Strong Moding in Concurrent Logic/Constraint Programming

Kazunori Ueda

Dept. of Information and Computer Science
Waseda University

Main source:
Ueda, K. and Morita, M., Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, Vol. 13, No. 1 (1994).

Introductory paper:
Ueda, K., I/O Mode Analysis in Concurrent Logic Programming. In *Theory and Practice of Parallel Programming*, LNCS 907, Springer, 1995.

Two Different Notions of Modes

1. Modes for reasoning about temporal properties (time-of-call/exit instantiation states) of variables

   e.g., Is `X` unbound when `p(X)` is called?

   – dependent on "computation rules"

2. Modes for reasoning about non-temporal properties

   e.g., Which occurrence of `X` in the configuration `p(X),q(X),r(X)` may instantiate `X` eventually?

   – independent of computation rules
   – closer to a language construct

Abstract interpretation (mainly) deals with 1. Strong moding deals with 2.

Logical Variables and Communication

Observation: Logical variables are used for:

1. blackboard (competitive) communication (many writers, many readers), or
2. cooperative communication under established protocols:
   (a) point-to-point communication (one writer, one reader),
   (b) multicasting or broadcasting (one writer, many readers).

   — Failure is regarded as exception.

In both LP and Concurrent LP, it seems important to

1. distinguish between the two uses, and to
2. be able to infer the communication protocols for cooperative communication.

Strong moding provides compile-time support for "structured" communication.

Linear and Non-linear Clauses

A clauses is *linear* iff each variable occurs exactly twice in it.

Observation: Many clauses are linear in both LP and Concurrent LP —

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

— though (of course) this is not always the case:

- Shared ground data
  `p(...,X,...) :- r(X), p(...,X,...).`
- Wildcards
  `p(_).`
- Receive, check, and use
  `p(...,X,...) :- X>0, p(...,X,...).`
- Nonlinear math
  `p(X,Y) :- Y is X*X.`

Concurrent Logic/Constraint Programming

Relational Language, Concurrent Prolog, Parlog, (Flat) GHC, KL1, FCP, Oc, Doc, Strand, Janus, AKL, Oz, . . .

Communication mechanism: two interpretations

	algebraic	logical
representing information	*substitutions*	*constraints (equality etc.)*
receiving	*matching*	*ask (entailment)*
sending	*unification*	*tell (publication)*

Basic constructs for reactive concurrent programming seem to be converging to

$$\boxed{\textit{ask} + \textit{(eventual) tell}}\ .$$

Logical Variables as Communication Channels

Pros:

- *Natural* — virtually no synchronization bugs.
- *Simple and expressive*
  – data- and demand-driven computation
  – incomplete messages with reply boxes
  – evolving process structures
  – (streams of)* streams
  – difference lists, . . .
- *A message sequence is a first-class object, not a special language construct*.

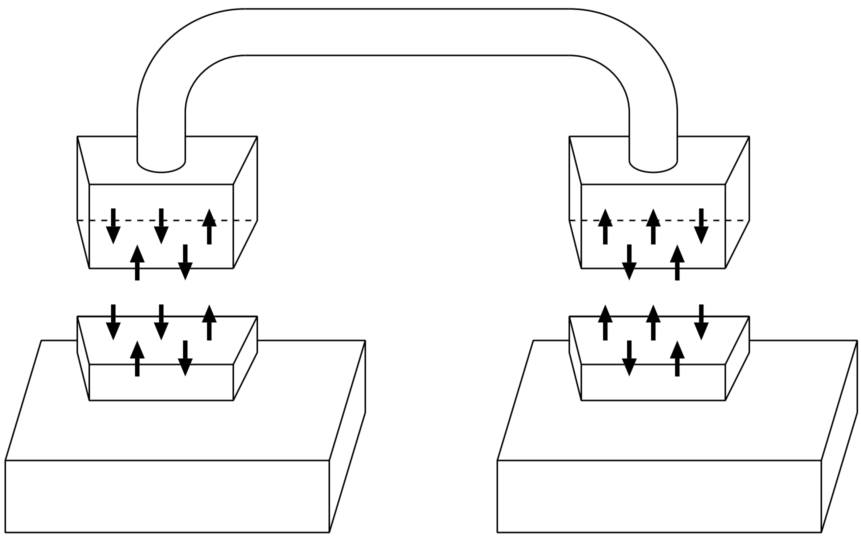
Cons:

- Bidirectionality may lead to inefficiency.
- Variables are inadvertently used for non-cooperative communication.
  — "Unification failure" means that the store collapses.

Modes: An Electric Device Metaphor

Signal cables may have various structures (arrays of wires/pins). However,

- 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	$\Longleftrightarrow$	Device
Variable	$\Longleftrightarrow$	Cable

Strong Moding

Deals with the *dynamic* (but implementation-independent) properties of programs *statically*:

- dataflow aspect (protocols)
- resource aspect

Shares advantages with strong typing:

1. helps programmers understand their programs better
2. compile-time detection of mode/type errors
3. compile-time establishment of fundamental properties
4. basic information for program optimization
5. encourages modular programming

Possible problems:

- explicit moding is burdensome
- monomorphism is too restrictive

Mode Inference: The Idea

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

We want to design a simple set of rules that allows us to infer properties such as:

- the 1st and 2nd arguments are input streams,
- the 3rd argument is an output stream, and
- the communication protocols used by these streams are identical.

Requirements:

- Retain the expressive power of the language.
- Allow efficient analysis.
- Allow separate analysis.

Syntax of a Subset of Flat GHC (or Oc)

$$
\begin{array}{rrcl}
\text{(program)} & P & ::= & \text{set of } R\text{'s} \\
\text{(program clause)} & R & ::= & A \ \texttt{:-} \ B \\
\text{(body)} & B & ::= & \text{multiset of } G\text{'s} \\
\text{(goal)} & G & ::= & T_1 \,\texttt{=}\, T_2 \quad | \quad A \\
\text{(atom)} & A & ::= & p(T_1\texttt{,} \ldots \texttt{,}T_n), \quad p \neq \text{`}\texttt{=}\text{'} \\
\text{(term)} & T & ::= & \text{(as in first-order logic)} \\
\text{(goal clause)} & Q & ::= & \texttt{:-} \ B
\end{array}
$$

For simplicity, we ignore

- guard goals and
- non-linear heads (i.e., non-left-linear clauses).

