並行論理プログラミング言語
GHC / KL1

上田 和紀

早稲田大学理工学部情報学科

ueda＠ueda.info.waseda.ac.jp

Copyright (C) 2000 Kazunori Ueda

逐次と並行

- 「並行」(concurrency) は、ごく普通の現象。
  – 社会
  – 宇宙
  – 個人
  – 人間 – 計算機系
  – 計算機システム内
  – 電子回路
- 「逐次」(sequentiality) は、計算機の世界に意図的に持ち込まれたもの。
  – cf. フォンノイマン計算機

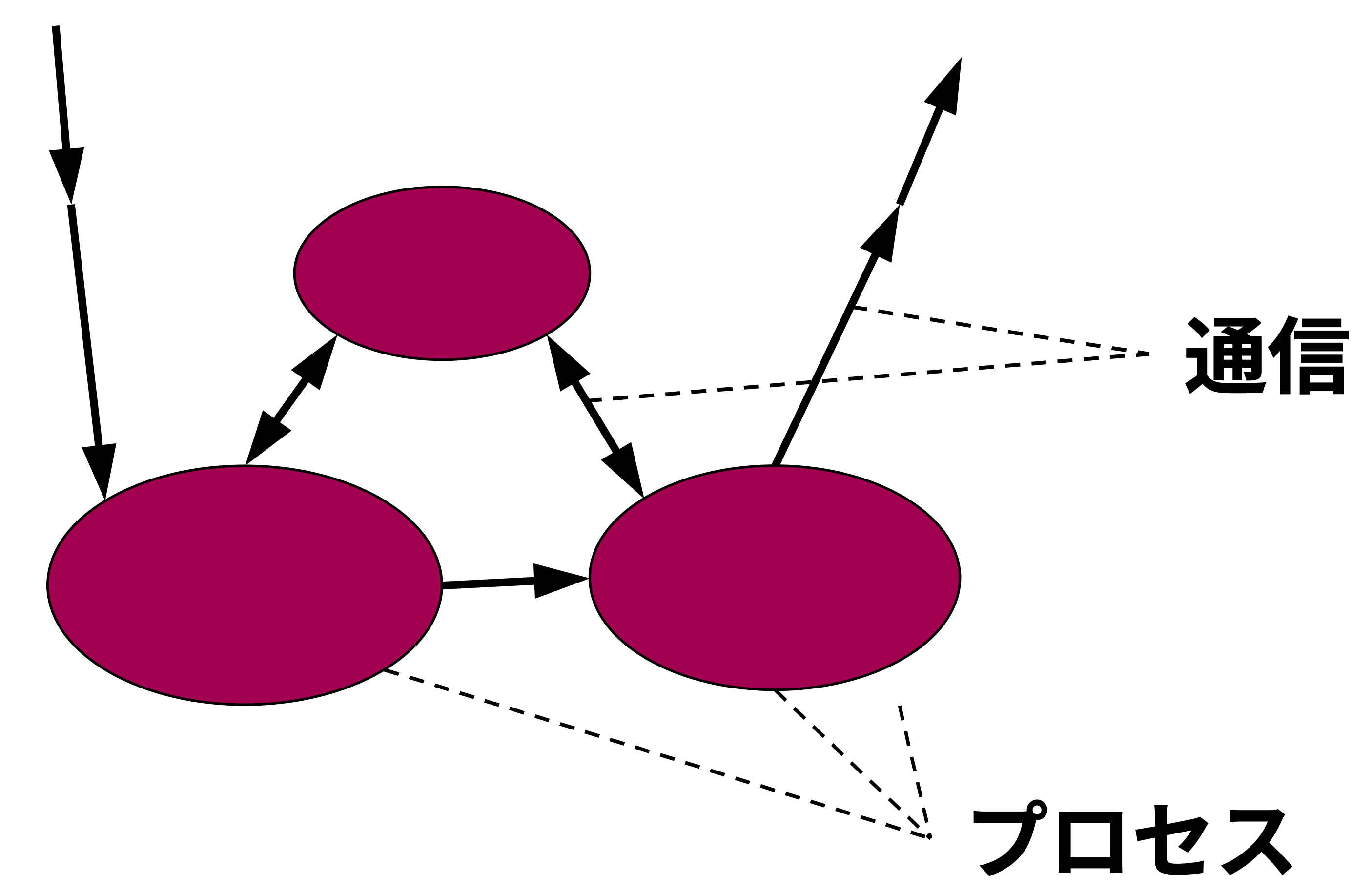
逐次と並行

- 仕事（プロセス）と通信・同期
  - 大きな仕事は、たいがいは複数の小さな仕事の集まりに分解できる。
  - 複数の仕事が独立ならば、並列処理は難しくない。
  - 独立でなければ、仕事の間で**通信**（情報交換）が必要となる。
  - 通信があれば、**同期**（待合せ）が必要となる。

