ICLP'01 tutorial

A Close Look at Constraint-Based Concurrency

Kazunori Ueda

Waseda University

Tokyo, Japan

Talk Outline

- Constraint-based concurrency (CBC)
  - Essence of constraint-based communication
  - Relation to name-based concurrency
- Type systems and analyses for CBC
  - modes (directional types) and linear types
- Strict linearity and its implications
- Capabilities: types for strict linearity with sharing

Papers

- **Resource-Passing Concurrent Programming.** In *Proc. Fourth Int. Symp. on Theoretical Aspects of Computer Software*, LNCS 2215, Springer, 2001, pp. 95-126.
- **Concurrent Logic/Constraint Programming: The Next 10 Years.** In *The Logic Programming Paradigm: A 25-Year Perspective*, Apt, K.R. *et al*. (eds.), Springer, 1999, pp.53-71.
- For other papers see bibliography.

Talk Outline

- Constraint-based concurrency (CBC)
  - Essence of constraint-based communication
  - Relation to name-based concurrency
- Type systems and analyses for CBC
  - modes (directional types) and linear types
- Strict linearity and its implications
- Capabilities: types for strict linearity with sharing

Constraint-Based Concurrency

- Concurrency formalism & language based on
  - *single-assignment* (write-once) channels and
  - constructors

  cf. name-based concurrency
- Also known as
  - concurrent logic programming
  - concurrent constraint programming (CCP)
- Born and used as languages (early 1980's); then recognized and studied as formalisms

Name-Based Concurrency

- Syntax of the (asynchronous) π-calculus

$P ::=$	$\bar{x}y.P$	(output – send y along x)
$\mid$	$x(y).P$	(input – receive y from x)
$\mid$	$\mathbf{0}$	(inaction)
$\mid$	$P \mid P$	(parallel composition)
$\mid$	$(y)P$	(hiding)
$\mid$	$[x=y]P$	(match)
$\mid$	$!P$	(replication)

- Structural congruence
  - $!P \equiv P \mid !P$
  - $[x=x]\,P \equiv P$
  - $(x)(P \mid Q) \equiv P \mid (x)Q$ if x is not free in P

Name-Based Concurrency

- Reduction semantics of the π-calculus

$$\overline{x(y).P \mid \bar{x}z.Q \rightarrow P\{z/y\} \mid Q}$$

$$\frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}$$

$$\frac{P \rightarrow P'}{(y)P \rightarrow (y)P'}$$

$$\frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

Single-Assignment Channels

- Also known as *logical variables*
- Can be written at most once
  - by *tell*ing a constraint (= partial information) on the value of the channel (*unification*)
    - e.g., tell S=[read(X)|S']
- Reading is non-destructive
  - by *ask*ing if a certain constraint is entailed (*term matching*)
    - e.g., ask ∃A∃S'(S=[A|S'])
  - covers both *input* and *match* in the π-calculus

Single-Assignment Channels

- The set of all published constraints (*tell*s) forms a *constraint store*.
- Since reading is non-destructive, constraint store is monotonic.
  - Still, it's amenable to garbage collection because of its highly local nature.
- The use of constraints for message passing doesn't necessarily involve consistency techniques.

Constraint-Based Communication

- Asynchronous
  - *tell* is an independent process (as in the asynchronous π-calculus)
- Polyadic (“many-place”)
  - constructors provide built-in structuring and encoding mechanisms
  - essential in the single-assignment setting
- Mobile
- Non-strict

Constraint-Based Communication

- Asynchronous
- Polyadic
- Mobile – channel mobility in the sense of the π-calculus
  - Channels
    - can be passed using another channel
    - can be fused with another channel
    - are first-class (processes aren't)
  - available since 1983 (Concurrent Prolog)
- Non-strict

Constraint-Based Communication

- Asynchronous
- Polyadic
- Mobile
- Non-strict
  - "Constraint-based" means computing with partial information
  - Yielded many programming idioms, including
    - (streams of)* streams
    - difference lists
    - messages with reply boxes

The Language (traditional LP syntax)

(program)	$P ::=$ set of R's
(program clause)	$R ::= A$:- | B
(body)	$B ::=$ multiset of G's
(goal)	$G ::= T_1 = T_2 \mid A$
(non-unif. atom)	$A ::= p(T_1, \ldots, T_n)$, $p \neq$ '='
(term)	$T ::=$ (as in first-order logic)
(goal clause)	$Q ::=$:- B

The Language (alternative syntax)

(program)	P ::= set of R's
(program clause)	R ::= $!\forall(A \,.\, B)$
(body)	B ::= multiset of G's
(goal)	G ::= $T_1 = T_2 \mid A$
(non-unif. atom)	A ::= $p(T_1, \ldots, T_n)$, $p \neq$ '='
(term)	T ::= (as in first-order logic)
(goal clause)	Q ::= B, P

The Language

(program)	P ::= set of R's
(program clause)	R ::= $!\forall(A \,.\, B)$
(body)	B ::= multiset of G's
(goal)	G ::= $T_1 = T_2$ | A
(non-unif. atom)	A ::= $p(T_1, \ldots, T_n)$
(term)	T ::= (as in first-order logic)
(goal clause)	Q ::= B, P

tell

rewrite rule with *ask*, choice, reduction & hiding

parallel composition

Reduction Semantics

- Concurrency

$$\frac{\langle B_1, C, P\rangle \rightarrow \langle B_1', C', P\rangle}{\langle B_1 \cup B_2, C, P\rangle \rightarrow \langle B_1' \cup B_2, C', P\rangle}$$

- Tell

$$\frac{}{\langle \{t_1 = t_2\}, C, P\rangle \rightarrow \langle \phi, C \cup \{t_1 = t_2\}, P\rangle}$$

Reduction Semantics

- Concurrency

$$\frac{\langle B_1, C, P\rangle \to \langle B_1', C', P\rangle}{\langle B_1 \cup B_2, C, P\rangle \to \langle B_1' \cup B_2, C', P\rangle}$$

- Tell

send t_2 through t_1 / fuse t_1 with t_2

defines an mgu unless collapsed

$$\frac{}{\langle \{t_1 = t_2\}, C, P\rangle \to \langle \phi, C \cup \{t_1 = t_2\}, P\rangle}$$

unguarded constraint is made observable

Reduction Semantics (cont'd)

- Ask

$$\frac{}{\langle \{b\}, C, P \cup \{h :- \mid B\} \rangle \rightarrow \langle B, C \cup \{b = h\}, P \cup \{h :- \mid B\} \rangle}$$

$$\left(\begin{array}{l} \text{if } E \models \forall (C \Rightarrow \exists vars(h)(b = h)) \\ \text{and } vars(h, B) \cap vars(b, C) = \phi \end{array} \right)$$

Reduction Semantics (cont'd)

- Ask

ask done and constraints were received by *h*'s args

$$\frac{}{\langle \{b\}, C, P \cup \{h :- \mid B\} \rangle \rightarrow \langle B, C \cup \{b = h\}, P \cup \{h :- \mid B\} \rangle} \quad \left(\begin{array}{l} \text{if } E \models \forall (C \Rightarrow \exists vars(h)(b = h)) \\ \text{and } vars(h, B) \cap vars(b, C) = \phi \end{array} \right)$$

syntactic equality theory over finite terms (can be generalized)

h matches *b* under *C*

Relation to Name-Based Concurrency

- Predicates (names of recursive procedures) can be regarded as global names of conventional (destructive) channels.
  - the only source of arbitration in CBC
- Variables are local names of write-once channels.
- Constructors are global, non-channel names for composing messages with reply boxes, streams, and other data structures.

