R gets an arbitrarily interleaved stream
  - No state transitions, must accept either kind of record
  - Even so, either substream can block the other one
  - Resources may currently be available for one substream
  - (implementation) Merger is a **demand-driven reordering buffer**

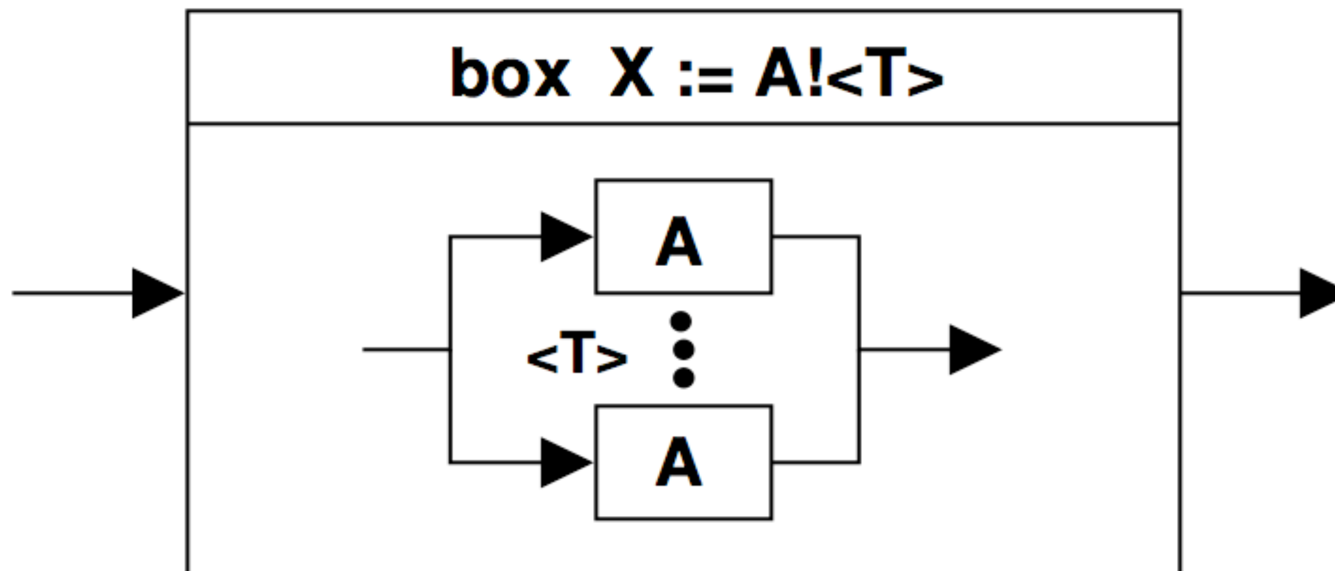
Network Combinators: Serial Replication





Network Combinators: Index Split

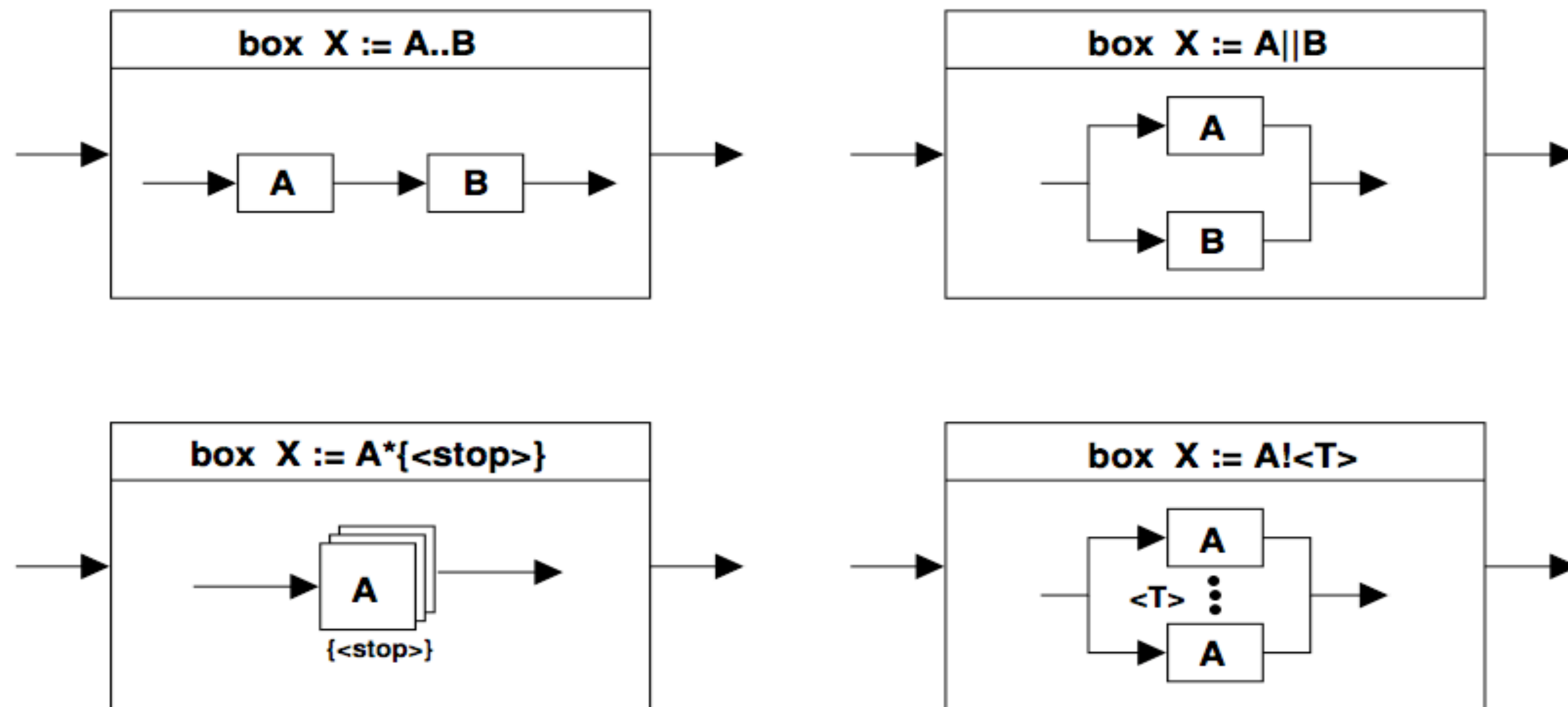




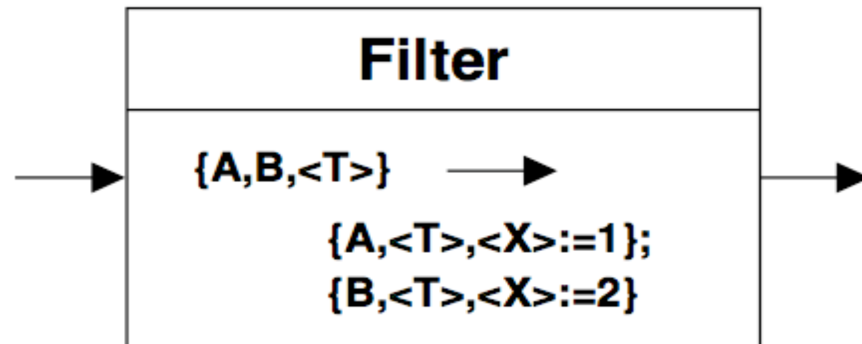
Concurrency control

- All combinators (except ..) are supported in two versions
- Deterministic
  - |, !, *
  - The merger joins the streams in-order
- Nondeterministic
  - ||, !!, **
  - The merger joins the streams out-of-order
- The .. combinator does not contain a merger, hence one version