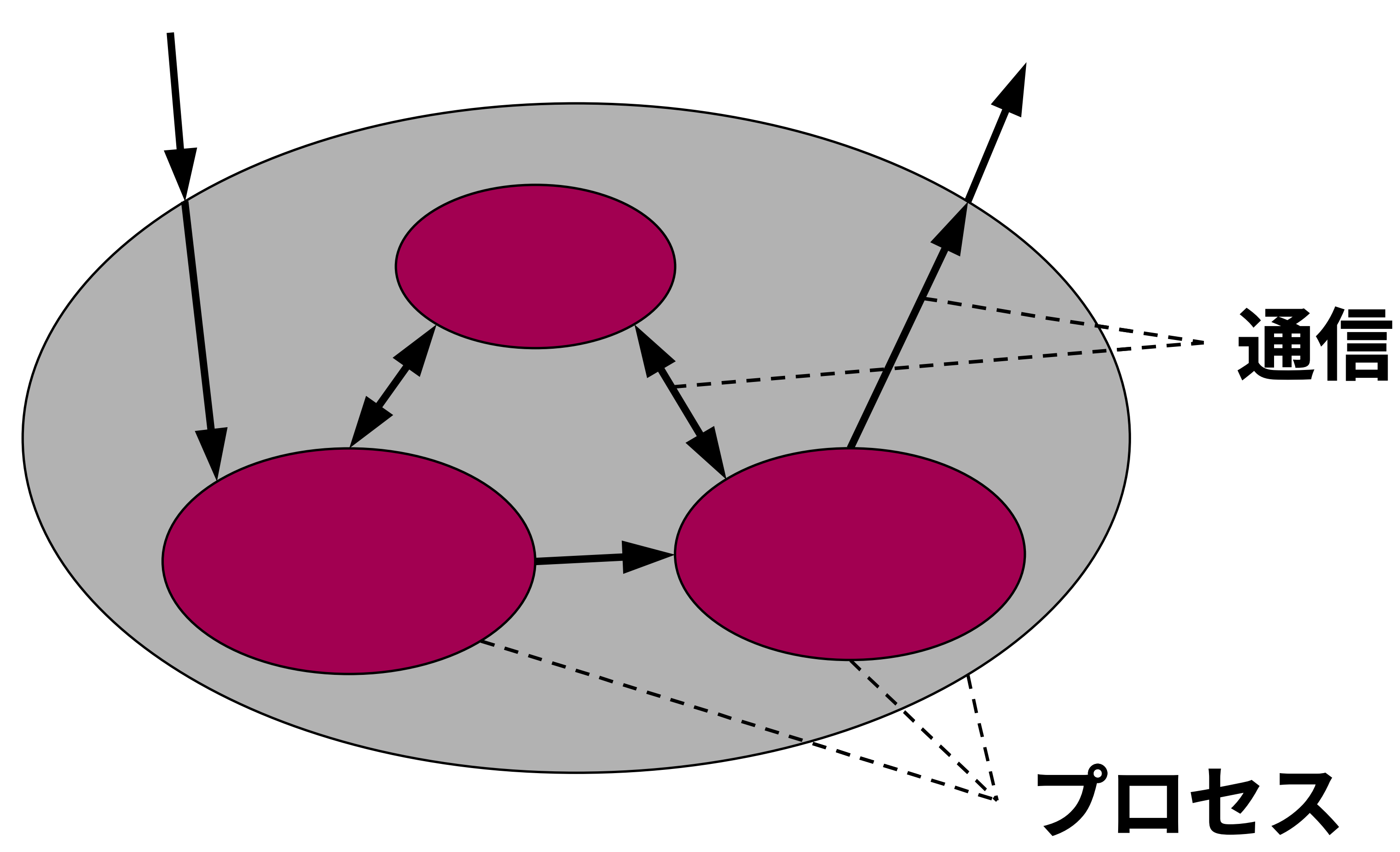
逐次と並行

- 典型的通信方法：
  - 電話：　同期通信
  - 郵便：　非同期通信
  - 黒板：　共有メモリ

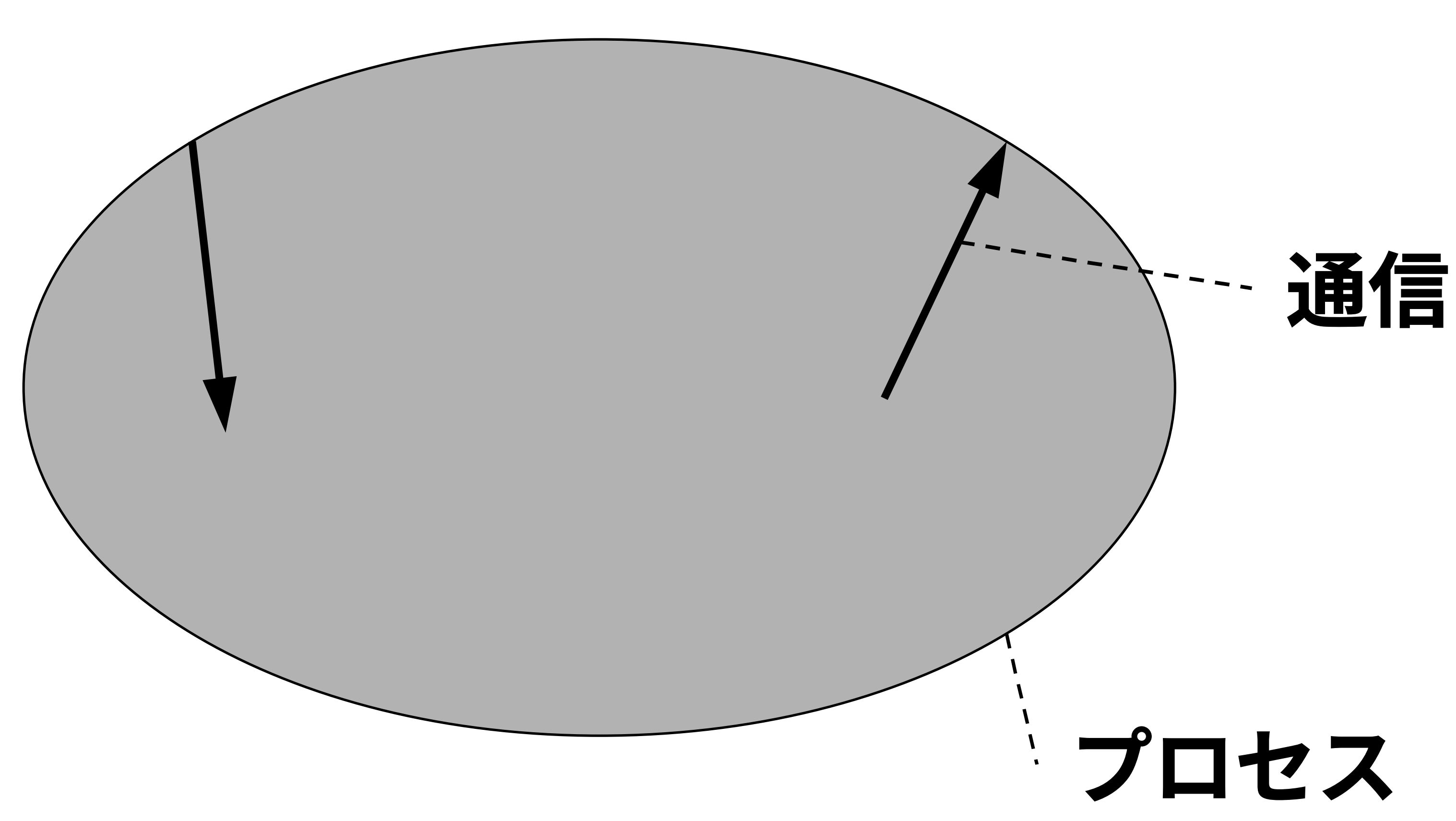
並行計算



並行計算



並行計算



並行・並列・分散

- 「仕事Ａと仕事Ｂを並行して進める」
  「２時から委員会Ａと委員会Ｂを並列開催」
- 管理職の仕事は並行＋並列プログラミング
  – 並行：仕事の分割、依存関係の解析
  – 並列：部下への仕事の割当て
- 並列処理の目的は、「分業」による時間効率の改善
- 分散処理の目的は、空間的な広がりの中での協調作業

並行プログラミングは難しいか?

- 意見１: Yes
  - 認知科学者によれば、人間の思考は本質的に逐次的である。
  - 言語機能が逐次プログラミングよりも単純になるはずがない。
- 意見２: No.
  - 適切な表現手段（言語）を使えば、並行処理の記述は、逐次処理と比べて別に難しくないはずだ。

並列プログラミングは難しいか?

- 答1: No.
  - 正しい並列プログラムを書くことは難しくない。つまり、適切な表現手段（言語）を使えば、並行プログラミングは難しくない。
- 答2: Yes.
  - 性能の良い並列プログラムを書くのは難しい。

並行プログラミング言語の設計

- 方法１: 逐次プログラミング言語（C, C++, Lisp, . . .）を拡張する
- 方法２: 逐次性のない枠組みから出発する（cf. 電子回路、論理式、数式、. . .）

並行論理プログラミング

- 並行論理プログラミング（Concurrent Logic Programming, Concurrent LP）

Concurrent LP
= LP + directionality (of dataflow)
= logic + embedded concurrency control

cf. Algorithm = Logic + (external) Control
(Robert Kowalski, 1979)

復習：論理プログラミング

- プログラムは確定節の集合

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

- ゴールを与えて、融合原理（resolution）によって答（解代入）を求める

```
:- append(X,Y,[P,Q,R]).
→ {X=[], Y=[P,Q,R]},  {X=[P], Y=[Q,R]},
   {X=[P,Q], Y=[R]},   {X=[P,Q,R], Y=[]}.
```

- Yes-No型質問 vs. 穴埋め質問

復習：論理プログラミング

- プログラム P の下で、ゴール g が解代入 θ を計算して成功したとすると

  $P \models g\theta$ （$g\theta$ は P の論理的帰結）

  が保証される（健全性）

- $g\sigma$ が P の論理的帰結であるならば、

  $\theta \leq \sigma$ （θ は σ と同程度以上に一般的）

  であるような解代入 θ を計算する（完全性）

復習：論理プログラミング

- たとえば、

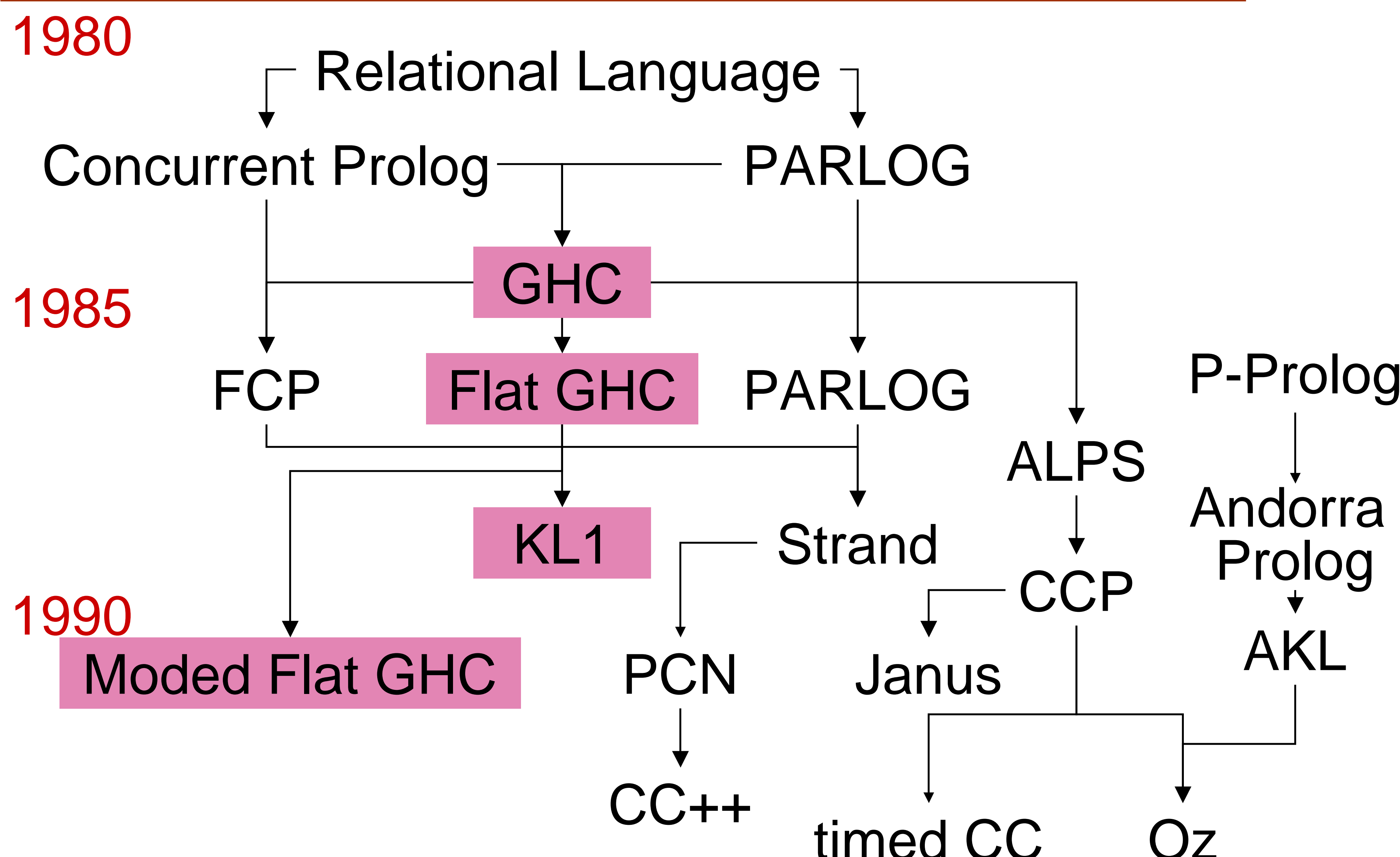
$$\forall Y(\mathrm{append}([],Y,Y)) \wedge$$
$$\forall X \forall Y \forall Z(\mathrm{append}(X,Y,Z) \Rightarrow \mathrm{append}([A|X],Y,[A|Z]))$$
$$\models \mathrm{append}([P],[Q,R],[P,Q,R])$$