Channels in CBC and NBC

- Write-once channels allow buffering with the aid of stream constructors
  - e.g., S=[read(X)|S'] (S': continuation)
- Channels in the asynchronous π-calculus are *multisets* of messages from which *input* operations remove messages
  - e.g., $a(y).Q \mid \bar{a}b \rightarrow Q\{b/y\}$
  - Being a multiset is another source of arbitration

Channels in CBC and NBC

- CBC and NBC get closer with *type systems*:
  - *mode* (= directional type) system for CBC
  - *linear* types for the π-calculus
- Both guarantees that only one process holds a write capability and use it once
  - hence they leave no sharp difference in non-destructive and destructive read,
  - except that CBC still allows multicasting and channel fusion

Communication in CBC and NBC

- In CBC,
  - *tell* subsumes two operations
    - output e.g., `X=3, X=[push(5)|X']`
    - fusion (of two channel names) e.g., `X=Y`
  - *ask* subsumes two operations
    - input (synchronization and value passing)
    - match (checking of values)
- However, match in *moded* CBC doesn't allow the checking of channel equality (cf. $L\pi$)

Channels in CBC Are Local Names

- Fallacy: constraint store is global, shared, single-assignment memory
- Channels are created as fresh local names that cannot be forged by the third party
  - the locality could be made explicit in configurations
- A new channel can be exported and imported only by using an existing channel
  - e.g., `p([create(S)|X']) :- | server(S), p(X').`

Talk Outline

- Constraint-based concurrency
  - Essence of constraint-based communication
  - Relation to name-based concurrency
- Type systems and analyses for CBC
  - modes (directional types) and linear types
- Strict linearity and its implications
- Capabilities: types for strict linearity with sharing

I/O Modes: Motivations

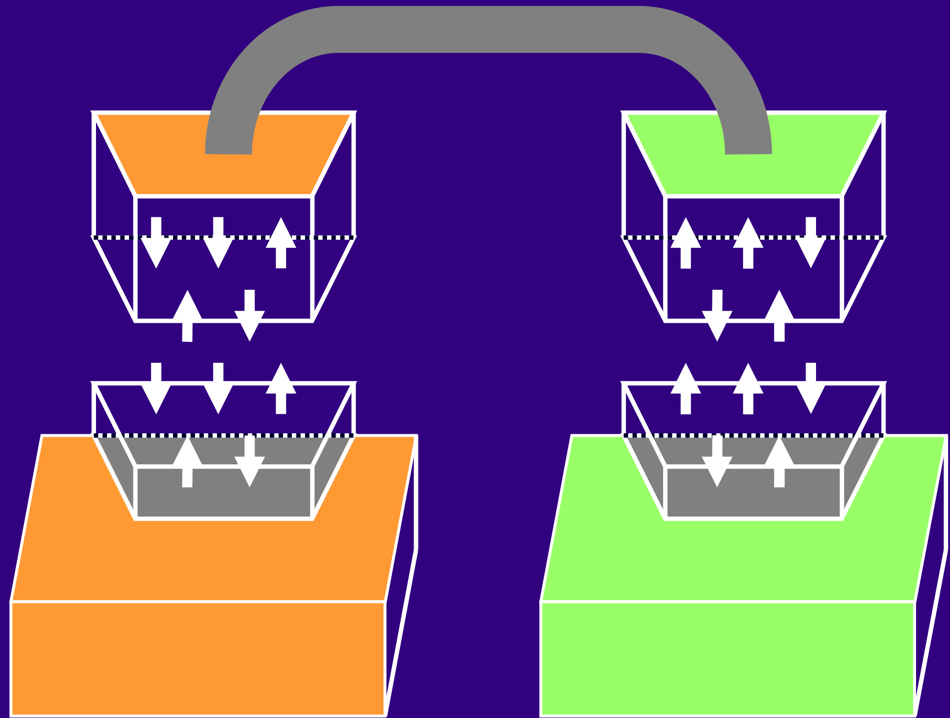
- Our experience with concurrent logic languages (Flat GHC) shows that logical variables are used mostly as *cooperative* communication channels with statically established protocols (point-to-point, multicasting)
- Non-cooperative use may cause collapse of the constraint store
  - e.g., $X = 1 \wedge X = 2 \wedge 1 \neq 2$ entails anything!

The Mode System of Moded Flat GHC

- Assigns *polarity* (+/–) *structures* to the arguments of processes so that the write capability of each part of data structures is held by exactly one process
- Unlike standard types in that modes are resource-sensitive
- Moding rules are given in terms of mode constraints (cf. inference rules)
- Can be solved (mostly) as unification over mode graphs (feature graphs with cycles)

An Electric Device Metaphor

- Signal cables may have various structures (arrays of wires and pins), but
  - the two ends of a cable, viewed from outside, should have opposite polarity structures, and
  - a plug and a socket should have opposite polarity structures when viewed from outside.



goal = device
variable = cable

Modes as Functions

- Given a "position" (of any procedure, of arbitrary depth), a mode function will answer the I/O mode of that position.

$$m : P_{Atom} \rightarrow \{in, out\}$$

- P_{Atom} : set of *paths* of the form

  $<p, i><f_1, i_1> \ldots <f_n, i_n> \quad (n \geq 0)$

  e.g.: <append, 2><., 2><., 1>
- P_{Term} : set of *paths* of the form

  $<f_1, i_1> \ldots <f_n, i_n> \quad (n \geq 0)$
- $m(p)$: mode at p
- m/p : modes at and below p ($P_{Term} \rightarrow \{in, out\}$)

Mode Constraints on a Well-Moding *m*

- Constructors occur at *input* positions
- Non-linear head variables occur at *fully input* positions (to check if they hold identical values)
- The two arguments of a unification body goal (tell) have complementary modes
- Variable occurring at $p_1, \ldots, p_k$ (head) and $p_{k+1}, \ldots, p_n$ (body) satisfies
  - $\boldsymbol{R}(\{m/p_1, \ldots, m/p_n\})$ (k=0)
  - $\boldsymbol{R}(\{\overline{m/p_1}, m/p_{k+1}, \ldots, m/p_n\})$ (k>0)

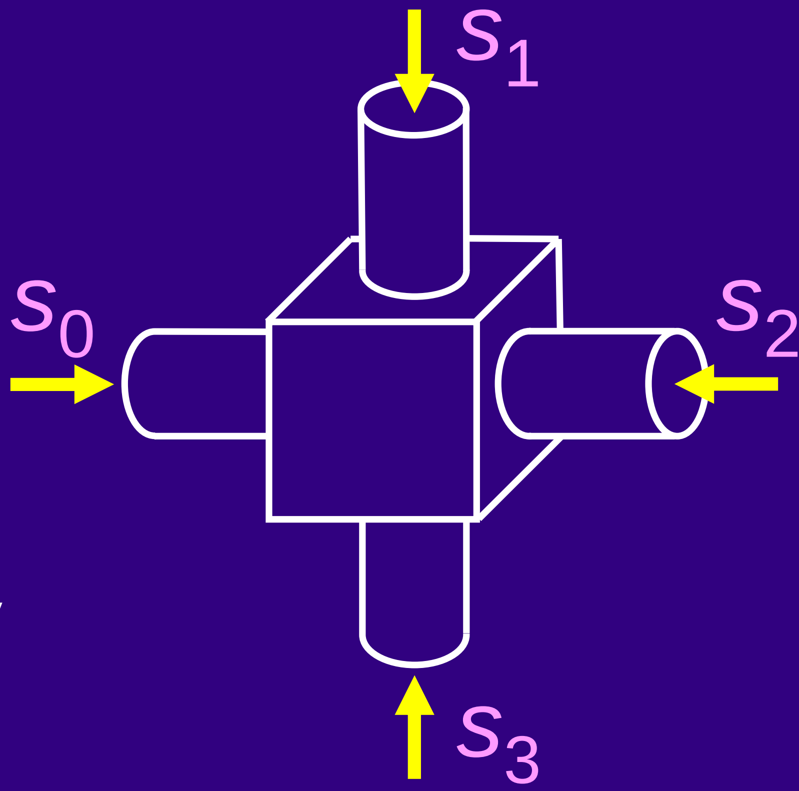
  where $\boldsymbol{R}(S) = \forall q \in P_{Term}\, \exists s \in S$