Network Combinators: Summary



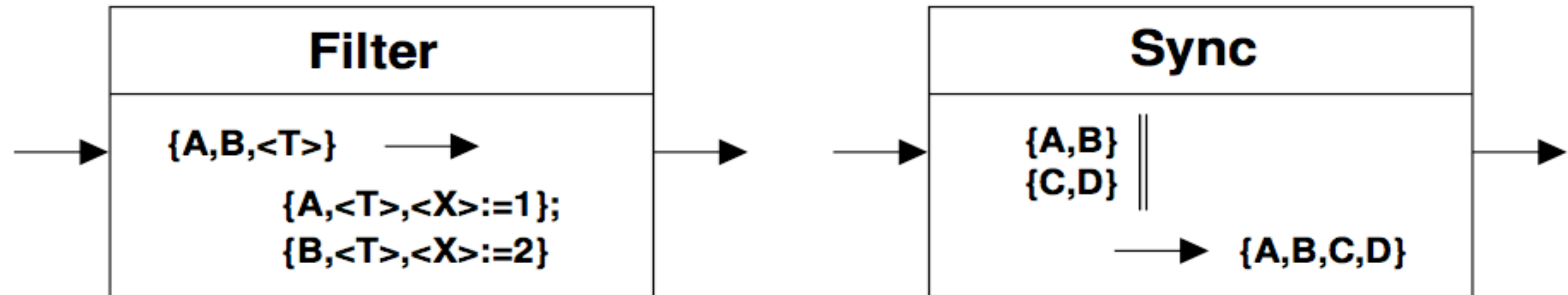
Primitive Boxes: Filter and Sync



housekeeping:

- eliminate record fields
- duplicate record fields
- add tags
- manipulate tag values
- ...

Primitive Boxes: Filter and Sync



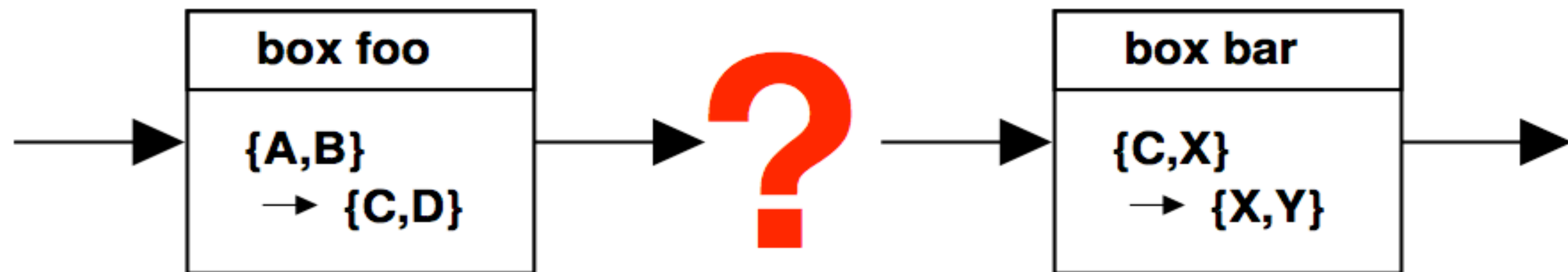
housekeeping:

- eliminate record fields
- duplicate record fields
- add tags
- manipulate tag values
- ...

synchronisation:

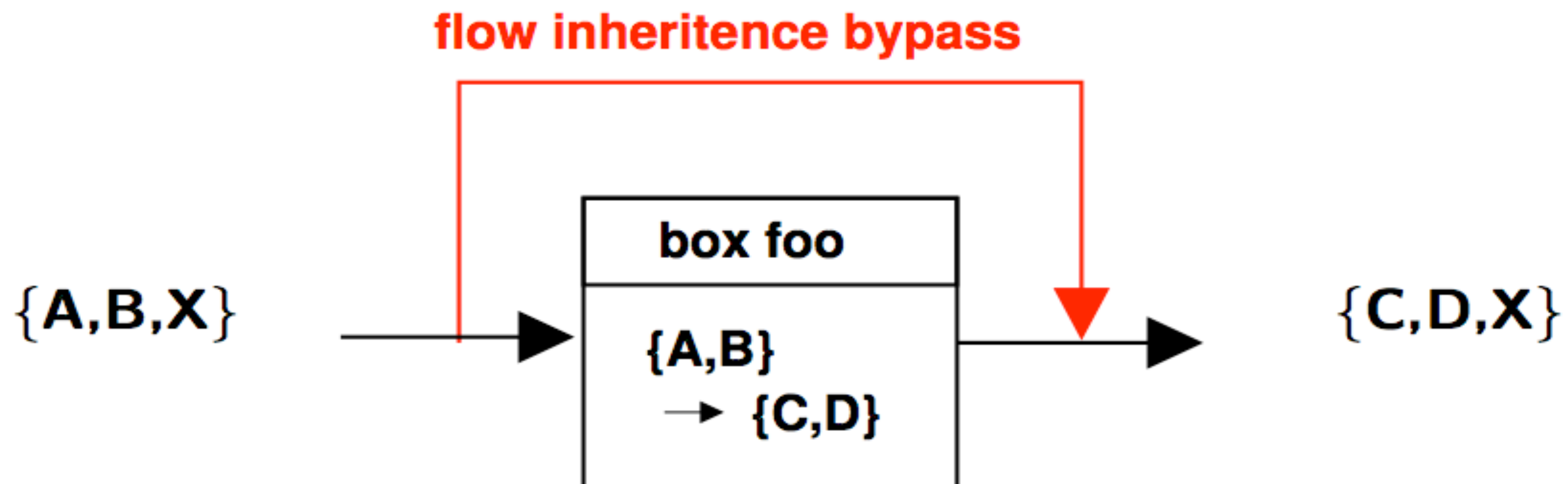
- keep record that matches pattern
- all patterns matched: release merged records
- pattern already matched: pass through