- また、代入 $\{X=[1,2],\ Y=[3]\}$ は正しい解代入だが、これは計算された解代入の一つ $\{X=[P,Q],\ Y=[R]\}$ によってカバーされている。

復習：論理プログラミング

- ゴール :- append(X,Y,[P,Q,R]) を与えて解代入 {X=[P], Y=[Q,R]} を得ると見るかわりに、ゴール append(X,Y,Z) と入力代入 {Z=[P,Q,R]} から出力代入 {X=[P], Y=[Q,R]} **を得ると見ることもできる。**
- つまり、プログラムとゴールの組は、入力代入から出力代入を計算する非決定的（non-deterministic）関数とも解釈できる。
- 並行論理プログラミングは、この見方を拡張。

並行論理プログラミングの系譜

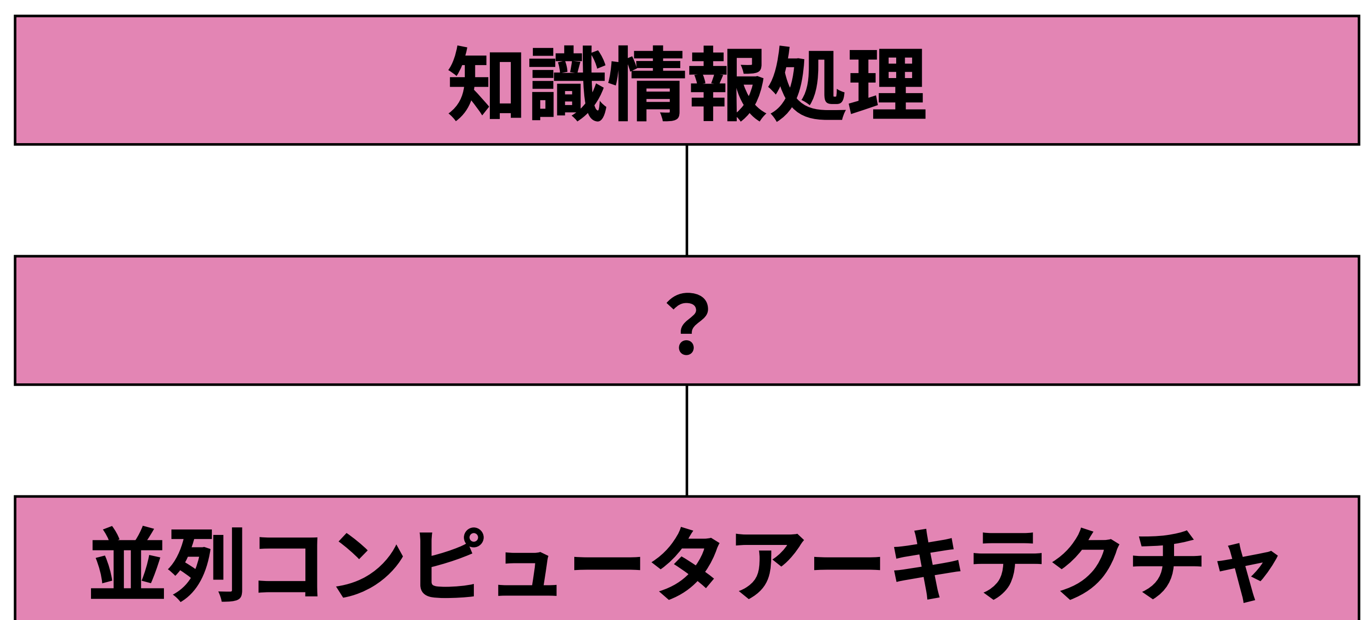


GHC / KL1 の設計目的

- 記号処理のための並行（・並列）プログラミング言語
  - 記号処理のために（cf. 手続き型言語）
    - »記号および記号構造の生成・操作機能
    - »自動メモリ管理機能
  - 並行・並列処理のために
    - »暗黙の並列性と同期機構 ー 自然な記述
    - »計算の物理資源へのマッピング ー 効率

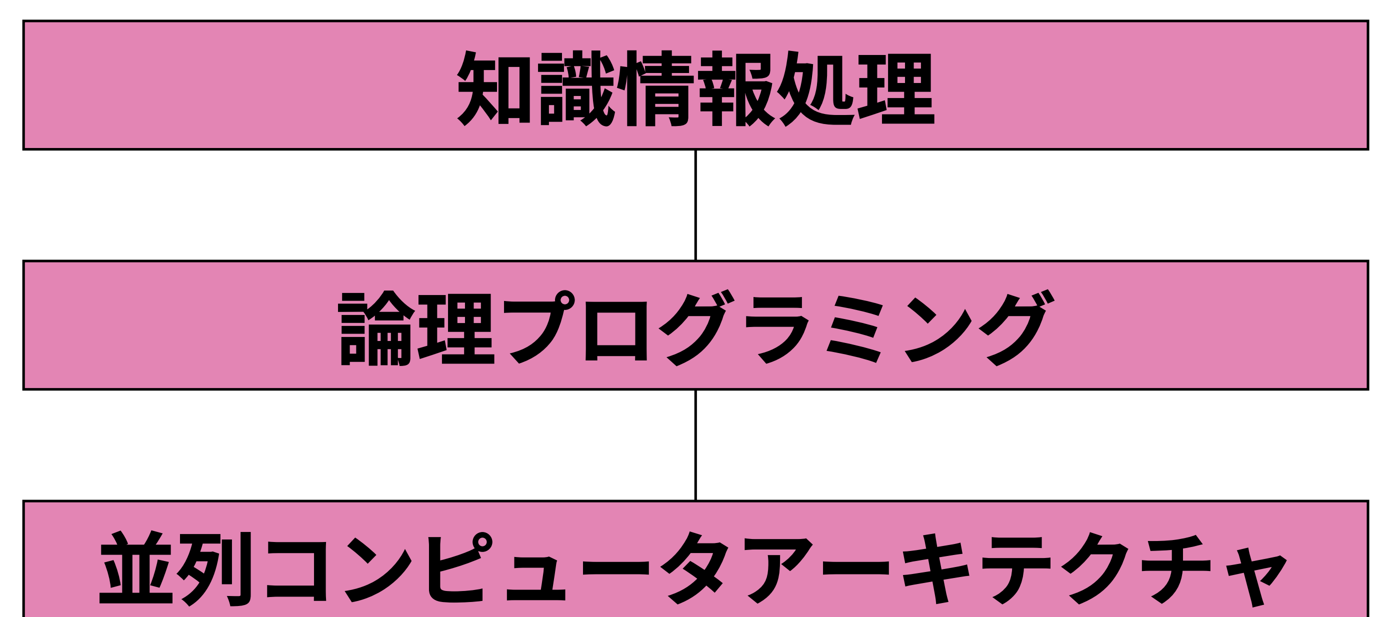
GHC / KL1 設計の経緯

- 第五世代コンピュータプロジェクト（1982〜1993）
  - 知識情報処理と並列処理を結ぶ方法論の追求



GHC / KL1 設計の経緯

- 第五世代コンピュータプロジェクト（1982～1993）
  - 作業仮説：論理プログラミングで結ぶ



GHC / KL1 設計の経緯

- 第五世代コンピュータプロジェクト（1982～1993）
  - 結果（cf. Comm. ACM, March 1993）

知識情報処理

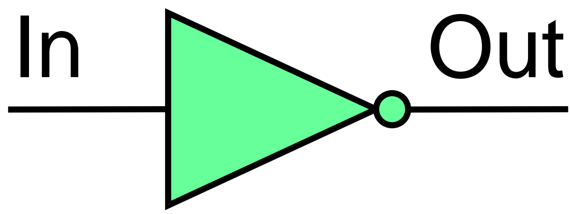
論理プログラミング

並行論理プログラミング

並列コンピュータアーキテクチャ

One-Shot Inverter in GHC

```
not(In,Out) :- In=0 | Out=1.
not(In,Out) :- In=1 | Out=0.
```