  $(s(q) = out \wedge \forall s' \in S \setminus \{s\} (s'(q) = in))$

Principles Behind the Constraints

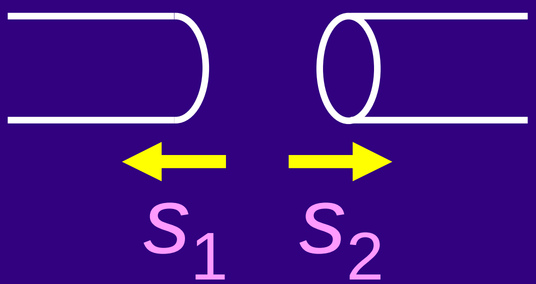
- A variable is a cable . . . . . or a hub.

$$\boldsymbol{R}(\{s_1, s_2\}) \Leftrightarrow s_1 = \overline{s_2}$$



$$\boldsymbol{R}(\{s_0, s_1, s_2, s_3\})$$

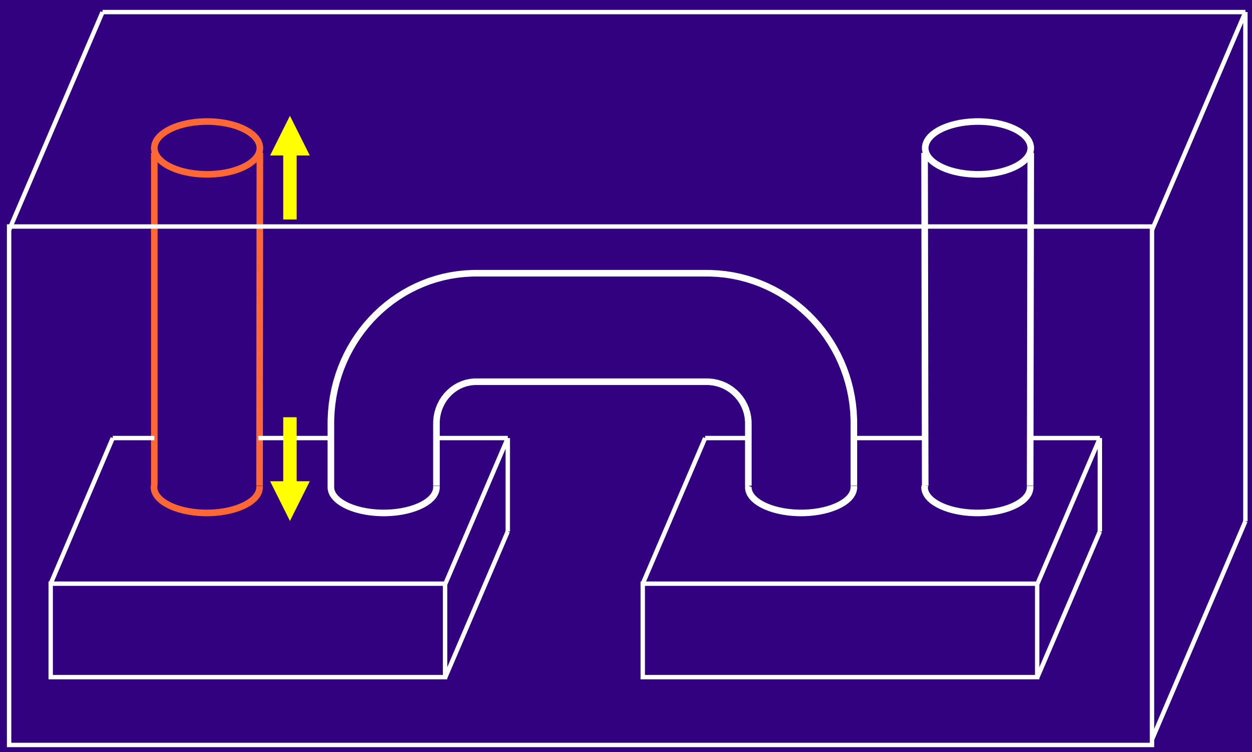
- Constraint for connectivity



$$s_1 = \overline{s_2}$$

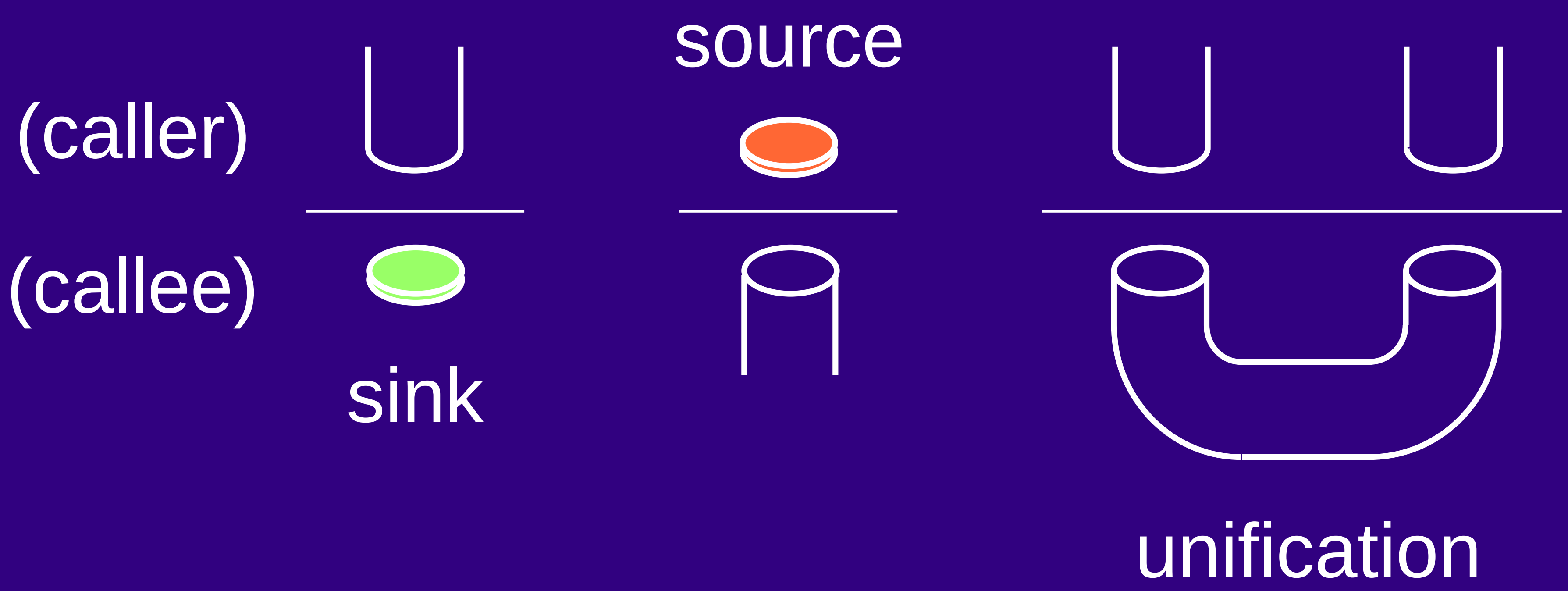
Principles Behind the Constraints

- Clause heads and body goals have opposite polarities, so do their arguments.