Operational Semantics

Configuration: $\langle B, C, V \rangle$ such that $\mathcal{V}_B \cup \mathcal{V}_C \subseteq V$ ($\mathcal{V}_F$: set of all variables in F)

The execution of `:-` B_0 starts with $\langle B_0, \emptyset, \mathcal{V}_{B_0} \rangle$.

$$\frac{\mathcal{P} \vdash \langle B_1, C, V \rangle \longrightarrow \langle B_1', C', V' \rangle}{\mathcal{P} \vdash \langle B_1 \cup B_2, C, V \rangle \longrightarrow \langle B_1' \cup B_2, C', V' \rangle}$$

$$\frac{}{\mathcal{P} \vdash \langle \{t_1 = t_2\}, C, V \rangle \longrightarrow \langle \emptyset, C \cup \{t_1 = t_2\}, V \rangle}$$

$$\frac{}{\begin{array}{l}\{h \text{ :- } B\} \cup \mathcal{P} \vdash \\ \quad \langle \{b\}, C, V \rangle \longrightarrow \langle B, C \cup \{\overline{b} = \overline{h}\}, (V \cup \mathcal{V}_{h \text{ :- } B}) \rangle\end{array}}$$

$$\begin{pmatrix} \text{if } \mathcal{E} \models \forall (C \Rightarrow \exists \mathcal{V}_h (\overline{b} = \overline{h})) \\ \text{and } \mathcal{V}_{h \text{ :- } B} \cap V = \emptyset \end{pmatrix}$$

The Mode System of Moded Flat GHC

- The mode system assigns polarity *structures* to predicate arguments

  — so that each part of data structures will be determined cooperatively, namely by *exactly one* process.

- Admits mode *inference* as well as mode *declaration* + mode *checking*.

- Constraint-based.

- Decidable and efficient — constraints can be solved (mostly) as unification over feature graphs.

The Mode System of Moded Flat GHC

A mode tells which process will determine which parts of data structures, or the dataflow aspect of communication protocols.

The 'parts' are specified by *paths* (= string of $\langle\text{symbol},\text{arg}\rangle$ pairs).

```
db([update(3,a),search(5,X)|...])
                          ↑
```

→ `X` occurs at the path

$$\langle \texttt{db}, 1\rangle\langle \texttt{.}, 2\rangle\langle \texttt{.}, 1\rangle\langle \texttt{search}, 2\rangle.$$

A set of program/goal clauses is *well-moded* if it satisfies all the mode constraints imposed by individual clauses.

→ Mode analysis

= constraint solving (in general)

= unification over feature graphs (in practice)

The Mode System: Definition

- *Pred*/*Fun*/*Var*: set of pred./func./variable symbols
- *Term*/*Atom*: set of terms/atoms over *Pred*, *Fun*, *Var*
- $N_p \stackrel{\text{def}}{=} \{1, 2, \ldots, n_p\}$, for each n_p-ary $p \in \mathit{Pred}$
- $N_f \stackrel{\text{def}}{=} \{1, 2, \ldots, n_f\}$, for each n_f-ary $f \in \mathit{Fun}$
- $P_{\mathit{Atom}} \stackrel{\text{def}}{=} (\sum_{p \in \mathit{Pred}} N_p) \times (\sum_{f \in \mathit{Fun}} N_f)^*$

  typical element: $\langle p, i\rangle\langle f_1, j_1\rangle \ldots \langle f_n, j_n\rangle$

- $\widetilde{a} : P_{\mathit{Atom}} \to \mathit{Var} \cup \mathit{Fun} \cup \{\bot\}$, for each $a \in \mathit{Atom}$ returns the symbol at the given path (or $\bot$).
- P_{Term} and $\widetilde{t}$ ($t \in \mathit{Term}$) are defined similarly.

Modes as functions

- $M \stackrel{\text{def}}{=} P_{\mathit{Atom}} \to \{\mathit{in}, \mathit{out}\}$

cf. definition of infinite trees

The Mode System (Continued)

$m : M \ (= P_{Atom} \to \{in, out\})$

$p \in P_{Atom}, \quad q \in P_{Term}$

Submodes

- $m/p : P_{Term} \to \{in, out\}, \quad q \mapsto m(pq)$
- $(m/p)/q \stackrel{\text{def}}{=} m/(pq)$

Mode inversion

- $\overline{in} \stackrel{\text{def}}{=} out, \quad \overline{out} \stackrel{\text{def}}{=} in$
- $\overline{m} : M, \quad p \mapsto \overline{m(p)}$
- $\overline{m/p} \stackrel{\text{def}}{=} \overline{m}/p$

Constant submodes

- $IN : P_{Term} \to \{in, out\}, \quad q \mapsto in$
- $OUT \stackrel{\text{def}}{=} \overline{IN}$

Mode Analysis

Purpose: To find a *well-moding* $m : M$ which satisfies all the mode constraints syntactically imposed by the (program, goal) pair.

Constraints imposed by a clause h :- B:

(HF) $\widetilde{h}(p) \in \mathit{Fun} \Rightarrow m(p) = \mathit{in}$

(BU) $(t_1 =_k t_2) \in B \Rightarrow m/\langle =_k, 1\rangle = \overline{m/\langle =_k, 2\rangle}$

(BF) $a \in B \wedge \widetilde{a}(p) \in \mathit{Fun} \Rightarrow m(p) = \mathit{in}$

(BV) $v \in \mathit{Var}$ occurs $n\,(\geq 1)$ times in h and B at $p_1, \ldots, p_n$, of which the occurrences in h are at $p_1, \ldots, p_k$ $(k \geq 0)$

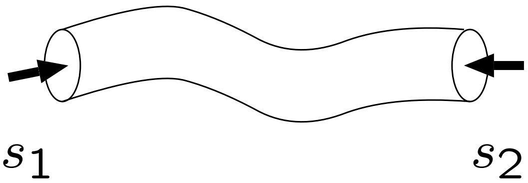
$$\Rightarrow \begin{cases} \mathcal{R}(\{m/p_1, \ldots, m/p_n\}), & k = 0; \\ \mathcal{R}(\{\overline{m/p_1}, m/p_{k+1}, \ldots, m/p_n\}), & k > 0; \end{cases}$$

where $\mathcal{R}$ is a 'cooperativeness' relation:

$$\mathcal{R}(S) \stackrel{\text{def}}{=} \forall q \in P_{\mathit{Term}}\, \exists s \in S$$
$$(s(q) = \mathit{out} \wedge \forall s' \in S \backslash \{s\}\, (s'(q) = \mathit{in}))$$