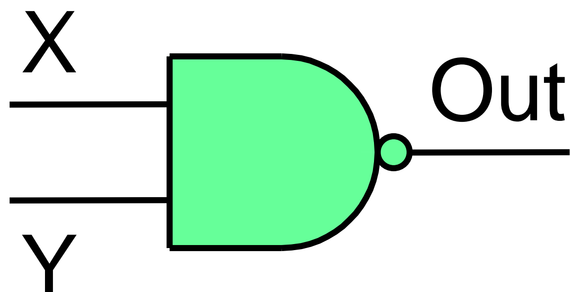


Or more concisely:

```
not(0,Out) :- true | Out=1.
not(1,Out) :- true | Out=0.
```

One-Shot NAND Gate

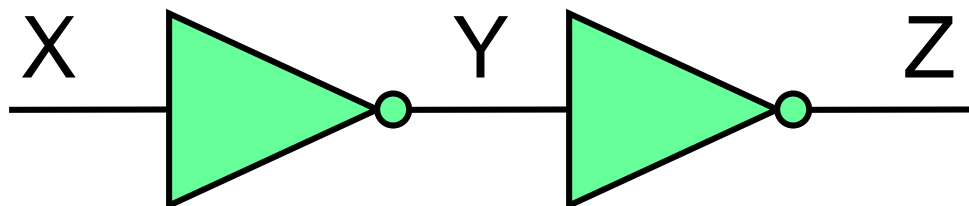
```
nand(0,Y,Out) :- true | Out=1.
nand(X,0,Out) :- true | Out=1.
nand(1,1,Out) :- true | Out=0.
```



Which rule will be selected if both X and Y are 0?

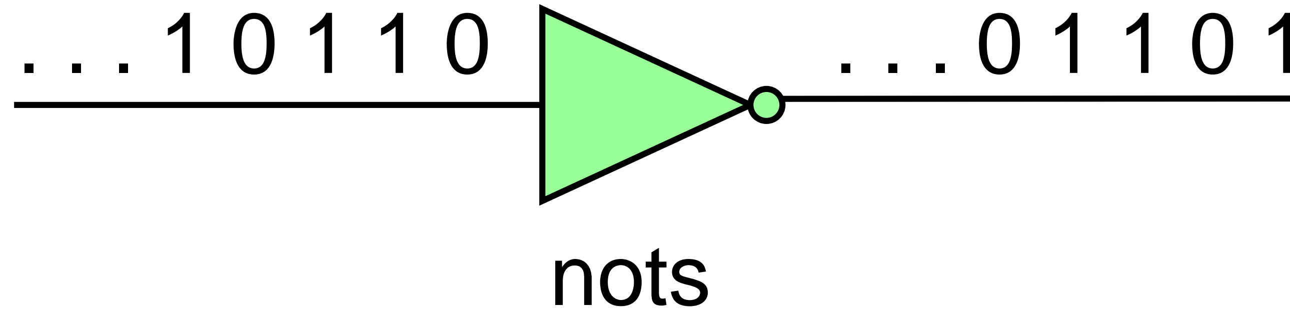
Cascaded Inverters

```
:- module main.
main :- true | not(Y,Z), not(X,Y),
   tty:ttystream([gett(X),putt([X,Y,Z]),nl]).
not(0,Out) :- true | Out=1.
not(1,Out) :- true | Out=0.
```



More on Inverters

Inverter accepting a sequence of input data



A sequence can be represented as a list.

Examples: [0,1,1,0,1]

[0,1,1,0,1|A]

[]

More on Inverters

```
nots([],     Y  ) :- true | Y=[].
nots([0|X],Y0) :- true | Y0=[1|Y], nots(X,Y).
nots([1|X],Y0) :- true | Y0=[0|Y], nots(X,Y).
```

Or, using “not”,

```
nots([],     Y  ) :- true | Y=[].
nots([A|X],Y0) :- true |
        not(A,B), Y0=[B|Y], nots(X,Y).
```

More on Inverters

- Behavior of “nots(X,Y)”:

Input	Output	Rest
X=[0,1,1,0,1]	Y=[1,0,0,1,0]	(none)

More on Inverters

- Behavior of “nots(X,Y)”:

Input	Output	Rest
X=[0,1,1,0,1]	Y=[1,0,0,1,0]	(none)
X=[]	Y=[]	(none)

More on Inverters

- Behavior of "nots(X,Y)":

Input	Output	Rest
X=[0,1,1,0,1]	Y=[1,0,0,1,0]	(none)
X=[]	Y=[]	(none)
X=[0,1,1,0,1|X′]	Y=[1,0,0,1,0|Y′]	nots(X′,Y′)

More on Inverters

- Behavior of "nots(X,Y)":

Input	Output	Rest
X=[0,1,1,0,1]	Y=[1,0,0,1,0]	(none)
X=[]	Y=[]	(none)
X=[0,1,1,0,1|X′]	Y=[1,0,0,1,0|Y′]	nots(X′,Y′)
(none)	(none)	nots(X,Y)

More on Inverters

- Behavior of "nots(X,Y)":

Input	Output	Rest
X=[0,1,1,0,1]	Y=[1,0,0,1,0]	(none)
X=[]	Y=[]	(none)
X=[0,1,1,0,1|X′]	Y=[1,0,0,1,0|Y′]	nots(X′,Y′)
(none)	(none)	nots(X,Y)
X=[2|_]	(reduction failure)	

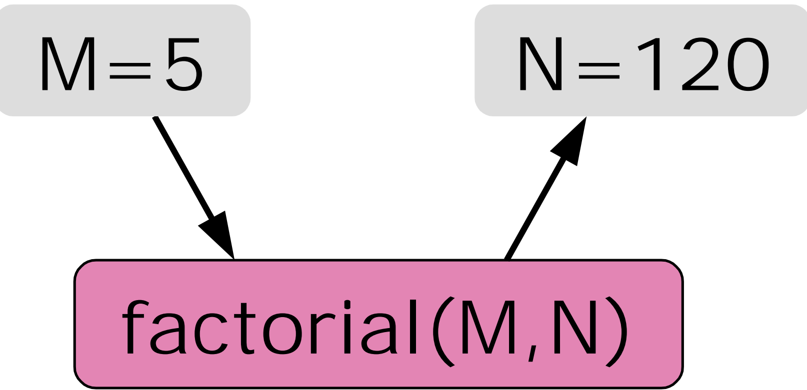
More on Inverters

- Behavior of "nots(X,Y)":

Input	Output	Rest
X=[0,1,1,0,1]	Y=[1,0,0,1,0]	(none)
X=[]	Y=[]	(none)
X=[0,1,1,0,1|X′]	Y=[1,0,0,1,0|Y′]	nots(X′,Y′)
(none)	(none)	nots(X,Y)
X=[2|_]	(reduction failure)	
X=[0|_], Y=[0|_]	(unification failure)	

Factorial

```
factorial(X,Y) :- X=:=0 | Y:=0.
factorial(X,Y) :- X > 0 |
   X1:=X–1, factorial(X1,Y1), Y:=X*Y1.
```



Factorial

```
factorial(X,Y) :- X=:=0 | Y:=0.
factorial(X,Y) :- X > 0 |
    X1:=X–1, factorial(X1,Y1), Y:=X*Y1.
```

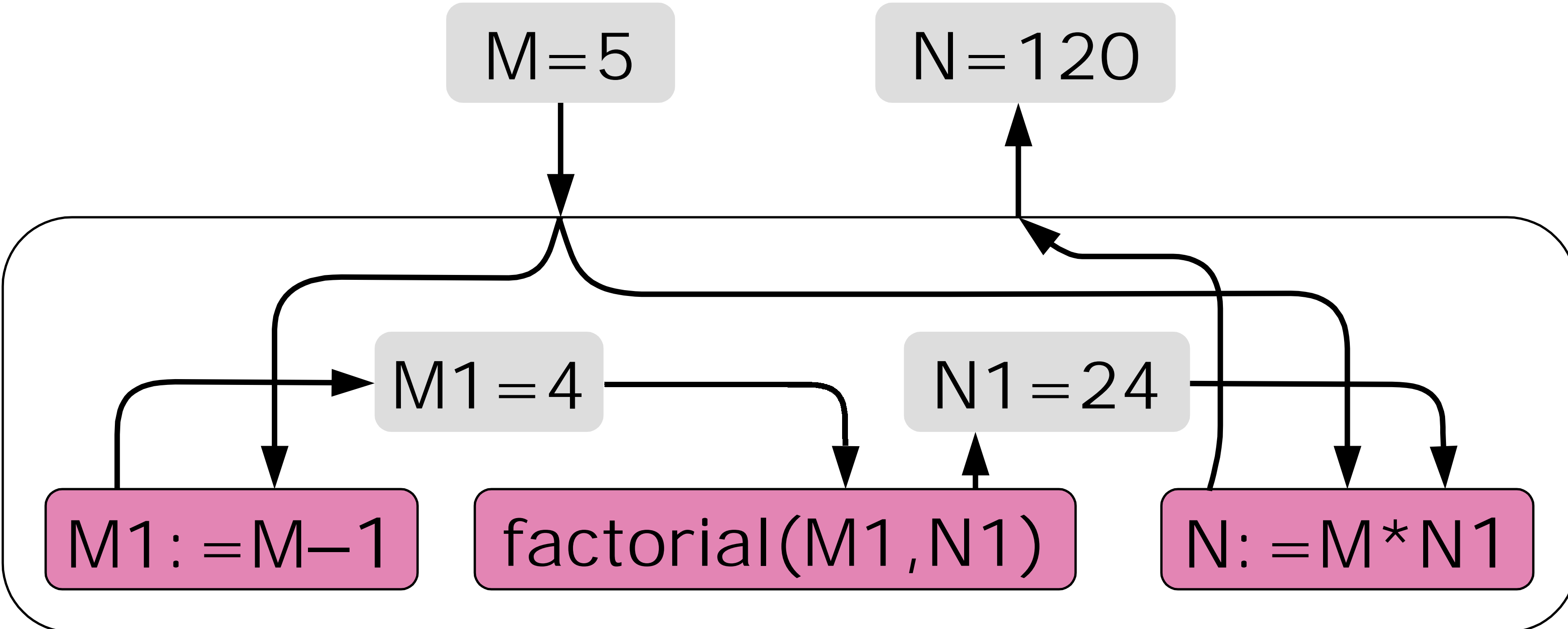
M=5

N=120

factorial(M,N)