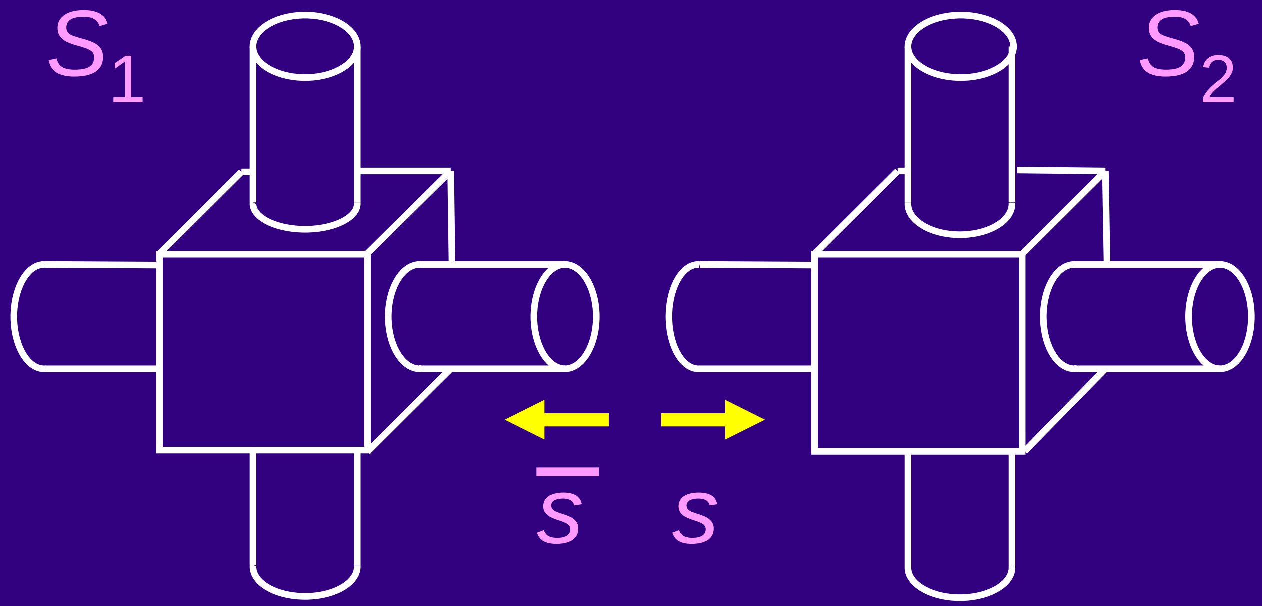


Principles Behind the Constraints

- Goal-head connection

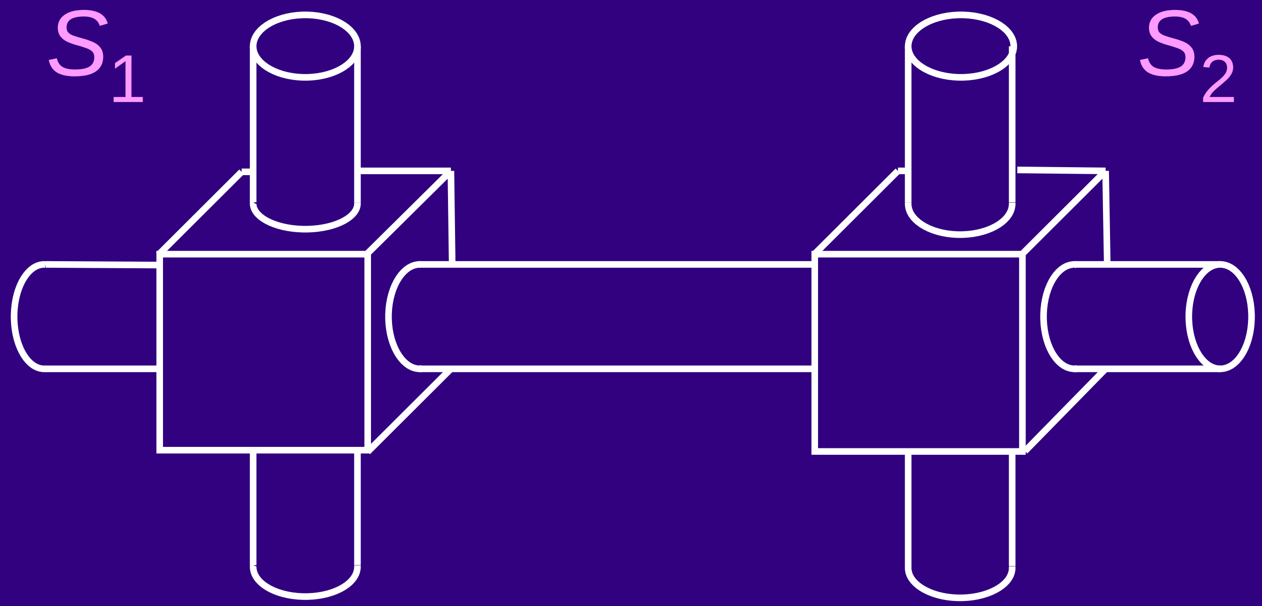


Resolution Principle



$$\boldsymbol{R}(\{\overline{s}\} \cup S_1) \wedge \boldsymbol{R}(\{s\} \cup S_2)$$

Resolution Principle



$$\boldsymbol{R}(\{\overline{s}\} \cup S_1) \ \wedge \ \boldsymbol{R}(\{s\} \cup S_2)$$

$$\Rightarrow \boldsymbol{R}(S_1 \cup S_2)$$

Moding: Implications and Experiences

- A process can pass a (variable containing) write capability to somebody else, but cannot duplicate or discard it.
- Two write capabilities cannot be compared
- Read capabilities can be copied, discarded and compared
  - cf. Linearity system
- Extremely useful for debugging – pinpointing errors and automated correction (!)
- Encourages resource-conscious programming

Moding: Implications and Experiences

- Encourages resource-conscious programming by giving weaker mode constraints to variables with exactly two occurrences
  - A singleton variable constrains the mode of its position to fully input or fully output.
  - A variable with three or more occurrences constrain the modes of more positions.
- Weaker constraints lead to more generic (= more polymorphic) programs

Theorems

- Unification degenerates to assignment to a variable.
- (Subject Reduction) A well-moding *m* is preserved by reduction
- (Groundness) When a program terminates successfully, every variable is bound to a constructor.