Problem: Incompatible Boxes



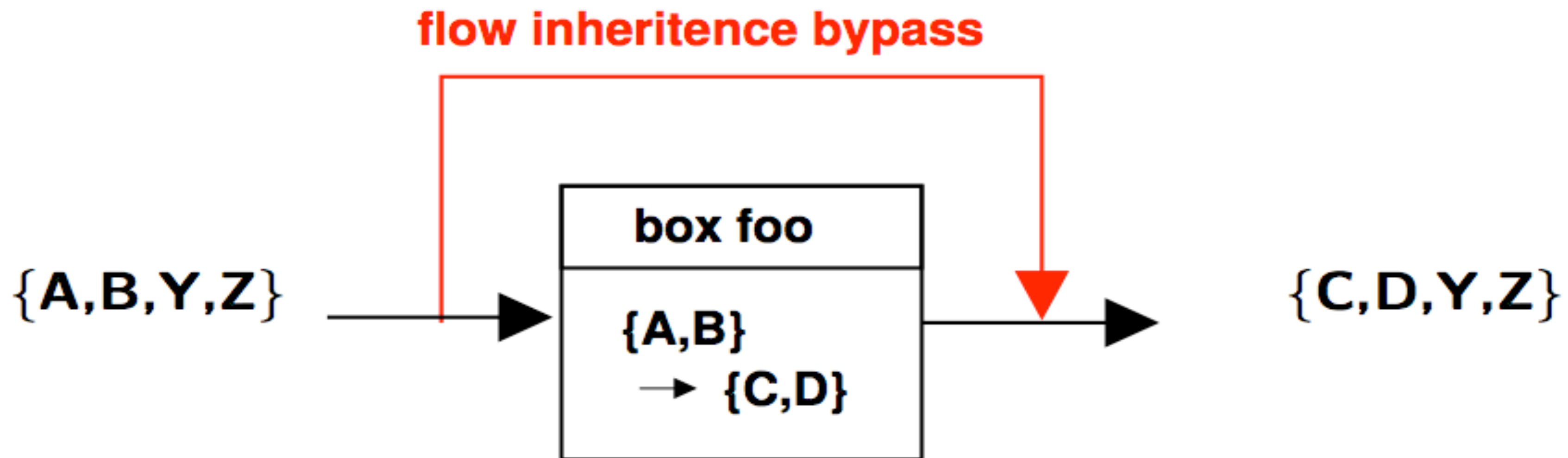
- Box code typically designed in isolation.
- Interfaces only partially overlap.
- Network composition unfeasible in practice?

Solution: Flow Inheritence



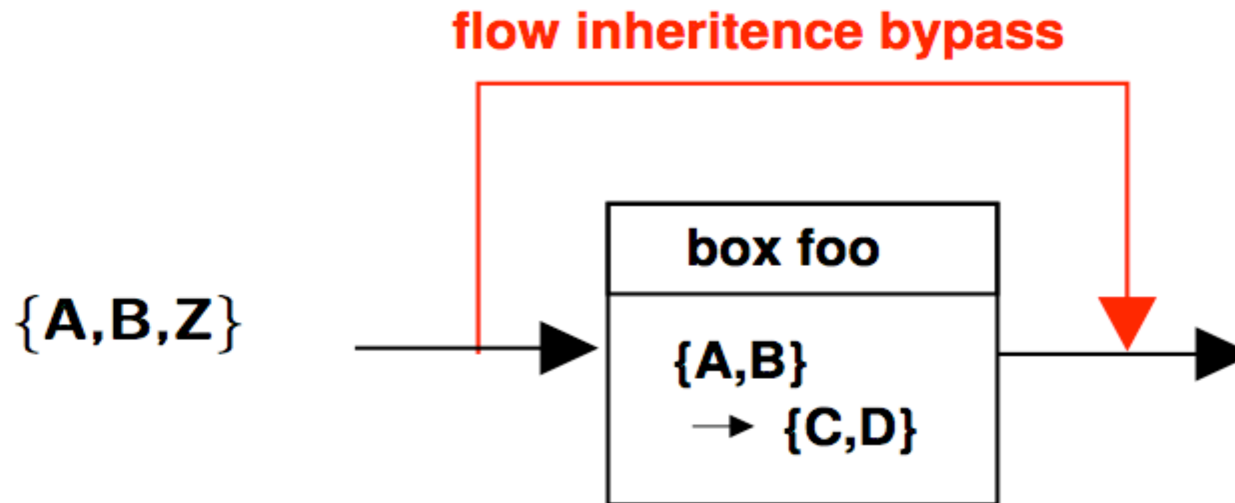
- Any "unwanted" field is bypassed to output channel
- and is attached to any record produced in response.

Solution: Flow Inheritence



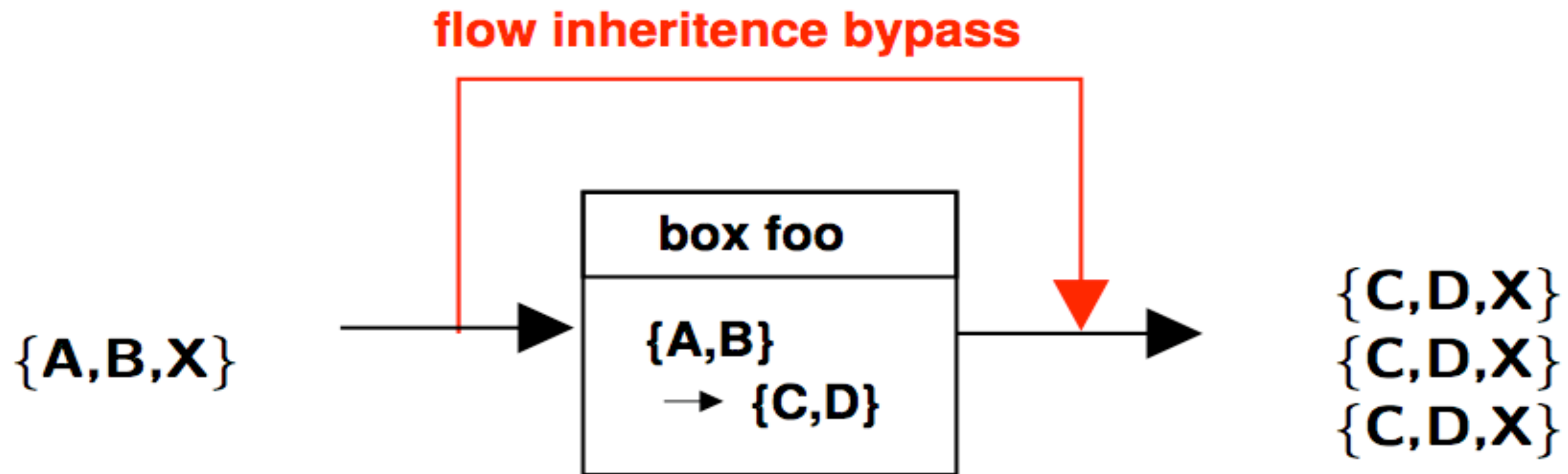
- Any "unwanted" field is bypassed to output channel
- and is attached to any record produced in response.