Factorial

```
factorial(X,Y) :- X=:=0 | Y:=0.
factorial(X,Y) :- X > 0 |
   X1:=X–1, factorial(X1,Y1), Y:=X*Y1.
```



GHC (Guarded Horn Clauses)

Concurrent LP

= LP + directionality (of dataflow)

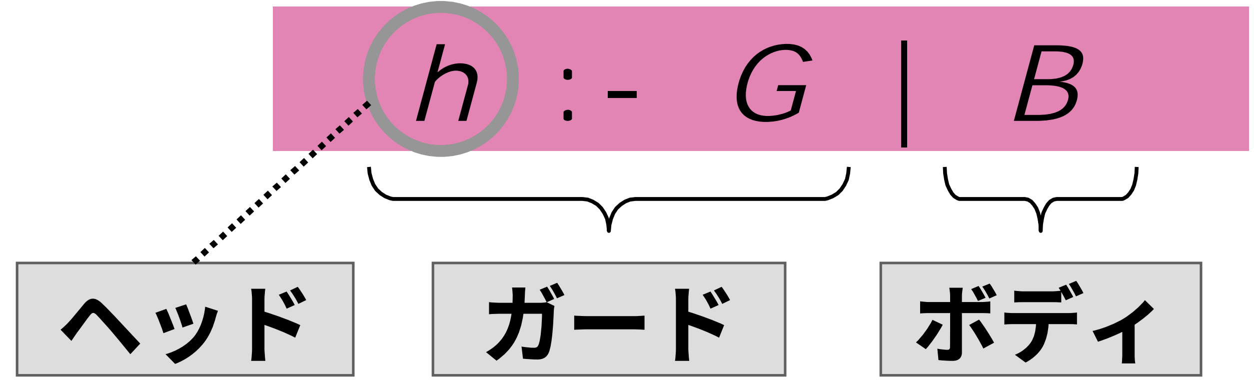
= logic + embedded concurrency control

GHC = Horn Clauses + Guards

(algorithm) (logic) (control)

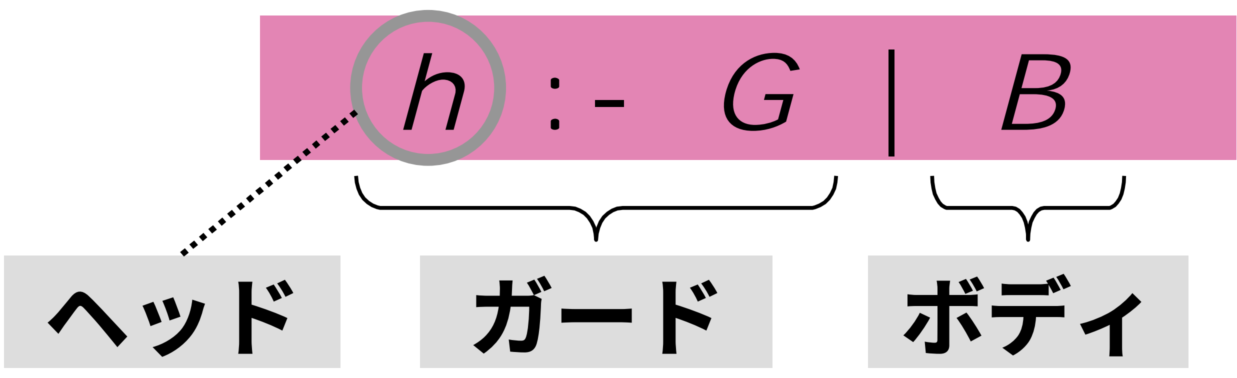
GHC の構文

- プログラムはプログラム節の集合。
- プログラム節はゴールの書換え規則。



- *h* : 原子論理式
- *G*, *B* : 原子論理式（ゴール）のマルチ集合

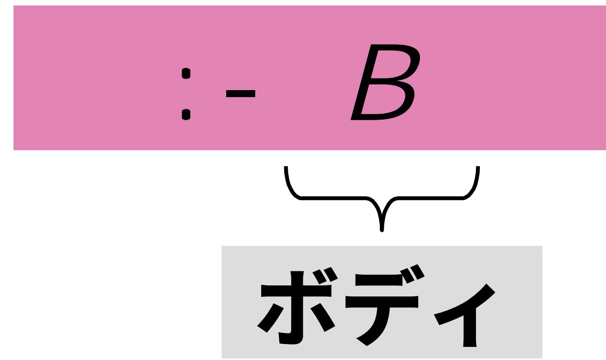
GHC の構文



- *h* : 原子論理式
  - 書き換えるゴールのテンプレート
- *G*, *B* : 原子論理式（ゴール）のマルチ集合
  - *G* : 書換えのための条件
  - *B* : 書換え後のゴールのマルチ集合
  - 空のマルチ集合は true と表記

GHC の構文

- プログラムを起動するには、**ゴール節**を使って、ゴールの初期マルチ集合を指定する。



- ゴール節で起動するのでなく、特定の述語を起動する処理系もある。
  – KLIC ではモジュール main の述語 main を起動。

単一代入変数（論理変数）

- （並行）論理プログラミング言語の変数は単一代入変数。
- 値の具体化（制約）はできるが変更はできない。
  - 具体化： X **不定** → X＝f(Y) → X＝f(5)
  - 変更：　 X **不定** → X＝1 → X＝2

単一化ゴールと非単一化ゴール

- 単一化ゴール
  - 単一化述語“ = ”は言語に組込み
  - ボディの単一化ゴール（$t_1 = t_2$）は、変数値を具体化して両辺を同じ形にする。
    - 情報の生成（＝送信）
    - 通常は、少なくとも一辺は変数である。つまり単一化は実質的に代入文。
    - ガードの単一化ゴールは、標準形への変換によって消去可能。

単一化ゴールと非単一化ゴール

- 非単一化ゴール
  - ユーザ定義のものと処理系提供のものとがある
  - プログラム節を用いて、他のゴールに書き換わってゆく
  - 非単一化ゴールを書き換えるときは、まず節のヘッドとマッチングをとる。マッチングでは、ヘッド側の変数値だけを具体化してよい
    - »情報の観測（＝受信）

ゴールの実行

- 複数のゴールは並列に実行してよい
- 書換規則の適用に十分な情報がなければ中断

```
main :- true | not(X,Y), not(1,X).
```

- 実行すべきゴールがなくなったら正常終了
- すべてのゴール中断したら永久中断

```
main :- true | not(X,Y), not(Y,X).
```

ゴールの実行

- どの書換規則も（永久に）適用できない非単一化ゴールがあったら書換えの失敗

```
main :- true | not(1,X), not(2,Y).
```

- 単一化ゴールが矛盾する情報（1＝2など）を生成したら（単一化の）失敗
（cf. 回路の短絡）

```
main :- true | not(0,X), not(1,X).
```

リスト処理 — append

```
:- module main.
main :- true | append([1,2,3],[4,5],X).
append([],      Y,Z  ) :- true | Y=Z.
append([A|X],Y,Z0) :- true |
             Z0=[A|Z], append(X,Y,Z).
```

cf. Prolog

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

リスト処理 — length

プログラム１

```
length([],       N) :- true | N:=0.
length([_|L0],N) :- true |
          length(L0,N0), N:=N0+1.
```

プログラム２（末尾再帰形）

```
length(L,N) :- true | length(L,0,N).
length([],       N0,N) :- true | N:=N0.
length([_|L0],N0,N) :- true |
          N1:=N0+1, length(L0,N1,N).
```

リスト処理 — list reversal

プログラム１

```
nreverse([],       Lr) :- true | Lr=[].
nreverse([A|L0],Lr) :- true |
    nreverse(L0,Lr0), append(Lr0,[A],Lr).
```

プログラム２（末尾再帰形）

```
reverse(L,Lr) :- true | reverse(L,[],Lr).
reverse([],      S,Lr) :- true | Lr=S.
reverse([A|L],S,Lr) :- true |
    reverse(L,[A|S],Lr).
```