Linearity: An Observation (cf. LNCS 1068)

- In (concurrent) logic programs, many of the program variables have *exactly two* occurrences.
  - Example:

```
append([],    Y,Z  ) :- true | Z=Y.
append([A|X],Y,Z0) :- true |
              Z0=[A|Z], append(X,Y,Z).
```

  - Counter-example:

```
p(...X...) :- true | r(...X...), p(...X...).
```

An Observation

- Another example: quicksort

```
qsort(Xs,Ys) :- true | qsort(Xs,Ys,[]).
qsort([],Ys0,Ys) :- true | Ys=Ys0.
qsort([X|Xs],Ys0,Ys3) :- true |
    part(X,Xs,S,L), qsort(S,Ys0,Ys1),
    Ys1=[X|Ys2], qsort(L,Ys2,Ys3).
part(_,[],S,L) :- true | S=[], L=[].
part(A,[X|Xs],S0,L) :- A≥X |
    S0=[X|S], part(A,Xs,S,L).
part(A,[X|Xs],S,L0) :- A<X |
    L0=[X|L], part(A,Xs,S,L).
```

An Observation

- Another example: quicksort

```
qsort(Xs,Ys) :- true | qsort(Xs,Ys,[]).
qsort([],Ys0,Ys) :- true | Ys=Ys0.
qsort([X|Xs],Ys0,Ys3) :- true |
    part(X,Xs,S,L), qsort(S,Ys0,Ys1),
    Ys1=[X|Ys2], qsort(L,Ys2,Ys3).
part(_,[],S,L) :- true | S=[], L=[].
part(A,[X|Xs],S0,L) :- A≥X |
    S0=[X|S], part(A,Xs,S,L).
part(A,[X|Xs],S,L0) :- A<X |
    L0=[X|L], part(A,Xs,S,L).
```

Another Observation

```
qsort(Xs,Ys) :- true | qsort(Xs,Ys,[]).
qsort([],Ys0,Ys) :- true | Ys=Ys0.
qsort([X|Xs],Ys0,Ys3) :- true |
    part(X,Xs,S,L), qsort(S,Ys0,Ys1),
    Ys1=[X|Ys2], qsort(L,Ys2,Ys3).
```

- Virtually all variables with ≥3 channel occurrences (nonlinear variables) are used for simple, one-way communication
- Many variables with exactly two occurrences (linear variables) have quite complex communication protocols

Linearity Analysis

- Statically distinguishes between shared and nonshared data structures
  - *shared* : possibly referenced by two or more pointers (when assignments are done by pointer sharing)
  - *nonshared* : referenced by only one pointer
    - Nonshared structures can be recycled as soon as read by the sole reader (compile-time garbage collection), as long as writers have no access to structure elements any more

Linearity Annotations

- We annotate all constructors in the body goals of program+goal clauses (cf. 1-bit reference counting)

$f^{1}(\quad , \quad , \quad)$	or	$f^{\omega}(\quad , \quad , \quad)$
not shared		possibly shared

- Closure conditions:
  - $f^{\omega}(\ldots\ g^{1}(\ldots)\ \ldots)$ — NO
  - $f^{1}(\ldots\ g^{\omega}(\ldots)\ \ldots)$ — OK

Linearity Annotations

- Example:

```
:- p([1,2,3],X), q([1,2,3],Y).
```

  - The 14 constructors can be given “1” if the lists are created separately, and should be given “ω” if the lists are shared.
- The annotations are dynamic (as reference counters are), but are to be compiled away by static linearity analysis

Extending Operational Semantics

```
  :- … p(… X …) … X= t … q(… X …)
→ :- … p(… t …) …        … q(… t …)
```

```
  :- … p(… t …) …
        p(… X …) :- | q(… X …), r(… X …).
→ :- … q(… t …), r(… t …) …
```

- X nonlinear → change the annotations in the term *t* to "ω"
- X linear → retain the original annotations

Linearity System

- Deals with the sharing aspects of programs
- Assigns linearity (*nonshared/shared*) structures to the arguments of processes so that as many parts of data structures as possible are guaranteed to be “nonshared”
- Unlike standard types in that linearities are resource-sensitive
- Can be solved (mostly) as unification over linearity graphs (feature graphs with cycles)

Output of klint v2

```
%%% Mode %%%
:- mode main:quicksort(1,3).
:- mode main:qsort(1,3,-3).
:- mode main:part(++,1,-1,-1).
:- modedef 1 = (+,[[-2|1]]).
:- modedef 2 = (-,[]).
:- modedef 3 = (-,[[2|3]]).

%%% Linearity %%%
:- lin main:quicksort(1,2).
:- lin main:qsort(1,2,2).
:- lin main:part(**,1,1,1).
:- lindef 1 = (?,[[**|1]]).
:- lindef 2 = (?,[[**|2]]).
```

Talk Outline

- Constraint-based concurrency
  - Essence of constraint-based communication
  - Relation to name-based concurrency
- Type systems and analyses
  - modes (directional types) and linear types
- Strict linearity and its implications
- Capabilities: types for strict linearity with sharing

Linear Variables Are Dipoles (1st step)

- Insertion sort

```
sort([],     S) :- | S=[].
sort([X|L0],S) :- | sort(L0,S0), insert(X,S0,S).
insert(X,[],      R) :-        | R=[X].
insert(X,[Y|L],  R) :- X ≤ Y | R=[X,Y|L].
insert(X,[Y|L0],R) :- X > Y | R=[Y|L],
                                insert(X,L0,L).
```

- From now on we disallow monopole (singleton) variables

Polarizing Constructors (2^{nd} step)

- Insertion sort

```
sort([],      S) :- | S=[].
sort([X|L0],S) :- | sort(L0,S0), insert([X|S0],S).
insert([X],       R) :-         | R=[X].
insert([X,Y|L],  R) :- X ≤ Y | R=[X,Y|L].
insert([X,Y|L0],R) :- X > Y | R=[Y|L],
                                   insert([X|L0],L).
```

- Linear constructors are also dipoles; the two occurrences of a linear constructor are two polarized instances of the same constructor.

Strict Linearity

- A program clause is called *strictly linear* if all variables and constructors are dipoles.
  - Constructors can now be regarded as channels that convey fixed values (and more importantly, *resources*) from head to body.
- A further step towards resource-conscious programming

Polarizing Constructors (cont'd)

- Are initial constructors and variables monopoles?

```
:- sort([3,1,4,1,5,9],X).
```

- A strictly linear (and symmetric) version is:

```
main([3,1,4,1,5,9],X) :- | sort([3,1,4,1,5,9],X).
```

which will be reduced finally to

```
main([3,1,4,1,5,9],X) :- | X = [1,1,3,4,5,9].
```

Programming Under Strict Linearity

- Append

```
append([],Y,Z) :- | Z=Y.
append([A|X],Y,Z0) :- |
            Z0=[A|Z], append(X,Y,Z).
```

- Strictly linear version

```
append([],Y,Z,U) :- | Z=Y, U=[].
append([A|X],Y,Z0,U) :- |
            Z0=[A|Z], append(X,Y,Z,U).
```

- The former is a *slice* of the latter.

Linearizing Server Processes (Hard)

- Stack server

```
stack([],              D     ) :- | true.
stack([push(X)|S],D          ) :- | stack(S,[X|D]).
stack([pop(X)|S],  [Y|D])      :- | X=Y, stack(S,D).
```

- Strictly linear version (1st attempt)

```
stack([](Z),              D     ) :- | Z=[](D).
stack([push([X|*],Y)|S],D       ) :- |
        Y=[push(*,*)|*], stack(S,[X|D]).
stack([pop(X)|S],         [Y|D]) :- |
        X=[pop([Y|*])|*], stack(S,D).
```

Linearizing Server Processes (Hard)

- Stack server

```
stack([],              D     ) :- | true.
stack([push(X)|S],D          ) :- | stack(S,[X|D]).
stack([pop(X)|S],  [Y|D]) :- | X=Y, stack(S,D).
```

- Strictly linear version (2nd attempt)

```
stack([](Z),                D     ) :- | Z=[](D).
stack([push([X|*],Z)|S],D        ) :- |
        Z=[push(*,*)|*], stack(S,[X|D]).
stack([pop(X,Z)|S],          [Y|D]) :- |
        X=[Y|*], Z=[pop(*,*)|*], stack(S,D).
```

Linearizing Server Processes (Hard)

- Strictly linear version

```
stack([](Z),                D      ) :- | Z=[](D).
stack([push([X|*],Y)|S],D      ) :- |
        Y=[push(*,*)|*], stack(S,[X|D]).
stack([pop(X,Z)|S],         [Y|D]) :- |
        X=[Y|*], Z=[pop(*,*)|*], stack(S,D).
```