Solution: Flow Inheritence



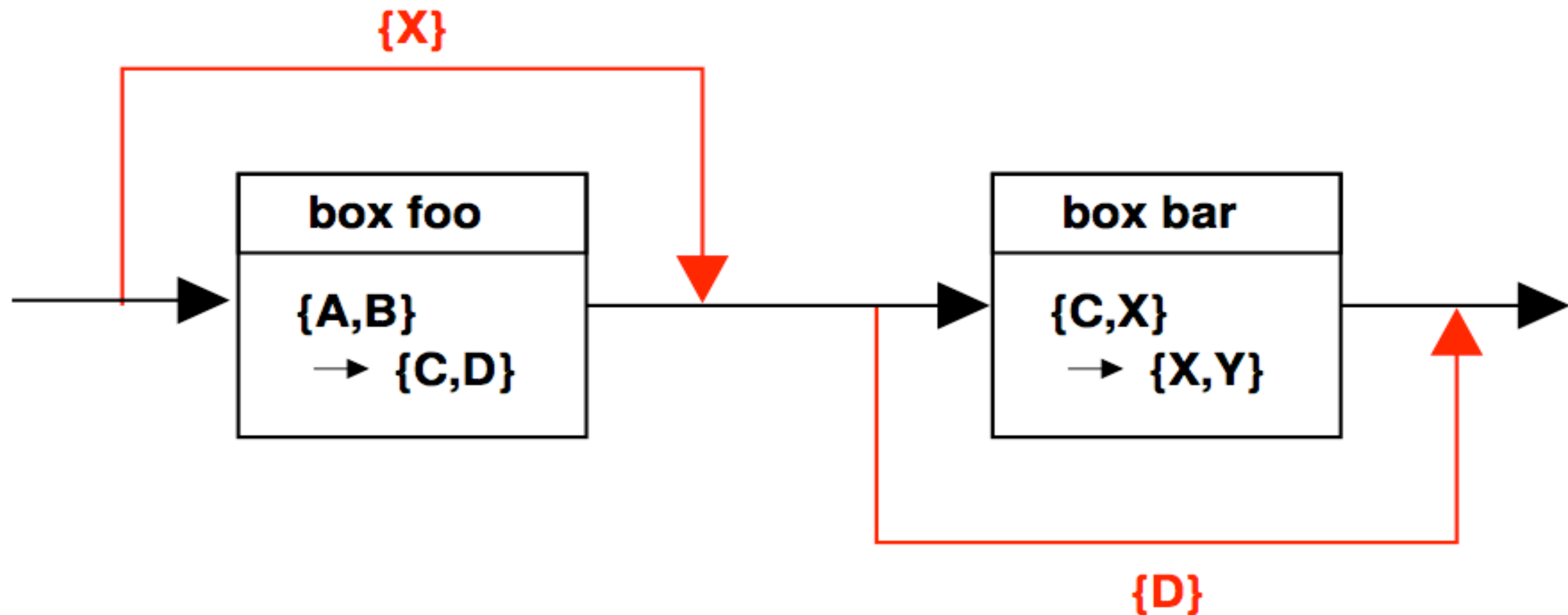
- Any "unwanted" field is bypassed to output channel
- and is attached to any record produced in response.

Solution: Flow Inheritence



- Any "unwanted" field is bypassed to output channel
- and is attached to any record produced in response.

Flow Inheritence: Enabling Network Combination



- Flow inheritence is the key to network glue.