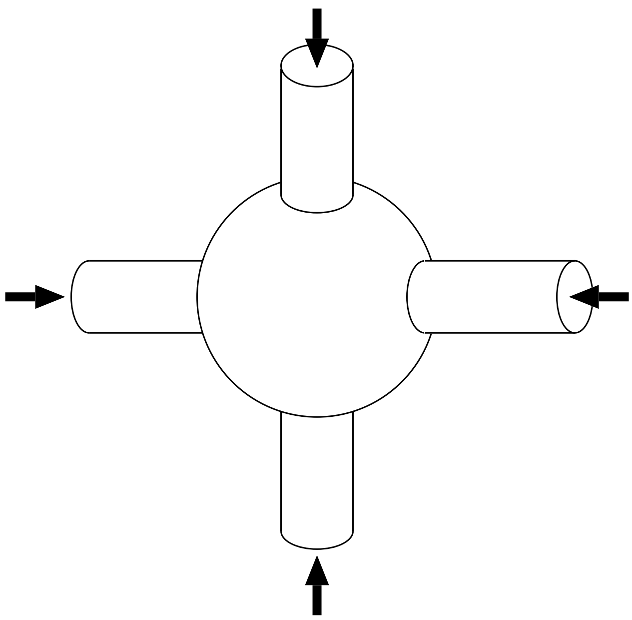
Principles Behind the Constraints

A Variable is a Cable . . .

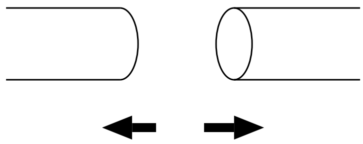


$$\mathcal{R}(\{s_1, s_2\}) \Leftrightarrow s_1 = \overline{s_2}$$

. . . Or a Hub.