  - A server doesn't want to keep envelopes ([|]) or cover sheets (push/pop)
  - "*" (void) is a non-constructor-non-variable symbol with *zero capability* (no write, no read)

Polarizing Predicates (3rd step)

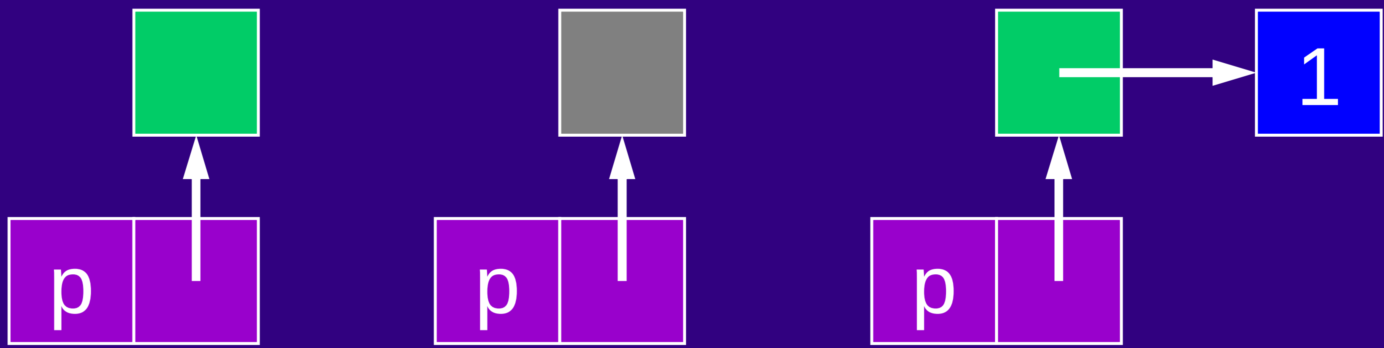
- Insertion sort

```
sort([],       S) :- | S=[], sort(*,*).
sort([X|L0],S), insert(*,*) :- |
        sort(L0,S0), insert([X|S0],S).
```

  - cf. CHR, cc(multiset)
- Goals with void arguments are free goals waiting for habitants
  - can be considered as implicitly given

Resource Aspect of Values

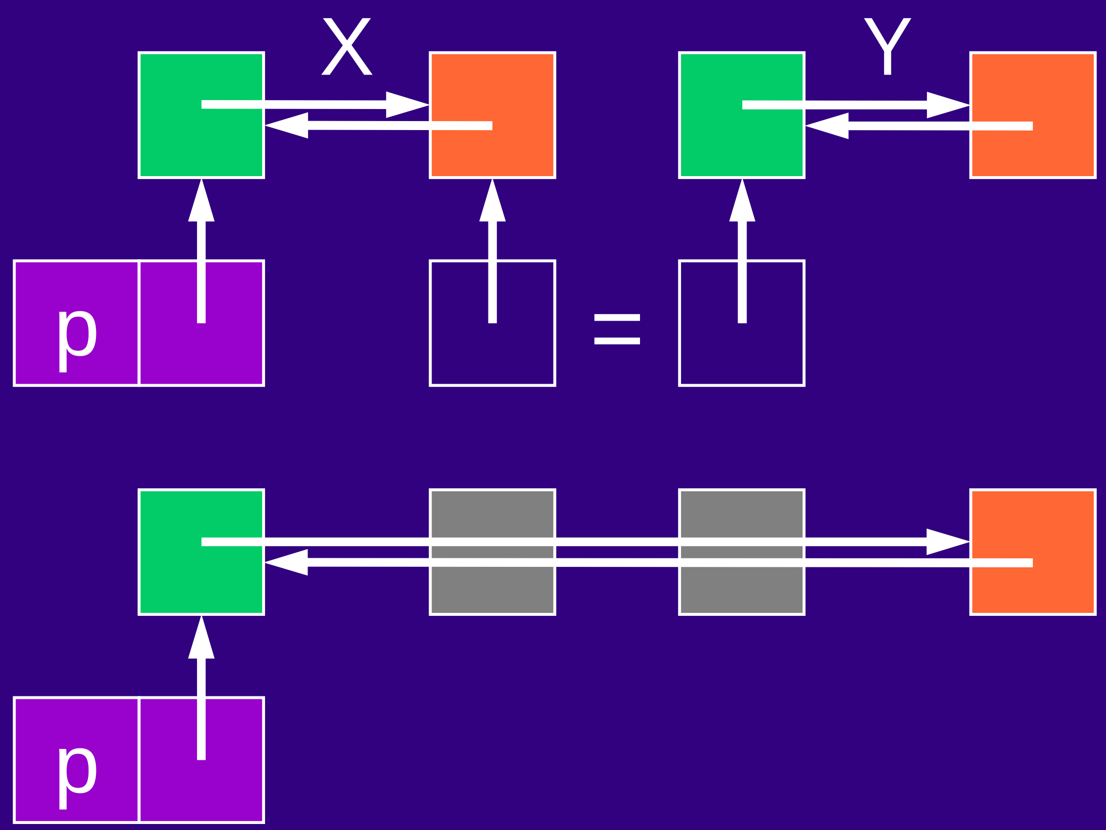
- Standard counting under the untyped setting
  - Void: 1 unit
  - Variable: 1 unit per occurrence
  - N-ary constructor and predicate: N+1 units
    - Arguments should point to variables or voids
  - e.g., p(X): 3 units, p(*): 3 units, p(1): 4 units



  - Typing can reduce dereferencing and space

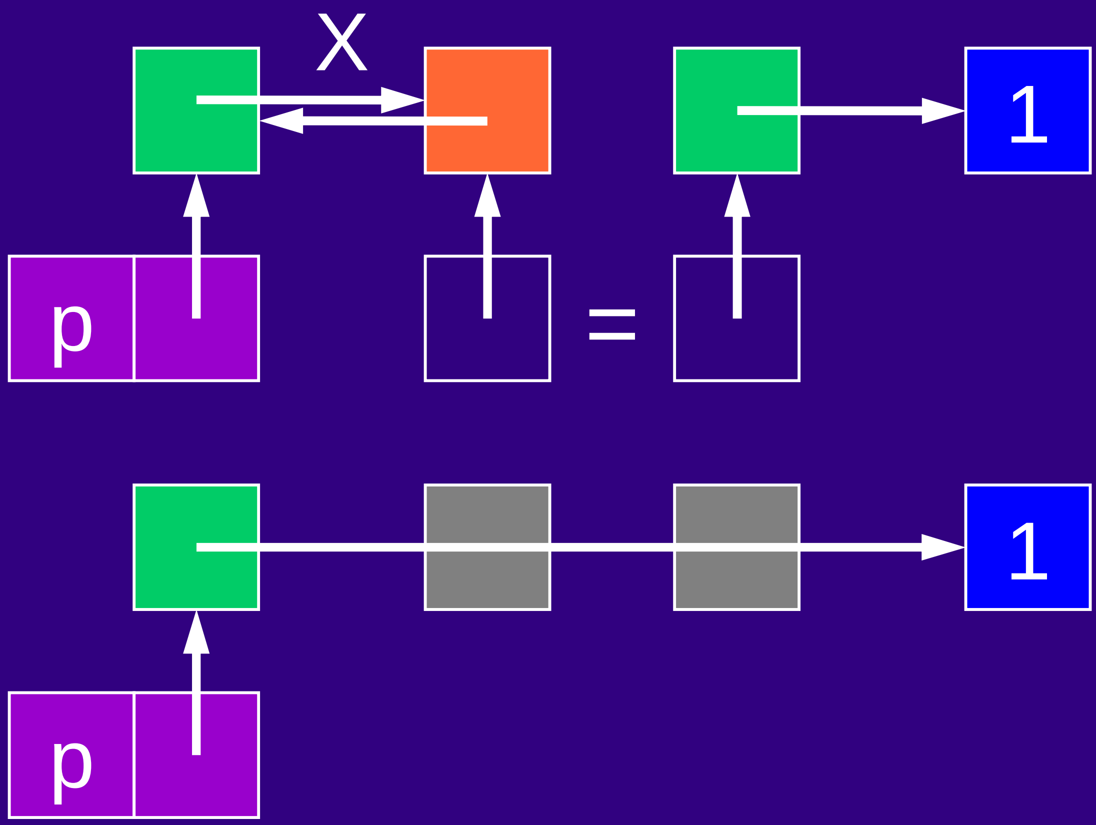
Constant-Time Property

- All entities are accessed by dereferencing exactly twice (yes, two is the magic number).