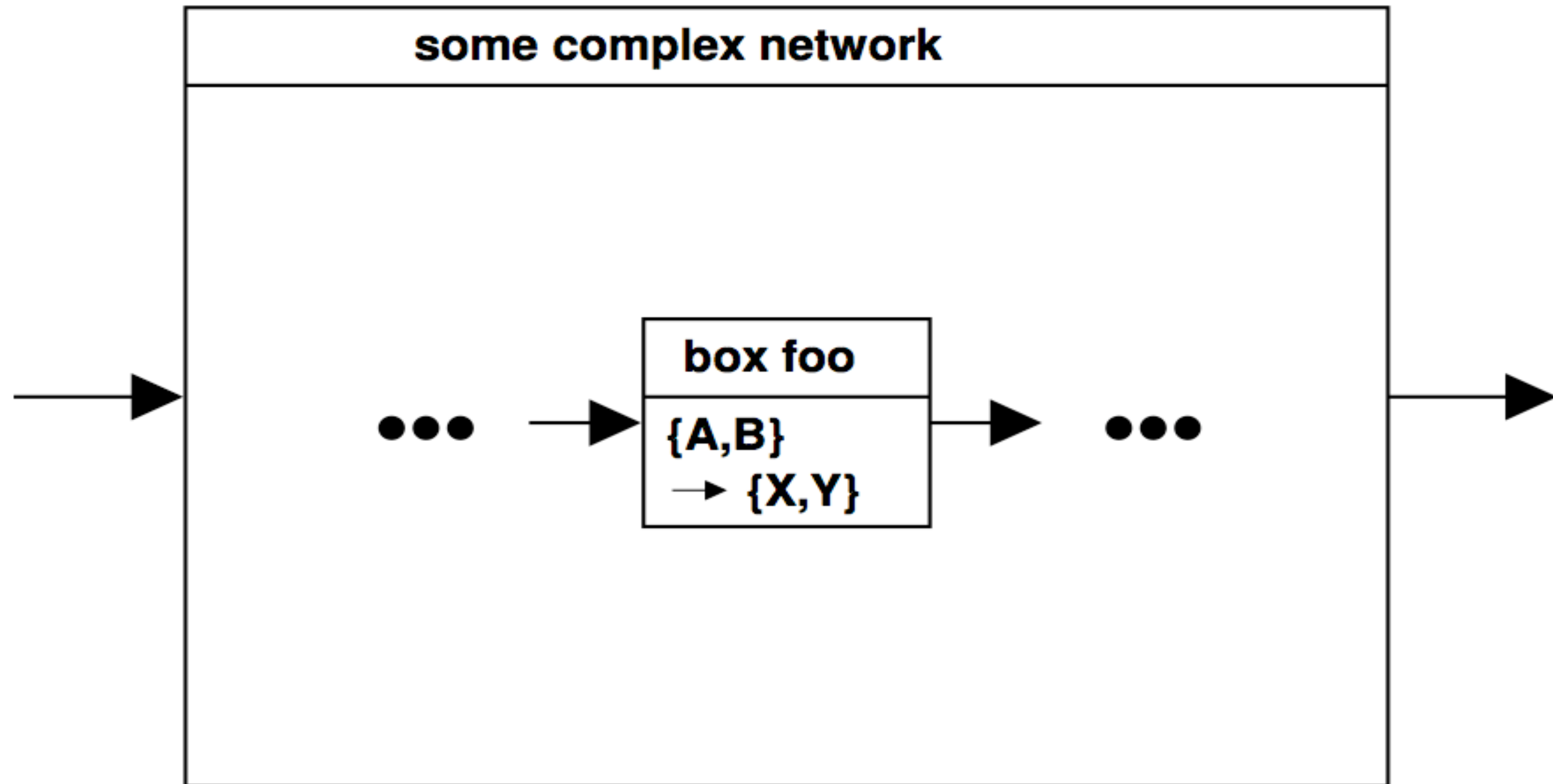
Box/net subtyping

- Further improvements to composability/flexibility
- The idea: replace a component by a more general one
  - Input type must be subtype of the input stream:
    - can read more variants
    - can read fewer fields
    - can produce fewer variants
    - can produce more fields.
  - Promotes incremental top-down design/implementation
    - first use generic components, test the whole solution, then
    - raise the box/net type until reach the type expected by the hole
  - Remember the input variants are **not necessarily** disjoint
    - Some further restrictions to ensure *type monotonicity*
    - The effect is to guarantee whole-net type predictability irrespective of component subtyping
- **does not impact concurrent behaviour directly!**
  - may affect synchro-cells by not outputting (enough of) a specific variants
  - easy to debug: insert monitoring boxes!
    - “snooping” components with {} -> {} type

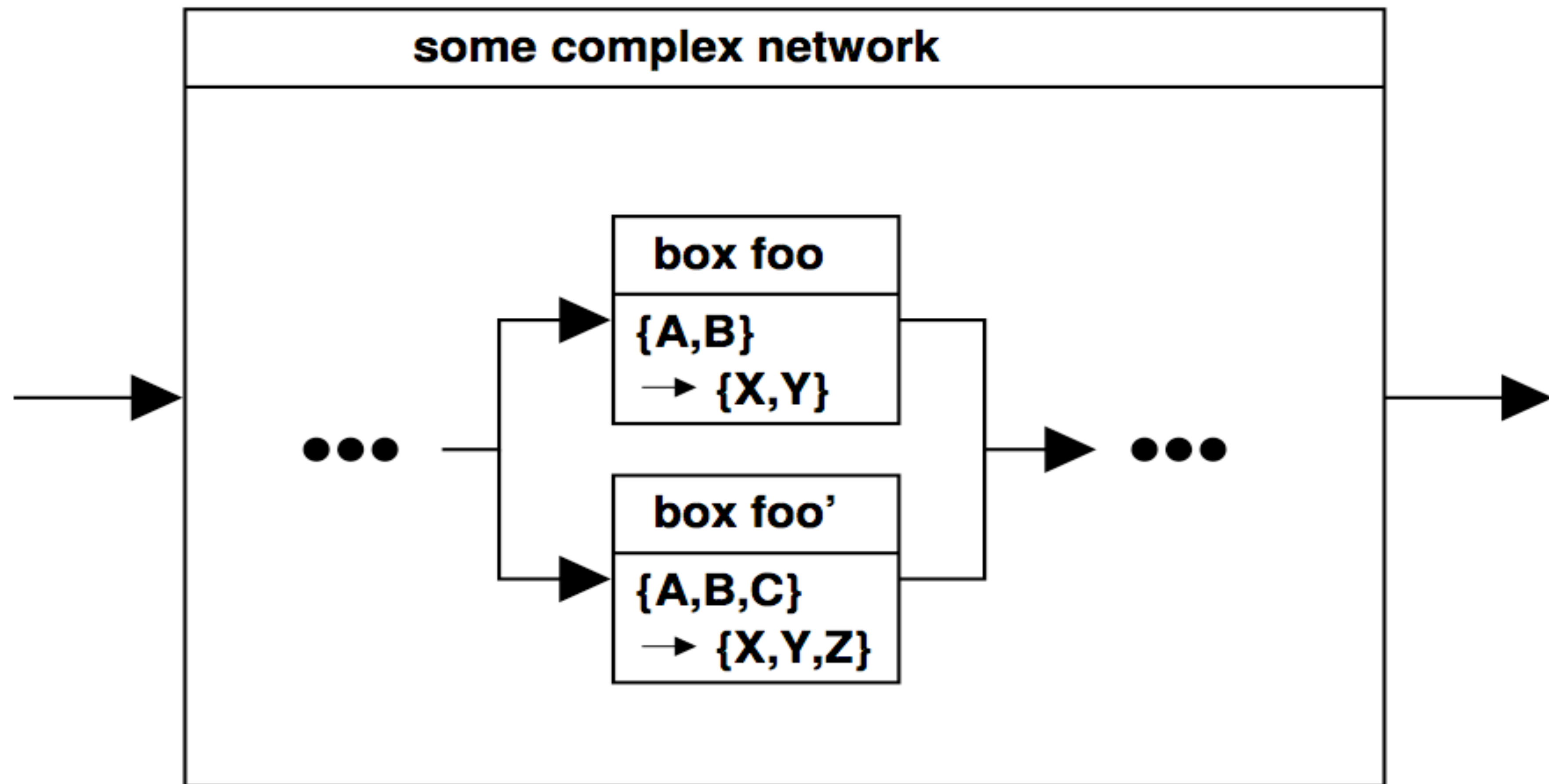
Flow Inheritance is safe

- Consider a box {a} -> {b}
- Give it {a,b}
- Since b is flow inherited, the output ~~{b,b}~~ {b}
- Which value for b?
- Rule: output takes precedence over inheritance
  - Consequence: F.I. safe, no need to know what other fields are supplied, will override.

Flow Inheritence: Enabling Adaptation



Flow Inheritence: Enabling Adaptation



- New fields are transparently transported to where they matter.

Examples

1. Asynchronous implementation of factorial
2. Particles in cells: plasma simulation

Factorial in Standard ML

Design principle:

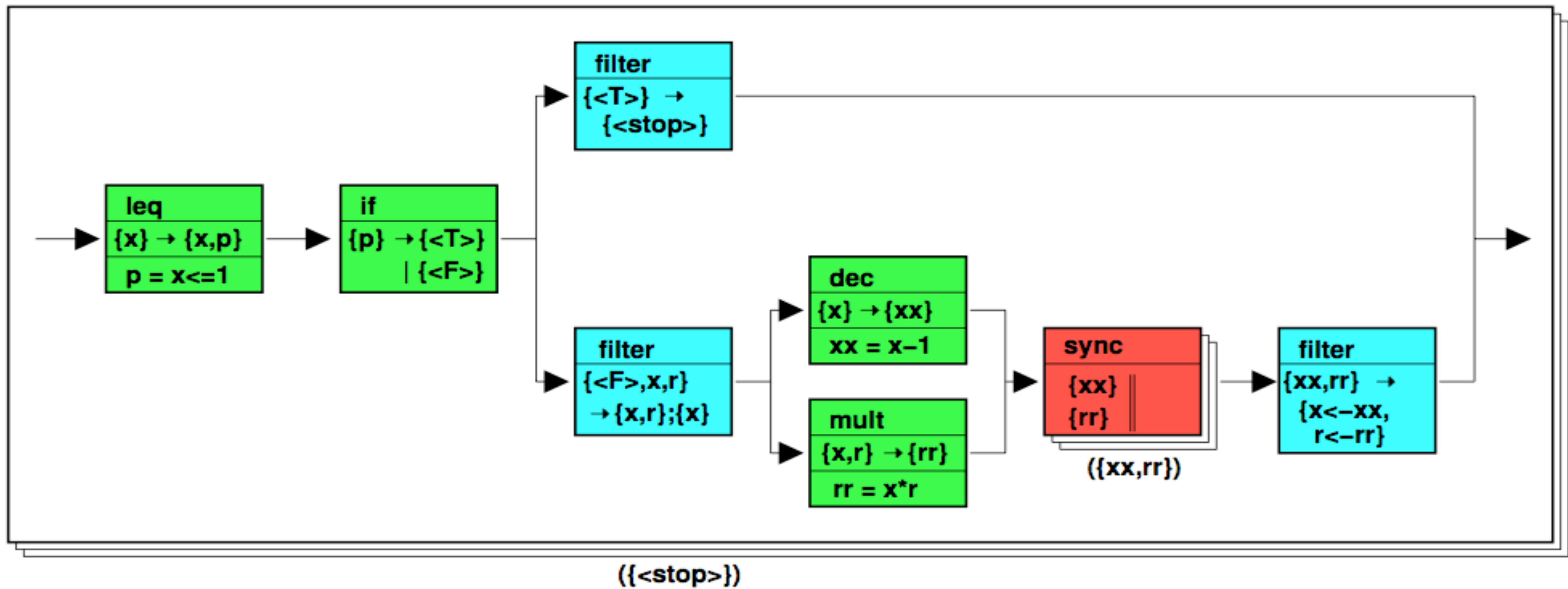
- Break down computation into atomic parts

```
fun fac n =
  let fun facit (x,y) = let val p = x<=1
                        in if p
                           then y
                           else let val a = x-1
                                    val b = x*y
                                in facit (a, b)
                                end
      val m = facit (n,1)
  in (n,m)
  end
```

Factorial in S-Net

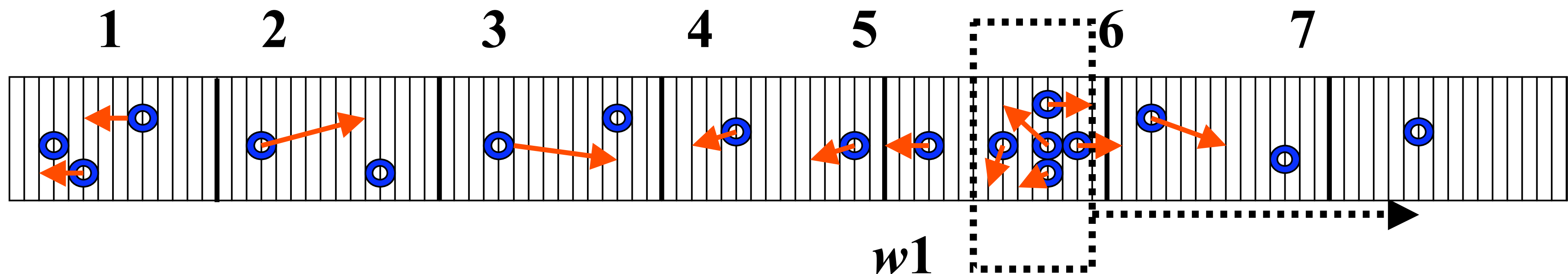
```
net facit {
  box leq ({x} -> {x,p});
  box if ({p} -> {<T>} | {<F>});
  box dec ({x} -> {a});
  box mult ({x,r} -> {b});
}
connect (leq..if..([{<T>}->{<stop>}]
                   || [{F,x,r}->{x,r};{x}]
                       .. (dec||mult)
                        .. ([|{a},{b}|]**({a,b}))
                         .. [{a,b}->{x<-a,r<-b}])) ** ({<stop>});
```

Factorial in S-Net: Graphical Representation



Particles in Cells (PIC)

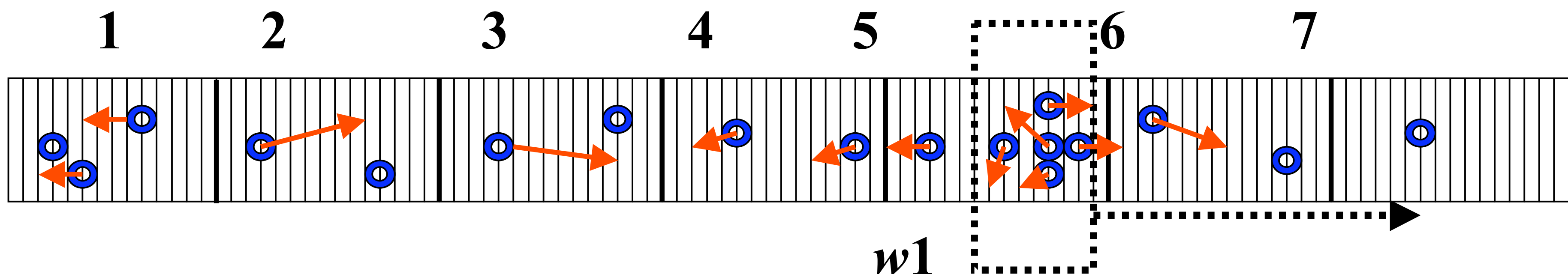
- Simulation of plasma particles interacting with each other via electromagnetic field
- Consider 1d for simplicity, and 1 sort of particles



- Field split evenly, perfectly balanced, particles imbalanced.
- “Windows” are introduced representing work to be delegated to other processors.