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

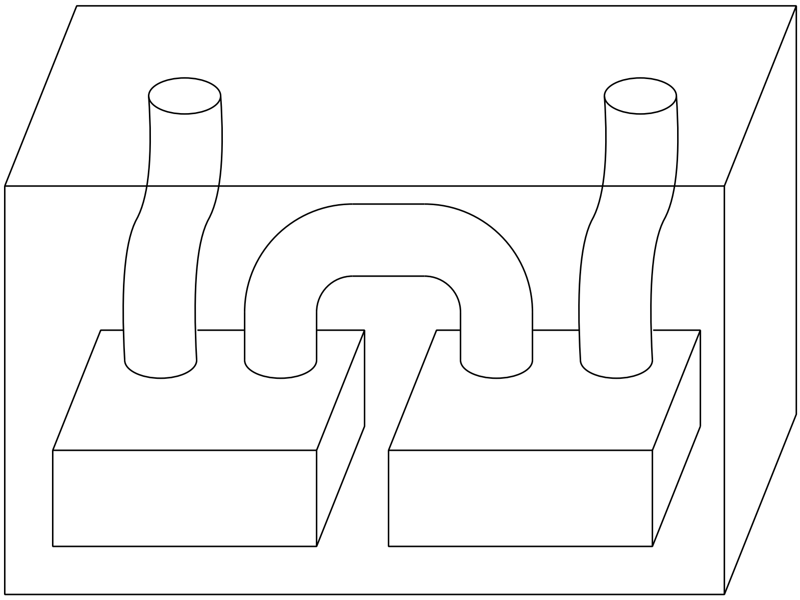


$$s_1 = \overline{s_2}$$

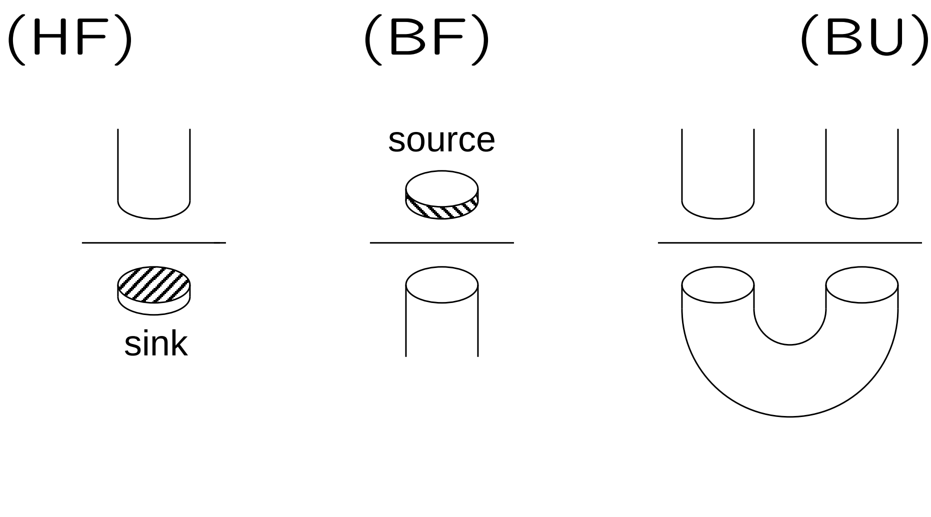
Constraint for Connection

Principles Behind the Constraints

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

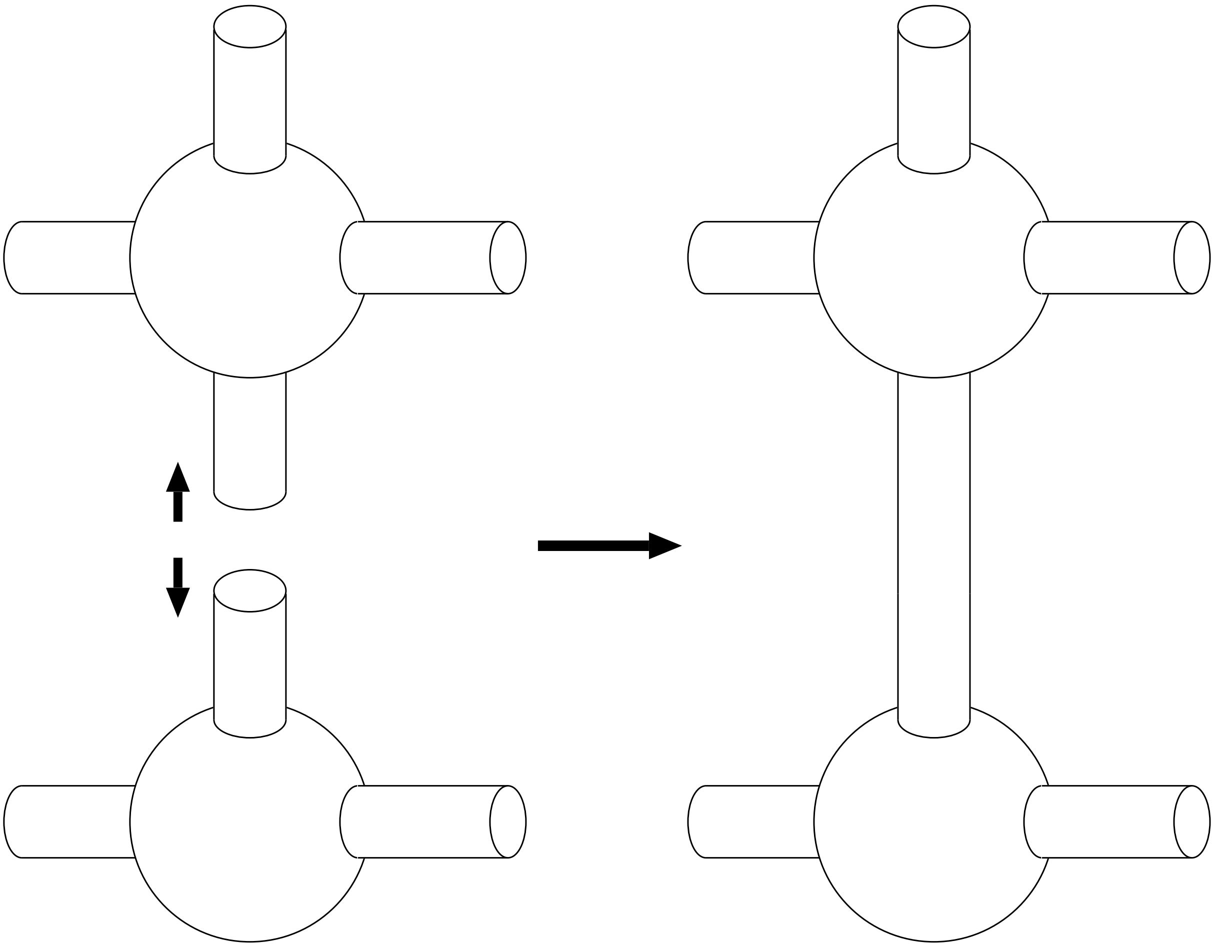


Goal-head connection



Resolution Principle

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



How Mode Analysis Works

$\texttt{merge([],Y,Z) :- Z=}_1\texttt{Y.}$

$\texttt{merge(X,[],Z) :- Z=}_2\texttt{X.}$

$\texttt{merge([A|X],Y,Z0) :- Z0=}_3\texttt{[A|Z],merge(X,Y,Z).}$

$\texttt{merge(X,[A|Y],Z0) :- Z0=}_4\texttt{[A|Z],merge(X,Y,Z).}$

From the third clause:

$m(\langle \texttt{merge}, 1\rangle) = in$ by (HF) applied to "."

$m/\langle \texttt{=}_3, 1\rangle = \overline{m/\langle \texttt{=}_3, 2\rangle}$ by (BU) applied to $\texttt{=}_3$

$m(\langle \texttt{=}_3, 2\rangle) = in$ by (BF) applied to "."

$m/\langle \texttt{merge}, 1\rangle\langle ., 1\rangle = m/\langle \texttt{=}_3, 2\rangle\langle ., 1\rangle$

by (BV) applied to A

$m/\langle \texttt{merge}, 1\rangle\langle ., 2\rangle = m/\langle \texttt{merge}, 1\rangle$

by (BV) applied to X

$m/\langle \texttt{merge}, 2\rangle = m/\langle \texttt{merge}, 2\rangle$

by (BV) applied to Y

$m/\langle \texttt{merge}, 3\rangle = m/\langle \texttt{=}_3, 1\rangle$ by (BV) applied to Z0

$m/\langle \texttt{=}_3, 2\rangle\langle ., 2\rangle = \overline{m/\langle \texttt{merge}, 3\rangle}$

by (BV) applied to Z

How Mode Analysis Works

```
merge([],Y,Z) :- Z=₁Y.
merge(X,[],Z) :- Z=₂X.
merge([A|X],Y,Z0) :- Z0=₃[A|Z],merge(X,Y,Z).
merge(X,[A|Y],Z0) :- Z0=₄[A|Z],merge(X,Y,Z).
```

Number of constraints generated: 24

($m(p) = in$: 6, $m/p_1 = m/p_2$: 12, $m/p_1 = \overline{m/p_2}$: 6)

By eliminating constraints on $=_k$, we obtain

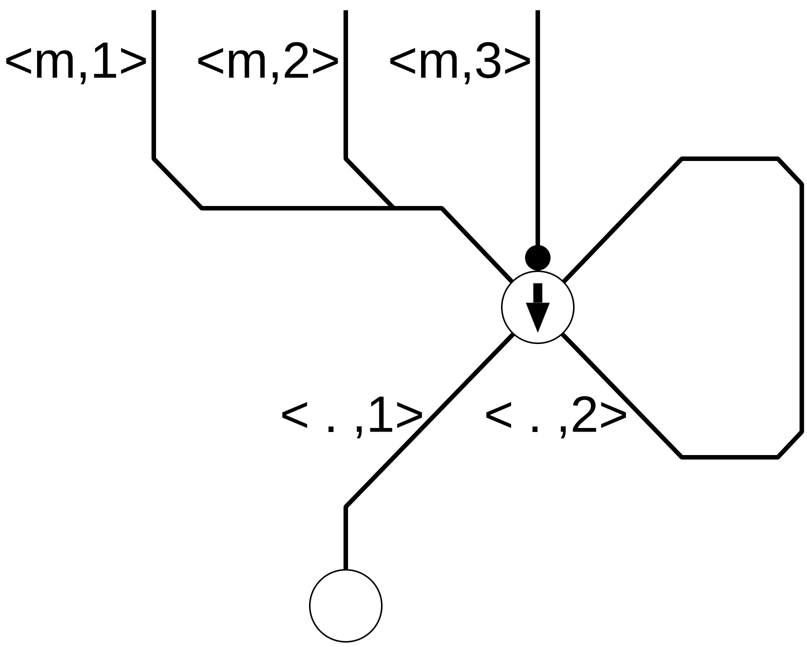
$$
\begin{aligned}
m(\langle \texttt{merge}, 1\rangle) &= in \\
m/\langle \texttt{merge}, 1\rangle\langle .\,, 2\rangle &= m/\langle \texttt{merge}, 1\rangle \\
m/\langle \texttt{merge}, 2\rangle &= m/\langle \texttt{merge}, 1\rangle \\
m/\langle \texttt{merge}, 3\rangle &= \overline{m/\langle \texttt{merge}, 1\rangle}
\end{aligned}
$$

How to deal with them efficiently?