リスト処理 — insertion sort

プログラム１

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

リスト処理 — insertion sort

プログラム２

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

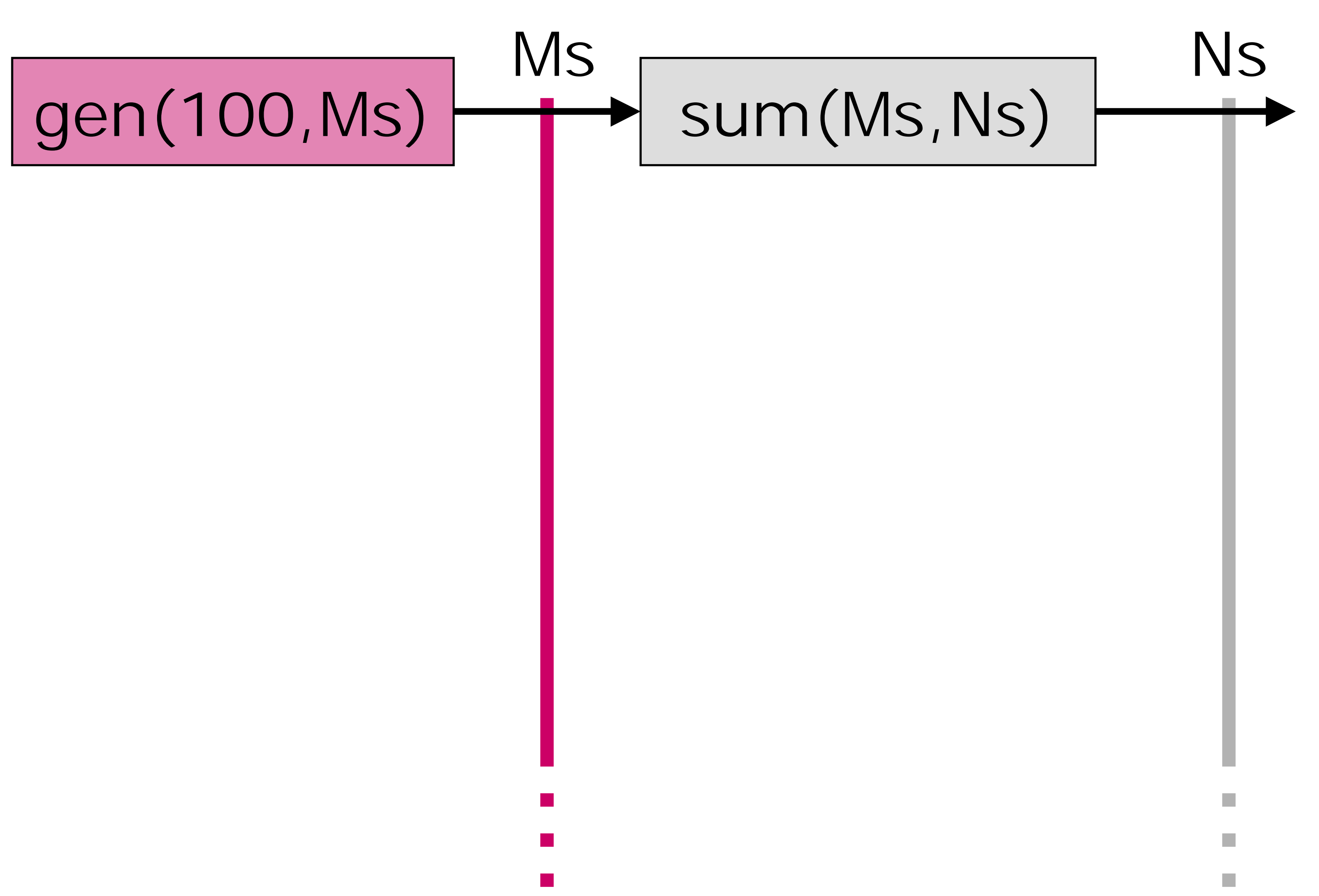
リストとストリーム

- 関数型プログラムにおけるリスト
  - 値呼び（call by value）で評価する限り、ボトムアップに作成される
- 論理型プログラムにおけるリスト
  - 先頭要素からトップダウンに（インクリメンタルに）作成することができる
    → ストリーム
    - 作り終わる前に読み始めることができる
    - 書き手と読み手の並行実行を可能にする

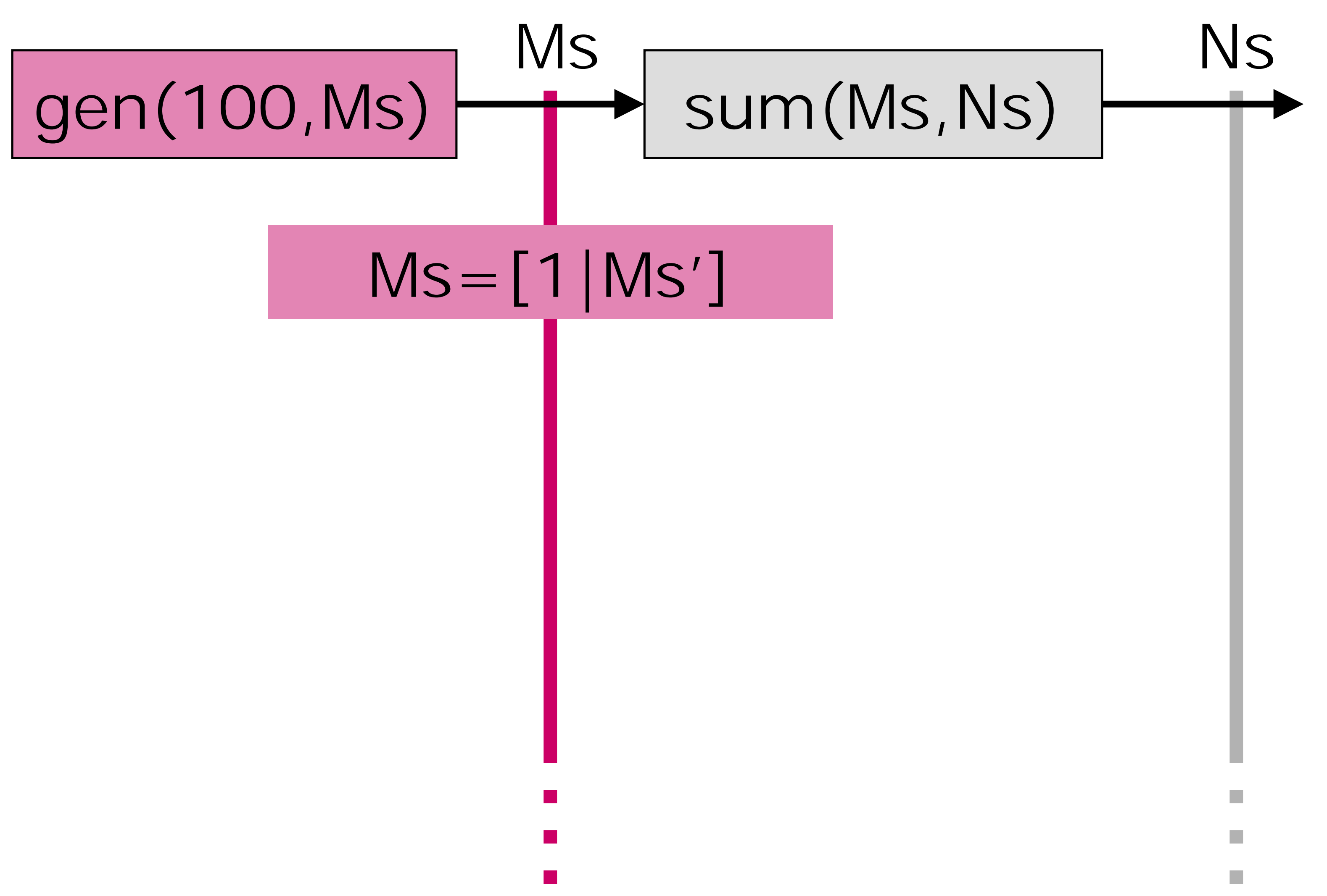
データ指向ストリーム計算

- プロセスとその接続関係を定義するのが並行論理プログラミング。
- プロセスは、多くの場合、ストリームを介して互いに通信する。
- ストリームは、並行オブジェクト指向言語のメッセージキューにほぼ対応。
- 単純なデータ駆動プログラムは、ストリームの生成プロセス、変換プロセス、消費プロセスからなる。