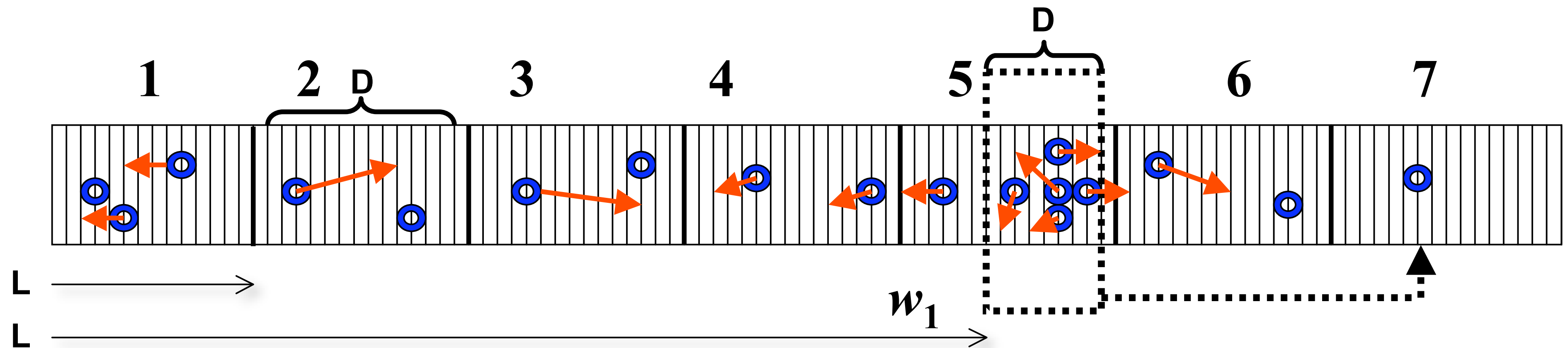
Basic concepts

- Particles are charged, each carrying a unit of charge.
- The field-grid nodes are assigned the charge of the particles near them, by interpolation.
- The field solver computes the field values due to the charges assigned to the nodes
- The particle pusher, applies EM forces to the particles due to the field values interpolated from the neighbouring nodes. The particles move accordingly.



Basic data structures

- The home record of a cell: particles pushed by the home base
  <p><A> ***Phi***, LD, nw, ***x, v***
- The window record: particles pushed by a deputy
  <p><A> LD, ***x, v***, <return><id>
- Processor tag <p>, stage <A>

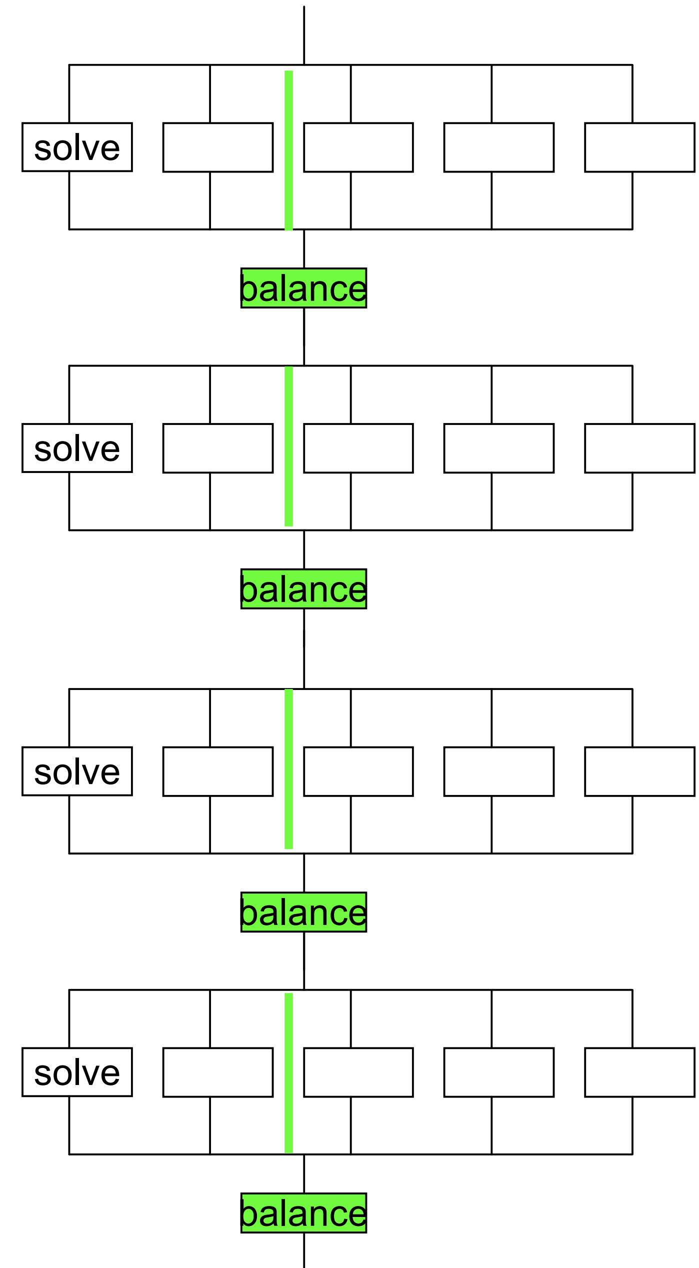


The net

<p><A> ***Phi***, LD, nw, ***x***, ***v***

<p><A> LD, ***x***, ***v***, <return><id>

```
net solution {
  net solve
  connect stageA || stageB || ...
}
connect
(([{} →{}] || solve!!<p>).. balance) * <out>
```



Stages A,B,C

<p>,<A>,***Phi***, LD, nw, ***x, v***
<p>,<A>,LD, ***x, v***, <return>,<id>

```
net solution {
  net solve
  connect stageA || stageB || stageC || .. ;
}
connect   (solve!!<p>.. balance) * <out>
```

stageA:	{<p>,<A>, Phi, LD, nw x}	→	{{<p>,<B>,LD,nw,rho}	comments
			{<p>,<C>,Phi,LD,nw,x}	as is
			{<p>,<C>,Right}	p = p –1
			{<p>,<C>,Left}}	p = p +1
	{<p>,<A>,LD,x,v,<return>,<id>}	→	{{<p>,<B>,LD1,rho1}	p = return
			{<p>,<id>,<F>,<return>,LD, x,v}	as is
			{<p>,<id>,<D>,<loc>,LD1}}	LD1=LD,loc=p,p=return

stageB = sync{{<p>,<B>,LD,nw,rho}{<B>,LD1,rho1}}..cagr

cagr : {<B>,LD,LD1,rho,rho1,nw} → {{<B>,LD,rho,nw}{<C>,rho}}

stageC = sync{{<p>,<C>,rho}{<C>,Phi,LD,nw,x,v}{<C>,Right}{<C>,Left}}..solve..[{<D>} →{<D>};{<F>}]

solve : {<C>,rho,Phi,LD,Right,Left} → {{<D>,Phi,LD}}

Stages D,E,F

<p><F> ***Phi***, LD, nw, ***x***, ***v***

<p><F> ***Phi***, LD, ***x***, ***v***, <return><id>

net solution {
  net solve
  connect stageA || stageB || stageC || .. ;
}
connect (solve!!<p>.. balance) * <out>

...