Constant-Time Property

- All entities are accessed by dereferencing exactly twice (yes, two is the magic number).



Talk Outline

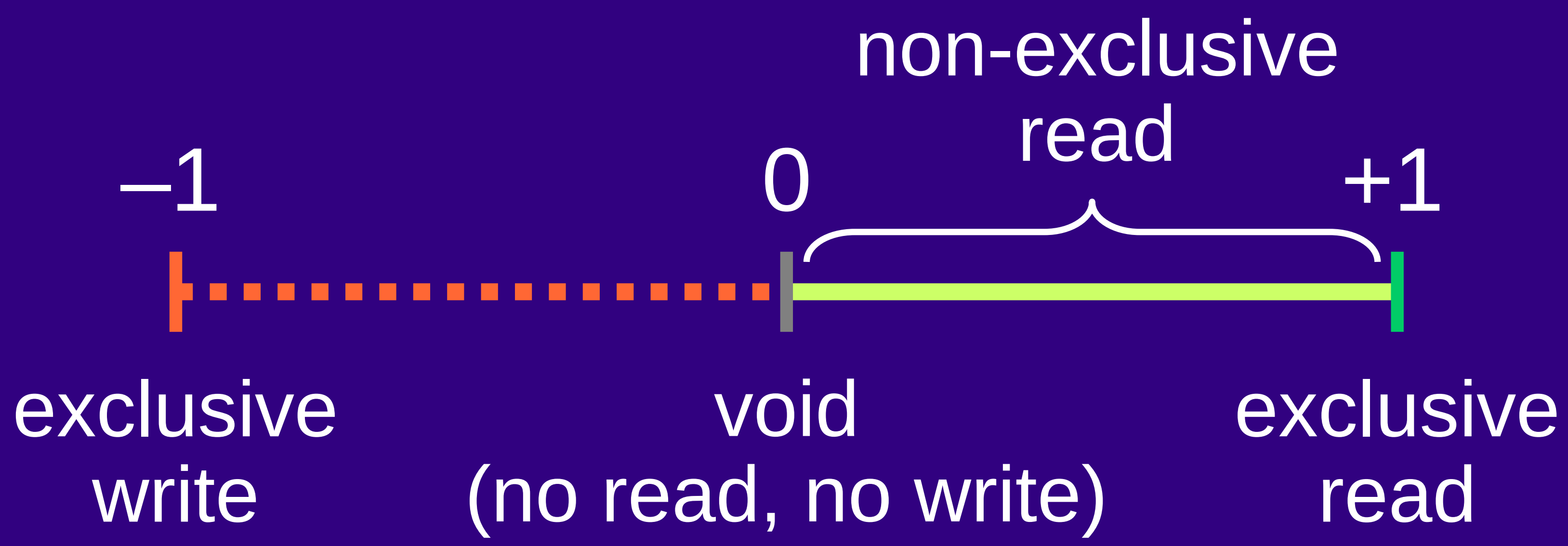
- Constraint-based concurrency
  - Essence of constraint-based communication
  - Relation to name-based concurrency
- Type systems and analyses
  - modes (directional types) and linear types
- Strict linearity and its implications
- Capabilities: types for strict linearity with sharing

Sharing under Strict Linearity

- Goals:
  1. To allow *concurrent* access to shared resource
     - e.g., large arrays used for table lookup
  2. To recover linearity after concurrent access
     - Can ω get back to 1?
- Two ways of concurrent access
  - *multiplicative* = full access to disjoint parts
    - already supported by mode+linearity
  - *additive* = read access to the whole structure

Let's Take a Reciprocal

- Mode {*in*,*out*} and linearity {*nonshared*, *shared*} can be unified and generalized in a simple setting, the [–1,+1] capability system.



- cf. Weighted reference counting

In Pursuit of Symmetry

- What's the meaning of (–1,0) capabilities?
- Example: concurrent read

```
read(X0,X) :- |
        read1(X0,X1), read2(X0,X2), join(X1,X2,X).
```

  - Suppose `read` receives `X0` with exclusive read capability **1** (**1**(p)=+1) and split it into two non-exclusive capabilities, α and $\mathbf{1}-\alpha$.
  - Then these capabilities will be returned through `X1` ($-\alpha$) and `X2` ($\alpha-\mathbf{1}$)
    - because they cannot be disposed

In Pursuit of Symmetry

- Example: concurrent read (cont'd)

```
read(X0,X) :- |
        read1(X0,X1), read2(X0,X2), join(X1,X2,X).
```

  – X1 (−α) and X2 (α−1) become logically the same as X0 (they must alias unless read*n* diverges or deadlocks)

  – Then the two aliases are joined by a clause with a nonlinear head:

```
join(A,A,B) :- | B = A.
```

  - The capabilities of the three args sum up to **0**.

Capability Annotations

- We annotate all constructors in (initial or reduced) goal clauses.
  - The annotations are to be compiled away

$f^{1}(\ \ ,\ \ ,\ \ )$ or $f^{\kappa}(\ \ ,\ \ ,\ \ )$

exclusive $(0 < \kappa < 1)$ non-exclusive

- Closure condition:
  - $f^{\kappa}(\ldots\ g^{1}(\ldots)\ \ldots)$ — NO
  - $f^{1}(\ldots\ g^{\kappa}(\ldots)\ \ldots)$ — OK

Extending Operational Semantics

```
  :- … p(… X …) … X= t … q(… X …)
→ :- … p(… t …) …        … q(… t …)
```

```
  :- … p(… t …) …
       p(… X …) :- | q(… X …), r(… X …).
→ :- … q(… t …), r(… t …) …
```

- X nonlinear → split the capabilities in the term *t* using any (e.g., random) numbers
- X linear → retain the original capabilities

Capability System

- A capability is a function

$$c : P_{Atom} \rightarrow [-1,+1]$$

- Polymorphic w.r.t. non-exclusive capabilities because they decrease by repeated splitting
  - So all goals created at runtime are distinguished using suffixes

Capability Constraints (= Typing Rules)

- For a unification goal (of the form $t_1 =_s t_2$),

  $c/\langle =_s,1\rangle + c/\langle =_s,2\rangle = \mathbf{0}$

- For a variable occurring at $p_1, \ldots, p_k$ (head) and $p_{k+1}, \ldots, p_n$ (body),

  $-c/p_1 - \ldots - c/p_k + c/p_{k+1} + \ldots + c/p_n = \mathbf{0}$

  (*Kirchhoff's Current Law*)