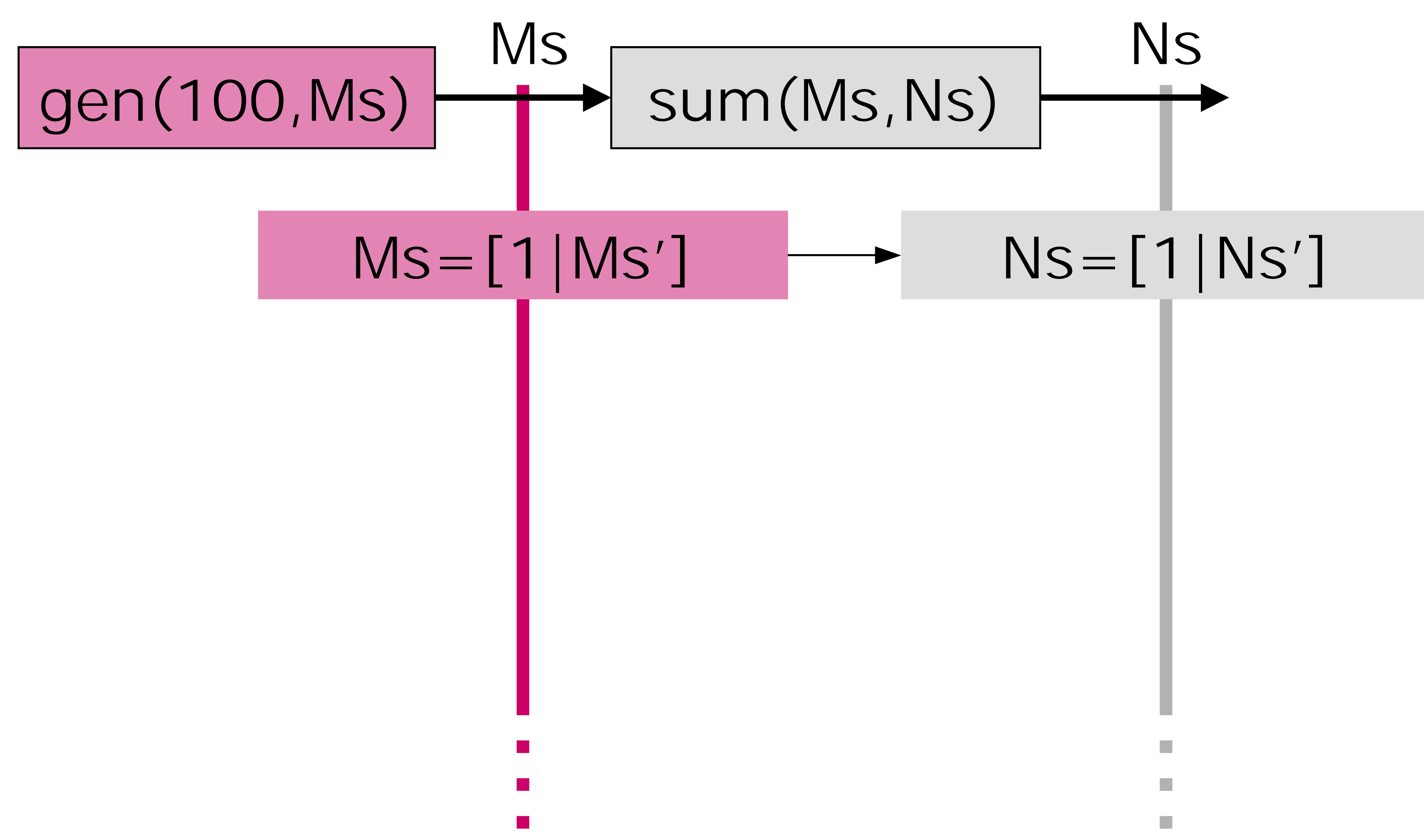
Prefix Sum



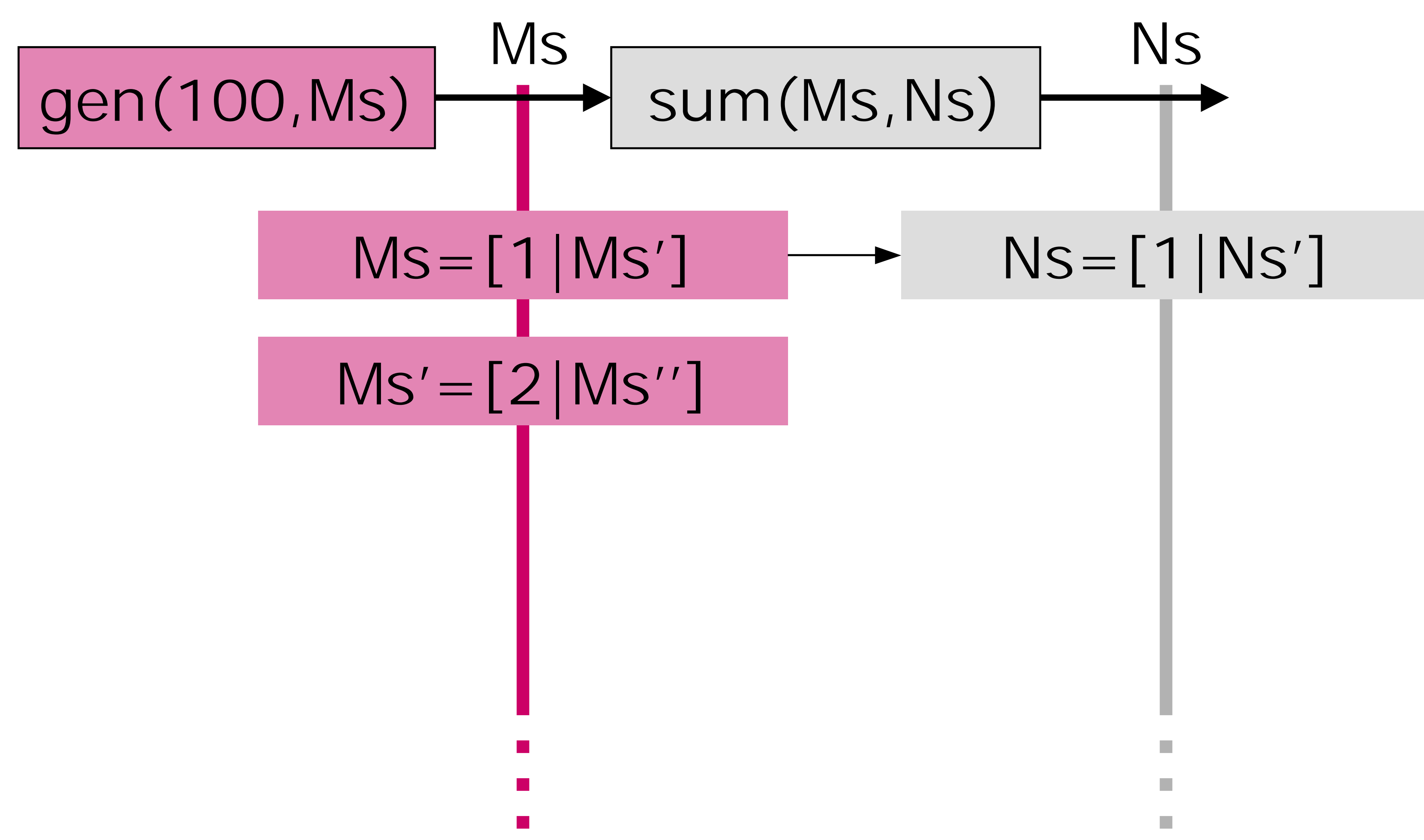
Prefix Sum



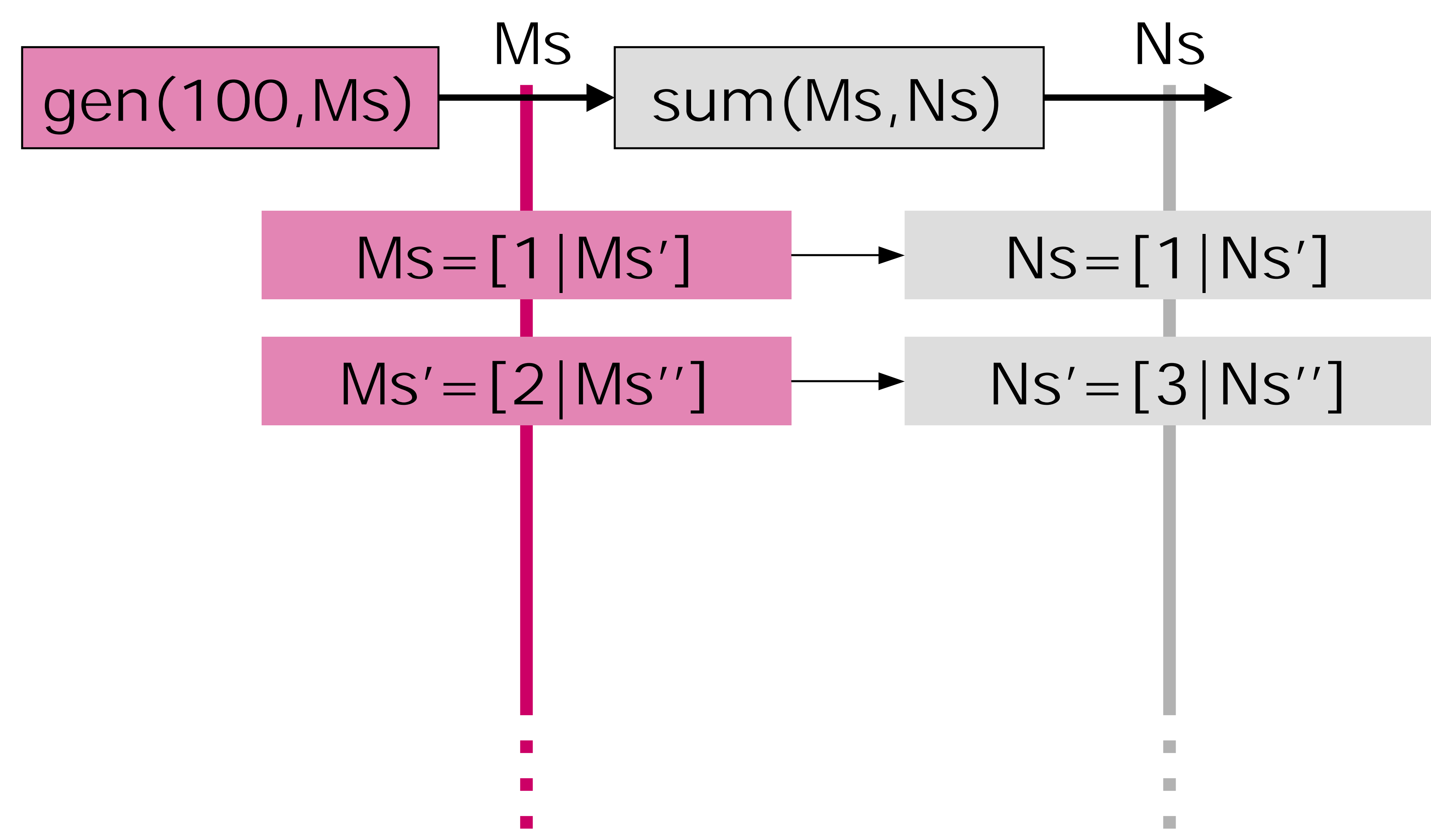