Mode Graphs and Principal Mode Schemes

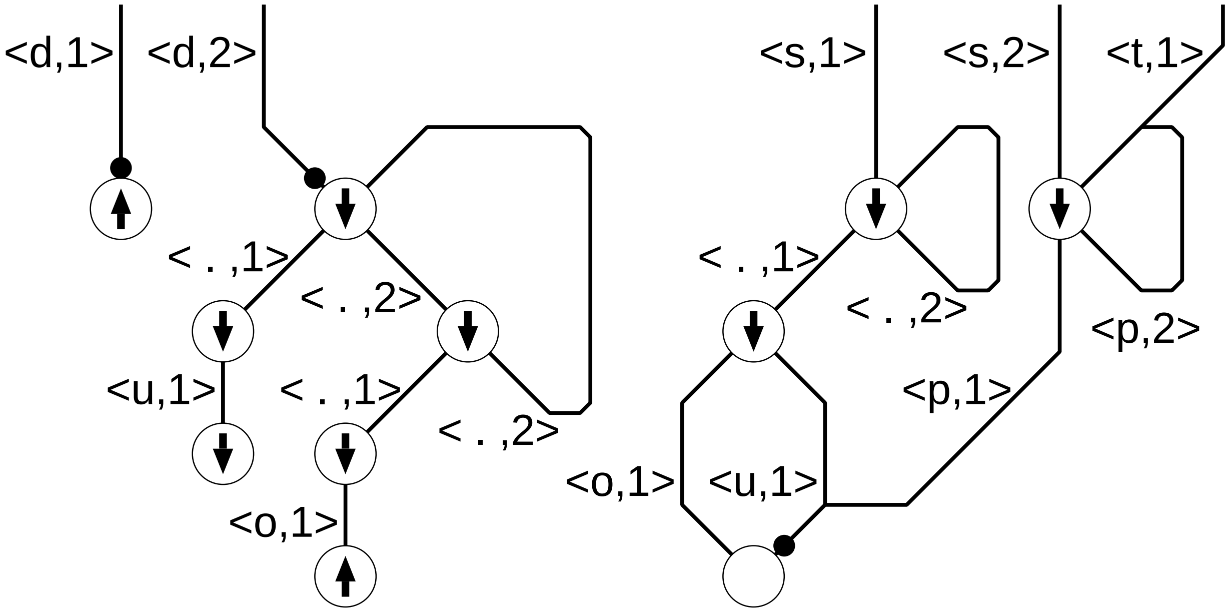
A set of mode constraints forms a "*principal mode scheme*" that can best be expressed as a *mode graph*.

$$
\begin{aligned}
m(\langle \texttt{merge}, 1\rangle) &= in \\
m/\langle \texttt{merge}, 1\rangle\langle ., 2\rangle &= m/\langle \texttt{merge}, 1\rangle \\
m/\langle \texttt{merge}, 2\rangle &= m/\langle \texttt{merge}, 1\rangle \\
m/\langle \texttt{merge}, 3\rangle &= \overline{m/\langle \texttt{merge}, 1\rangle}
\end{aligned}
$$



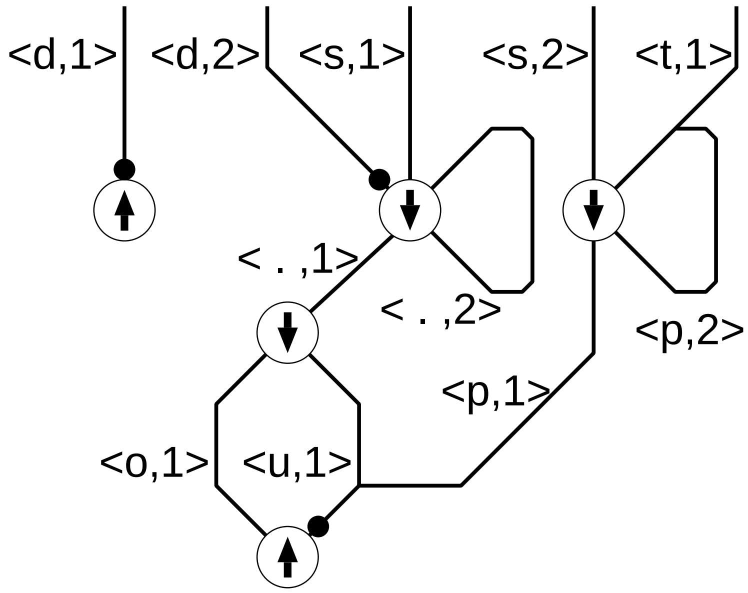
A Stack & Driver Example

```
drive(M,S) :- M=:=0 | S =₁ [].
drive(M,S) :- M=\=0 | S =₂ [push(M),pop(N)|S1],
     subtract(N,1,N1), drive(N1,S1).
stack([],          D) :- terminate(D).
stack([push(X)|S],D) :- stack(S,p(X,D)).
stack([pop(X)|S], p(Y,D1)) :-
     X =₃ Y, stack(S,D1).
```