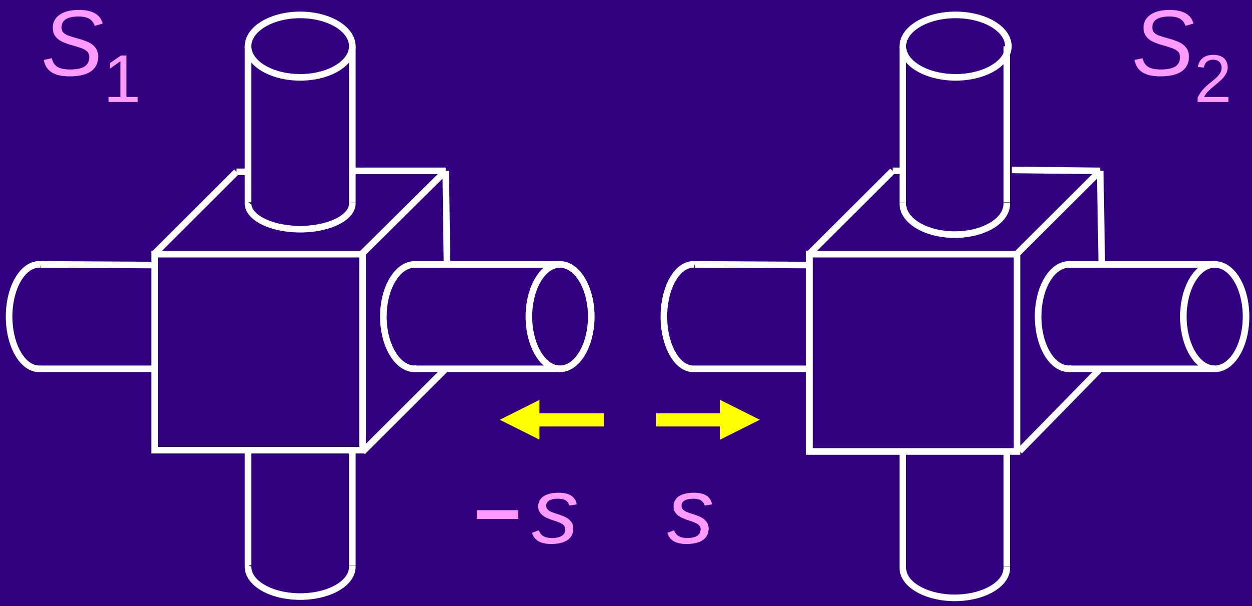
  and exactly one of $\{-c/p_1, +c/p_{k+1}, \ldots, +c/p_n\}$ is negative

- For a nonlinear head variable at p, $c/p > \mathbf{0}$

Capability Constraints (= Typing Rules)

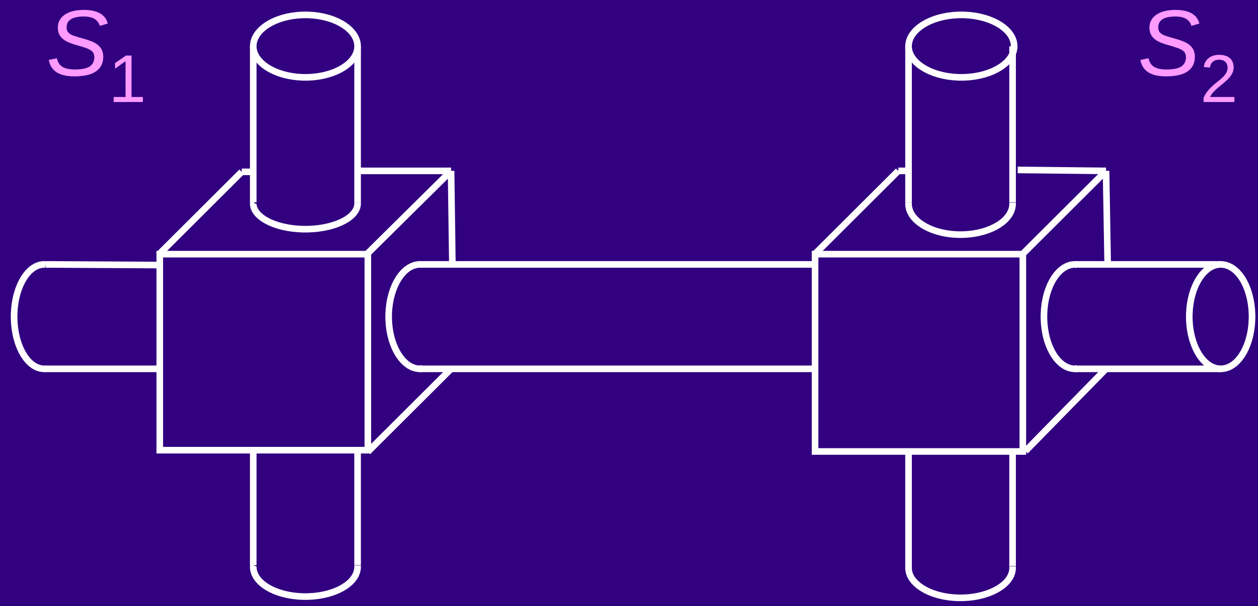
- A constructor *f* in head/body must find its partner with matching capability (> **0**) in body/head, respectively
  - If *f* is exclusive, only top-level capability match is required; the constructor name and the arguments can be changed
  - Otherwise, full match is required
- A void path has a zero capability
- A non-void path has a non-zero capability

Kirchhoff's Current Law



$$-s + \Sigma S_1 = \mathbf{0} \ \wedge \ s + \Sigma S_2 = \mathbf{0}$$

Kirchhoff's Current Law



$$-s + \Sigma S_1 = \mathbf{0} \;\wedge\; s + \Sigma S_2 = \mathbf{0}$$

$$\Rightarrow \Sigma(S_1 \cup S_2) = \mathbf{0}$$

Example

```
p(X,Y,...) :- | r(X,Y1), p(X,Y2,...), join(Y1,Y2,Y).
p(X,Y,...) :- | X=Y.
join(A,A,B) :- | B=A.
```

- Suppose $c/<r_{s.1},1> + c/<r_{s.1},2> = \mathbf{0}$ **and** $c/<p_{s_0},1> = \mathbf{1}$. Then $c/<p_{s_0},2> = \overline{\mathbf{1}}$ holds, while all subgoals carry non-exclusive capabilities.
  - All capabilities distributed to the r's will be fully collected as long as all the r's return what they are given.

Properties

- Degeneration of unification to assignment
- Subject reduction
- Conservation of constructors
  - A reduction will not gain or lose any constructor in the goal
- Groundness
- Non-sharing of constructors at “exclusive” positions
- Partial solution to extended occur-check
  - detection of X = X (suicidal unification)

Related Work

- Relating CCP and π
  - new calculus (γ, ρ, Fusion, Solo, ...)
  - encoding one in the other
- Variants of π with nicer properties
- (Linear) types in other computational models
  - π, λ, typed MM, session types, ...
- Linear languages
  - Linear Lisp, Lilac, Linear LP, ...
- Compile-time GC
  - Mercury, Janus, ...
  - compiling streams into message passing

Conclusions

- A strictly linear, polarized subset of Guarded Horn Clauses
  - retains most of the power of CBC
  - allows resource sharing within the linear framework
- Capability type system supporting strict linearity
- A step towards a unified framework for non-sequential computing

Future Work

- Type reconstructor
- Occur-check problem
- Time (as well as space) bounds
- Programming support
  - help (1) writing strictly linear programs or (2) reconstructing them from their slices
- Constructs for mobile/real-time/embedded computing + implementation

Final Remark

- Constraint-based type systems can make CBC a simple, powerful, and safe language for parallel, distributed, and real-time computing. Its role in CBC is analogous to, but probably more than, the role of type systems in the λ-calculus.