{<p>,<A>,LD, x, v,<return>,<id>} → {{<p>,<B>,LD1,rho1} p = return
{<p>,<id>,<E>,<return>,LD,x,v} as is
{<p>,<id>,<D>,<loc>,LD1}} LD1=LD, loc=p, p=return

...

stageC = sync{{<p>,<C>,rho}{<C>,Phi,LD,nw,x,v}{<C>,Right}{<C>,Left}}..solve..[{<D>} –>{<D>};{<F>}]
solve : {<C>,rho,Phi,LD,Right,Left} → {{<D>,Phi,LD}}
stageD = synch{{<p>,<D>,Phi,LD,nw}{<id>,<D>,<loc>,LD1}}..fieldfetch
fieldfetch : {<D>,<loc>,<id>,Phi,LD,LD1,nw} → {{<D>,Phi,LD,nw}{<E>,<p>,<id>,Phi}} nw cnts to 0, p=loc
stageE = synch{{<p>,<E>,<return>,LD, x, v}{<E>,<p>,Phi}} !! <id>..[{<E>} –>{<F>}]
stageF : {<F>,Phi,LD,x,v} → {<G>,Phi,LD,x,v}

Stage A in detail: A1

			comments
stageA: {<p>,<A>,Phi,LD,nw,x}	→	{{<p>,<B>,LD,nw,rho}	
		{<p>,<C>,Phi,LD,nw,x}	as is
		{<p>,<C>,Right}	p = p –1
		{<p>,<C>,Left}}	p = p +1
{<p><A> LD, x, v <return><id>}	→	{{<p>,<B>,LD1,rho1}	p = return
		{<p>,<id>,<F>,<return>,LD, x, v}	as is
		{<p>,<id>,<D>,<loc>,LD1}}	LD1=LD, loc=p,p=return

stageA = stageA1 || stageA2

stageA1 = [{<A>nw} → {<B>nw};{<C>nw};{<C>,<s>}]..

(

[{<B>} → {<B>}]..interpolate ||

[{<C><s>} → {<C>}]..getEndPoints ||

[{} → {}]

)

interpolate : {x, LD} → {{rho, LD}}

getEndPoints : {<p>,Phi, LD} → {{<p>,Right}{<p>,Left}}

Stage A in detail: A2

			comments
stageA: {<p>,<A>,Phi, LD,nw,x}	→	{{<p>,<B>LD,nw,rho}	
		{<p>,<C>,Phi,LD,nw,x}	as is
		{<p>,<C>,Right}	p = p –1
		{<p>,<C>,Left}}	p = p +1
{<p><A> LD, x, v <return><id>}	→	{{<p>,<B>,LD1,rho1}	p = return
		{<p>,<id>,<F>,<return>,LD,x,v}	as is
		{<p>,<id>,<D>,<loc>,LD1}}	LD1=LD, loc=p,p=return

stageA2 = [{<A>,<return>} → {<B>,<return>};{<F>,<return>};{<D>}]..

(

[{<B>,<return>} → {<B><p=return>}]..interpolate ||

[{<D>,<return>,<p>} → {<D>,<loc=p>,<p=return}] ||

[ ]

)

interpolate : {x, LD} → {{rho, LD}}

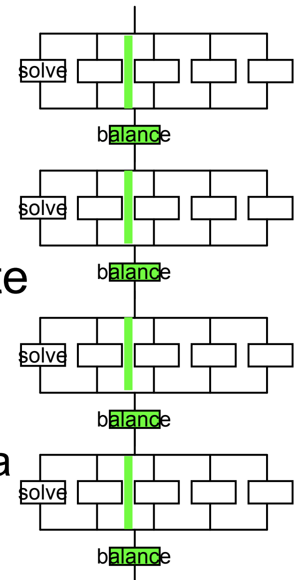
Further stages

```
net solution {
  net solve
  connect stageA || stageB ||...
}
connect   (solve!!<p>.. balance) * <out>
```

<p>,<G>,***Phi***, LD, nw, ***x, v***

<p>,<G>,***Phi***, LD, ***x, v***,<return>,<id>

- Clipper takes {x,LD} and finds x's that fall outside the domain of the cell, sends them to neighbouring cells, stage H synchronising on this
  - For windows, the runaway x's should be returned to the home cell
- Load balancer collects statistics about cells and windows, uses a separate datatype for its own state and sends records via the replicator *
  - Can kill windows, by simply setting D=0, then all x's outside the window
  - Can create windows by increasing nw and splitting off part of the x array into a new window record
  - Can move windows from processor to processor transparently



Stream processing - track record

Early days:

- Kahn, G.: The semantics of a simple language for parallel programming. In Rosenfeld, L., ed.: Information Processing 74, Proc. IFIP Congress 74. August 5-10, Stockholm, Sweden, North-Holland (1974) 471-475

  Infinite-capacity, deterministic process network. Properties useful for parallel processing.
- Ashcroft, E.A., Wadge, W.W.: Lucid, a nonprocedural language with iteration. Communications of the ACM 20 (1977) 519-526

  First language to introduce the basic idea of a block that transforms input sequences into output ones. Variables represent sequences. Ops apply stream-wise. Temporal ops.

The 90s

- Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data- flow programming language LUSTRE. Proceedings of the IEEE 79 (1991) 1305-1320

- Berry, G., Gonthier., G.: The Esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming 19 (1992) 87-152

! introduced explicit recurrence relations over streams and further developed the concept of synchronous networks

! Still used today:

- Binder, J.: Safety-critical software for aerospace systems. Aerospace America (2004) 26-27

Ramifications

- Caspi, P., Pouzet, M.: Synchronous kahn networks. In Wexelblat, R.L., ed.: ICFP ’96: Proceedings of the first ACM SIGPLAN International Conference on Functional Programming. (1996) 226-238
  synchronous view, elimination of buffering; pragmatics
- Caspi, P., Pouzet, M.: A co-iterative characterization of synchronous stream functions. In Bart Jacobs, Larry Moss, H.R., Rutten, J., eds.: CMCS ’98, First Workshop on Coalgebraic Methods in Computer Science Lisbon, Portugal, 28 - 29 March 1998. (1998) 1-21

  semantics of the synchronous case
- Michael I. Gordon et al: A stream compiler for communication-exposed architectures. In: Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA. October 2002. (2002)

  asynchronous static-rate OOP approach to streaming

And more...

- Stephens, R.: A survey of stream processing. Acta Informatica 34 (1997) 491-541

  the only survey available to date

- Babcock, B., et al.: Models and issues in data stream systems. In: Proc. of the 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems (PODS 2002), Wisconsin, May 2002. (2002) 1-16

  applications in databases (not just signal proc. graphics, multimedia, etc)

Conclusions

- SNet suggests a top down design style
- Records contain the state of computation, float between boxes
- Topology induced by tagging and type match
- Define signatures and insert synchronisers first
- Then refine signatures down to further networks
- Finally the lowest level boxes should be stateless and generic

State of the project

- EU funded (part of a €4M effort in Framework VI)
- SNet Definition available as an internal report
  - v1 deadline 31 January
  - Drafts available from project Web site
- Type theory
  - Inference algorithms developed, proven computationally feasible
  - Further extensions allow box genericity w.r.t. field types
- Semantics
  - In progress. Basic semantics available in the report.
- Compiler
  - Being produced by VTT
- Run-time library
  - v1 ready from CTCA
- Targets:
  - Unix threads; C.Jesshope's microthreaded architecture
  - Future targets: down to gates and up to Grids