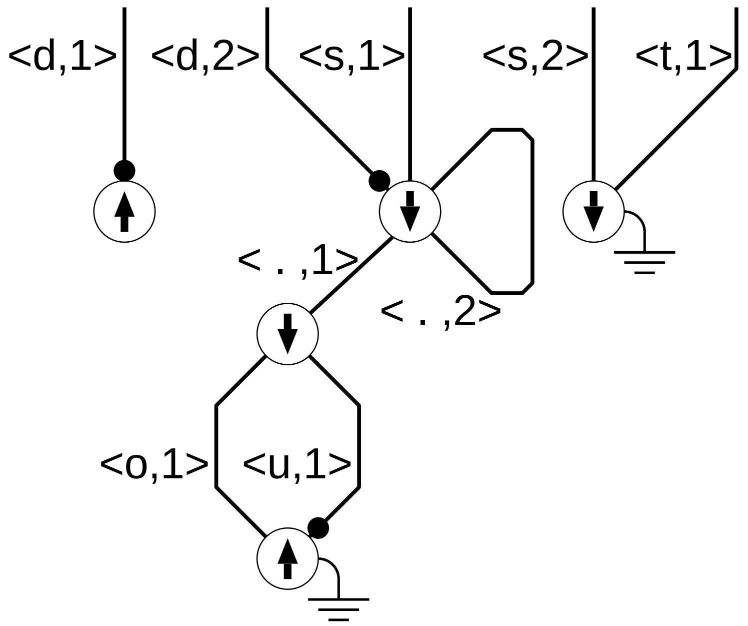


Unification of Mode Graphs

```
...,stack(S,none),drive(10,S),...
```

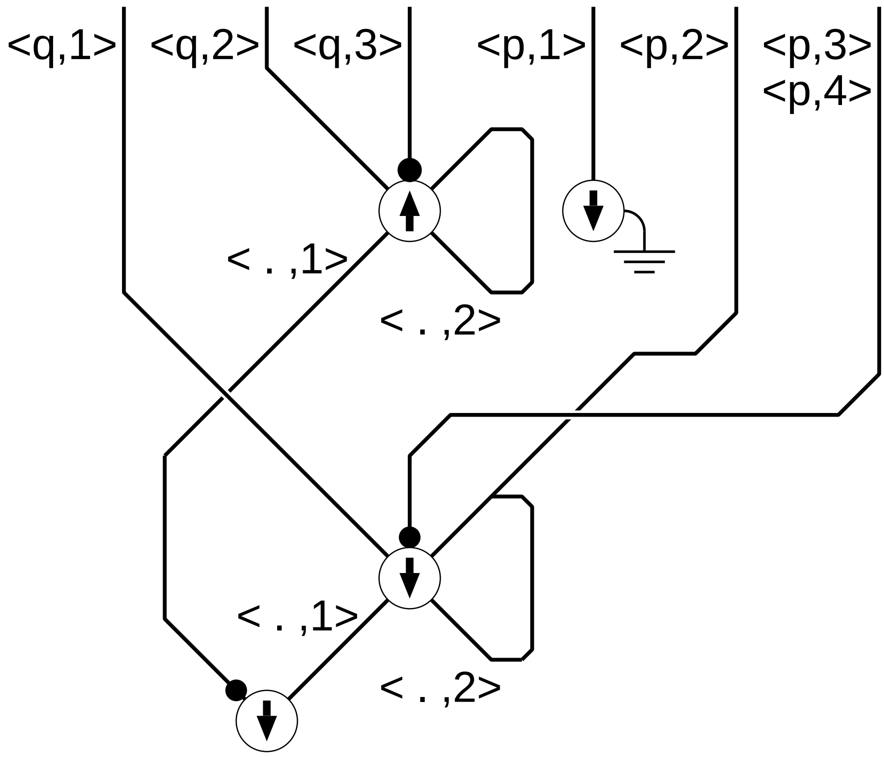



```
terminate(D).
```



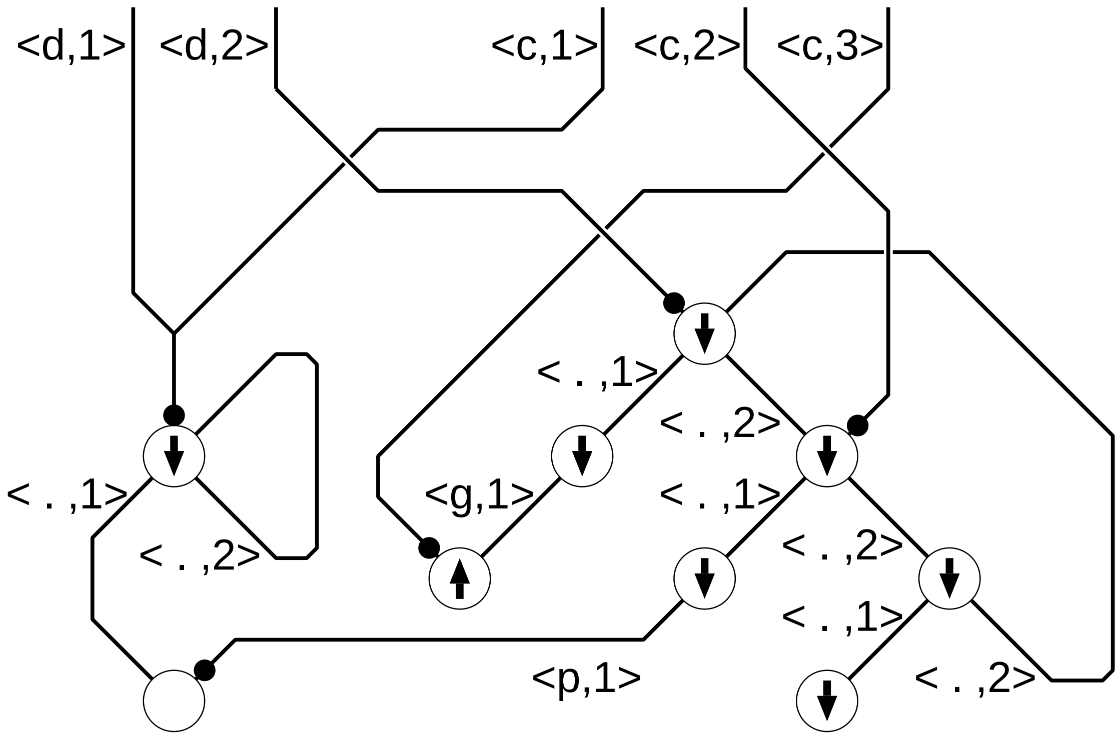
Difference Lists

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



Mutual Recursion

```
driver(Fs,IOs0):-
  IOs0=[gett(X)|IOs1],checkinput(Fs,IOs1,X).
checkinput(Fs, IOs, done):- Fs=[],IOs=[].
checkinput(Fs0,IOs0,more):-
  Fs0=[N|Fs1],IOs0=[putt(N),nl|IOs1],
  driver(Fs1,IOs1).
```



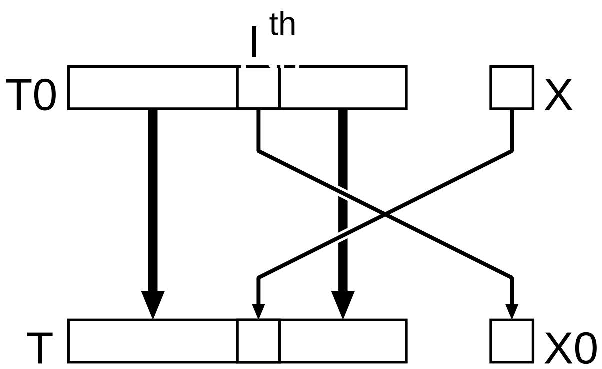
Arrays

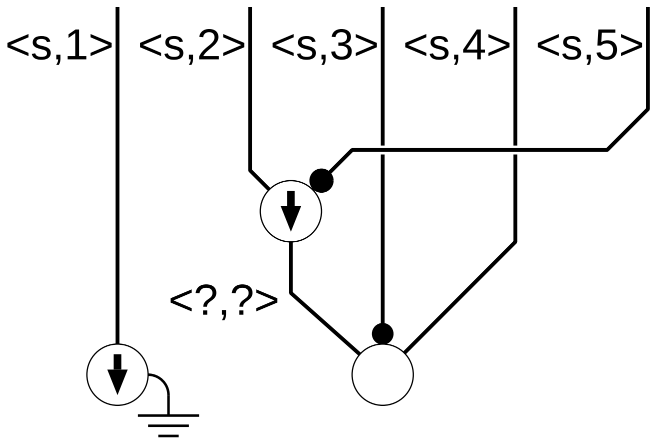
In Prolog, `arg` (and destructive `set_arg`) provides array access functionalities.

Principle: For built-in predicates, consider the mode constraints imposed by their virtual definitions.

Under the mode system, the *basic* access operation should be:

```
set_arg(I, T0, X0, X, T)
```





Moral: Values are resources. Array elements should be *removed* when accessed.

Arrays

In Prolog, `functor(-,+,+)` initializes the arguments of the created structure with distinct fresh variables, which are *instantiated* if necessary.

Under the mode system, the arguments should be initialized by constants, which are *updated* by `set_arg`/5.

— The update is in-place if the array occurs at a single-reference path.

Other generic array operations:
– swap
– split
– concatenate
– change-shape (for multidimensional arrays),
. . .

Singleton Variables

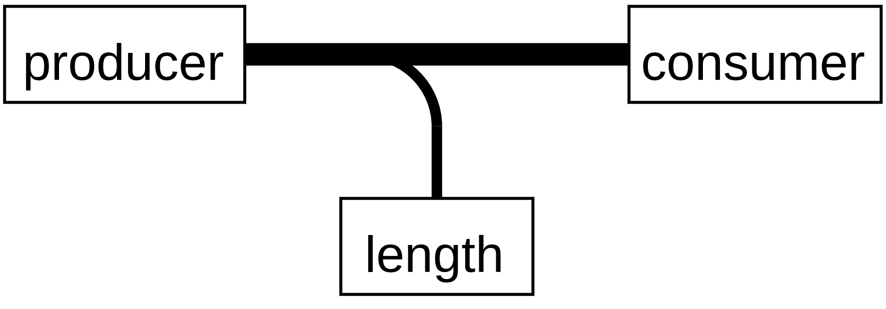
A singleton variable may well be a misspelled variable.
cf. a "dangling" cable and an unplugged socket

`p(X0, ...):- ...,p(XO, ...).`
$\rightarrow m/\langle \mathtt{p}, 1\rangle = IN \wedge m/\langle \mathtt{p}, 1\rangle = OUT$
$\rightarrow$ mode error

Note: Not all singleton variables indicate errors.

```
length([],   N0,N):- N:=N0.
length([_|L],N0,N):-
     N1:=N0+1,length(L,N1,N).
```

— `length` will be used as a "byway" process without affecting the protocol of the rest of the processes.



Well-Moded Programs Do Not Go Wrong

P: a program
B: (body of) a goal clause
m: a mode

Lemma 1
If $B : m$ and B contains $t_1 =_k t_2$, at least one of t_1 and t_2 is a variable.

Theorem 1*
If $P : m$, $B : m$ and $P \vdash \langle B, \emptyset, V \rangle \longrightarrow \langle B', C', V' \rangle$, then C' is consistent and $B' \cdot C' : m$.

($B' \cdot C'$: B' instantiated by C')

Corollary 1*
If $P : m$, $B : m$ and $P \vdash \langle B, \emptyset, V \rangle \xrightarrow{*} \langle B', C', V' \rangle$, then C' is consistent and $B' \cdot C' : m$.

******Holds unless the extended occur check (which excludes unification of the form v=v) fails.*

Extended Occur Check

A goal of the form $v = v$ creates a meaningless short-circuit.

The pair

$$
\begin{array}{lrl}
G: & & \texttt{:- p(A,A), q(A), r(A).} \\
P: & \texttt{p(X,Y)} & \texttt{:- X=Y.}
\end{array}
$$

imposes $m/\langle \mathtt{q}, 1\rangle = m/\langle \mathtt{r}, 1\rangle = IN$,
while the reduced goal

$$
G': \quad \texttt{:- q(A), r(A).}
$$

imposes $m/\langle \mathtt{q}, 1\rangle = \overline{m/\langle \mathtt{r}, 1\rangle}$, which would violate Theorem 1.

Groundness Property Follows from the Termination Property

Theorem 2
Suppose $P : m$, $B : m$, and B has succeeded under the extended occur check. Then for each v in B, a unification goal $\overset{-}{v} = \overset{+}{t}$ or $\overset{+}{t} = \overset{-}{v}$ must have been executed.

Corollary 2
The final store maps all the variables in B to *ground* terms.

Cost of the Mode Analysis

- The number of constraints imposed: O(n) ($\forall$-quantified or non-quantified), where n is the size of the program

- Adding one unary/binary constraint

  $\approx$ unification of a feature graph with another small feature graph

  $\approx$ merging of top-level features + merging of submode graphs

  $\rightarrow$ O($d{\cdot}\alpha(n)$) time, where
  d: size of the subgraph to be unified ('complexity' of data structures),
  α: inverse of the Ackermann function

- Total cost:

  O($nd{\cdot}\alpha(n)$) for all-at-once analysis
  O($n \log n + nd{\cdot}\alpha(n)$) for separate analysis

Non-Unary/Binary Constraints

- Imposed by non-linear clauses.
- Cannot be represented by mode graphs.
- Should be delayed — many of $n(>2)$-ary constraints will be reduced to unary/binary ones by other constraints.

  ```
  p(X, ...) :- r(X, ...), p(X, ...)
  ```
  $\rightarrow m/\langle \mathtt{r}, 1\rangle = \mathit{IN}$,
  $\quad m/\langle \mathtt{p}, 1\rangle = (\mathit{unconstrained})$

- Some constraints may remain unreduced, whose satisfiability must be checked eventually.
- The practical solution is to let programmers declare the modes of the paths where non-linear variables occur.

Implications to Programming Style

1. Advocates the "*programming as wiring*" paradigm, or (equivalently) *programming with linear clauses*.

   — leads to more generic mode schemes
   — encourages "structured dataflow"
   — less error-prone

2. Encourages graceful termination.

   — A process cannot discard its arguments upon termination if it contains variables to instantiate. (cf. Corollary 2)
   (e.g., an output stream must be terminated by `[]`.)

   — All streams will be closed upon termination.

Path-Based Program Analysis

Path-based analysis can be used also for

- the distinction between one-to-one and possibly one-to-many communication
  — which paths are 'shared' paths?
- type systems
  — which paths are used for the data obeyed by the constraint system $\mathcal{C}$?
  — what function symbols may appear at this path?

Resources received at linear (non-shared) paths in a clause head can be

- reclaimed (compile-time garbage collection) if not passed to the body, or
- locally recycled to represent new data used in the body,

without any run-time checking (e.g., reference counting).

Resource-Conscious Programming

Insertion sort example:

```
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).
```

By slight modification, becomes *linear w.r.t. data resources*:

```
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).
```

Extension: Polymorphic Modes

A polymorphic predicate is allowed to have different modes for each call.

Typical example: ‘`=`’ (unification)

For polymorphic predicates,

- compute their principal mode schemes (i.e., mode graphs), and
- allow different calls to have different instantiations of them. (e.g., $\texttt{merge}_1$, $\texttt{merge}_2$, ...)

This can be implemented by making a *copy* of the mode graph for each call, rather than the original graph.

(cf. ML: $(\lambda x.A)E$ vs. let $x = E$ in A)

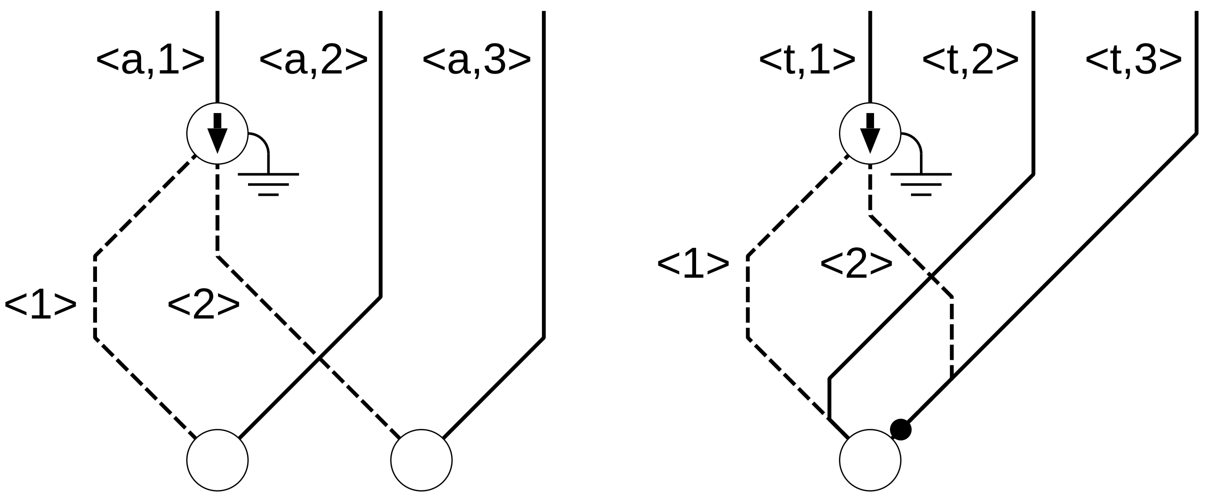
Extension: Higher-Order

`call` is just a predicate with the constraint $m/\langle \texttt{call}, 1\rangle = m$ (by confusing pred. and func. symbols).

`apply` needs extension.

```
twice(P,X,Z):- apply(P,X,Y),apply(P,Y,Z).
```

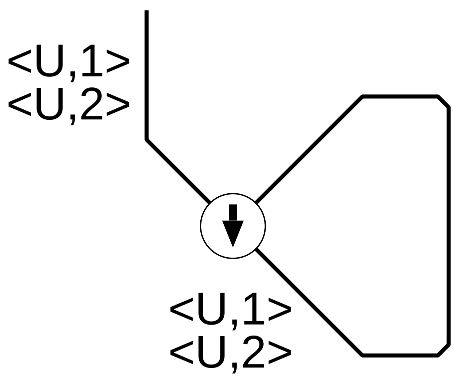
Whether `P` is
– a predicate symbol,
– a list of clauses (ground representation), or
– a compiled code with mode information,
it is a ground term at the first-order, but must have a predicate mode as well. The moding in the monomorphic case would be:



Extension: Non-Herbrand Constraint Systems

- Rational terms — Immediate.
- Numerical constraints — Can be moded if dataflow can be determined statically.
- Equational theories
  — Associativity and commutativity can be included naturally (they preserve resources).
  — Idempotency involves resource contraction/copying.

Example: Bags (= multisets) enjoy
$t_1 \cup t_2 = t_2 \cup t_1$ and $t_1 \cup (t_2 \cup t_3) = (t_1 \cup t_2) \cup t_3$.
So the paths for bags should obey the constraint:



Constraint-Based Program Analysis

- Abstract interpretation usually computes fixpoints by iteration, while constraint-based analysis computes fixpoints by unification (or constraint solving)
  — on the assumption that a single iteration should lead to a fixpoint.
- Constraint-based analysis provides unified treatment of
  - *declaration* — constraints provided by a programmer,
  - *checking* — consistency checking between constraints from the program and those given by programmers,
  - *inference* — constraint solving.
- Incremental — inherently amenable to separate analysis.

Related Work and Future

Languages

– Strand (Foster and Taylor)
– Doc (Hirata)
– A'UM (Yoshida et al.)
– Moded Flat GHC (Ueda et al.)
– Janus (Saraswat et al.)

Implementation of the Mode System

– MGTP-based (Koshimura et al.)
– Mode graph (Ueda)
– Another mode graph (Tick et al.)

Applications and Future Work

– Message-oriented Implementation (sequential and parallel) (Ueda et al.)
– Optimized distributed unification
– Style checker
– High-performance computing
. . .