Prefix Sum



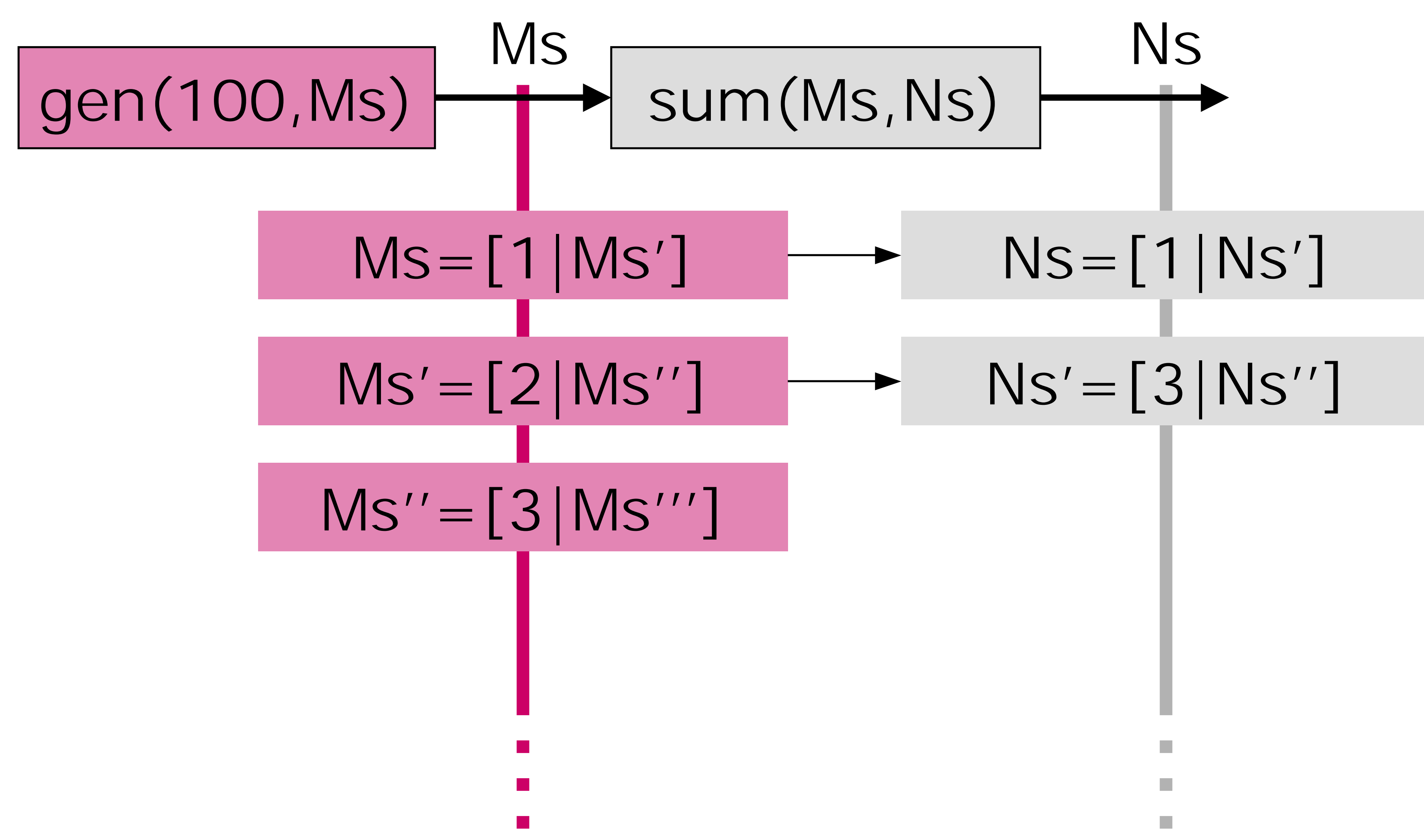
Prefix Sum



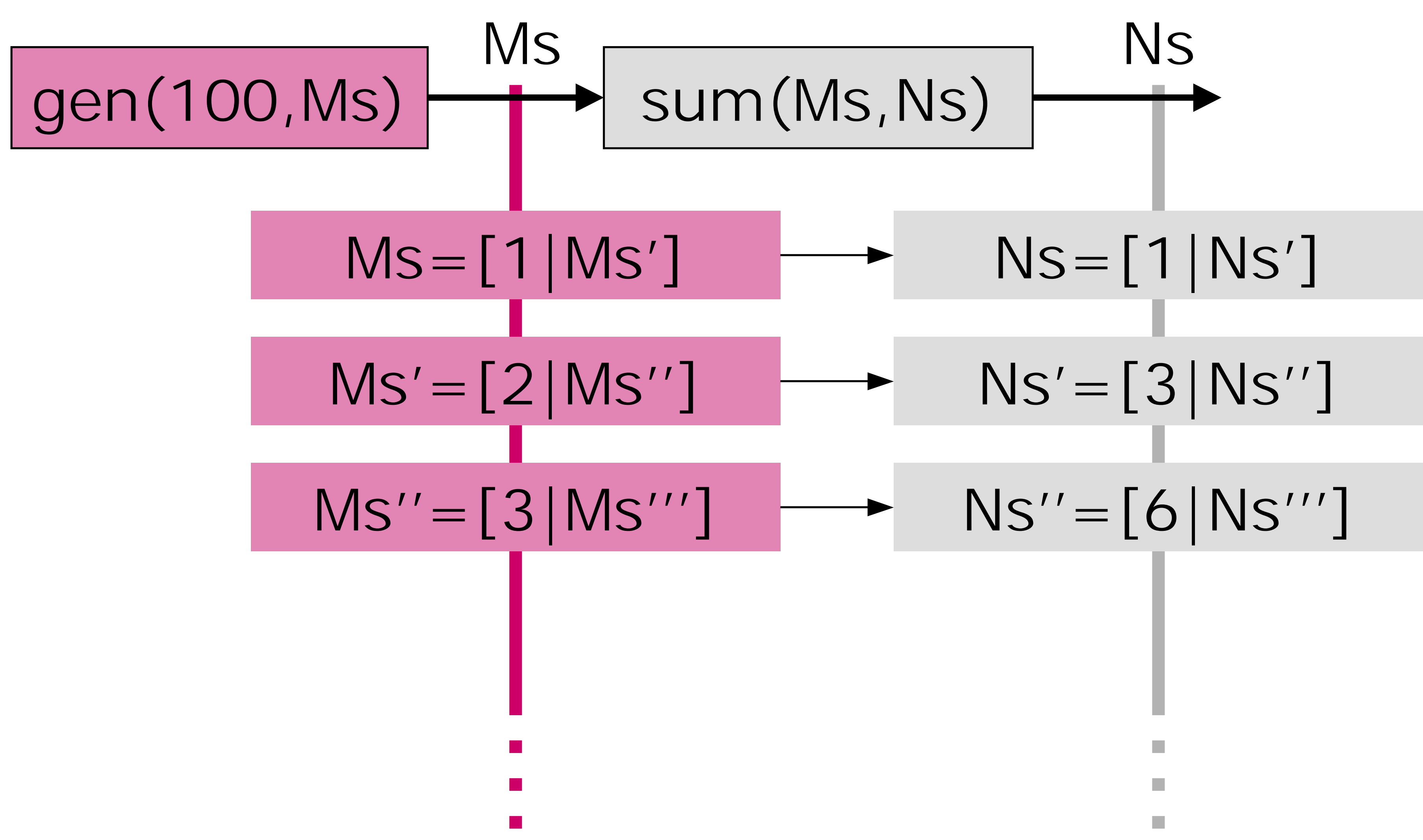
Prefix Sum



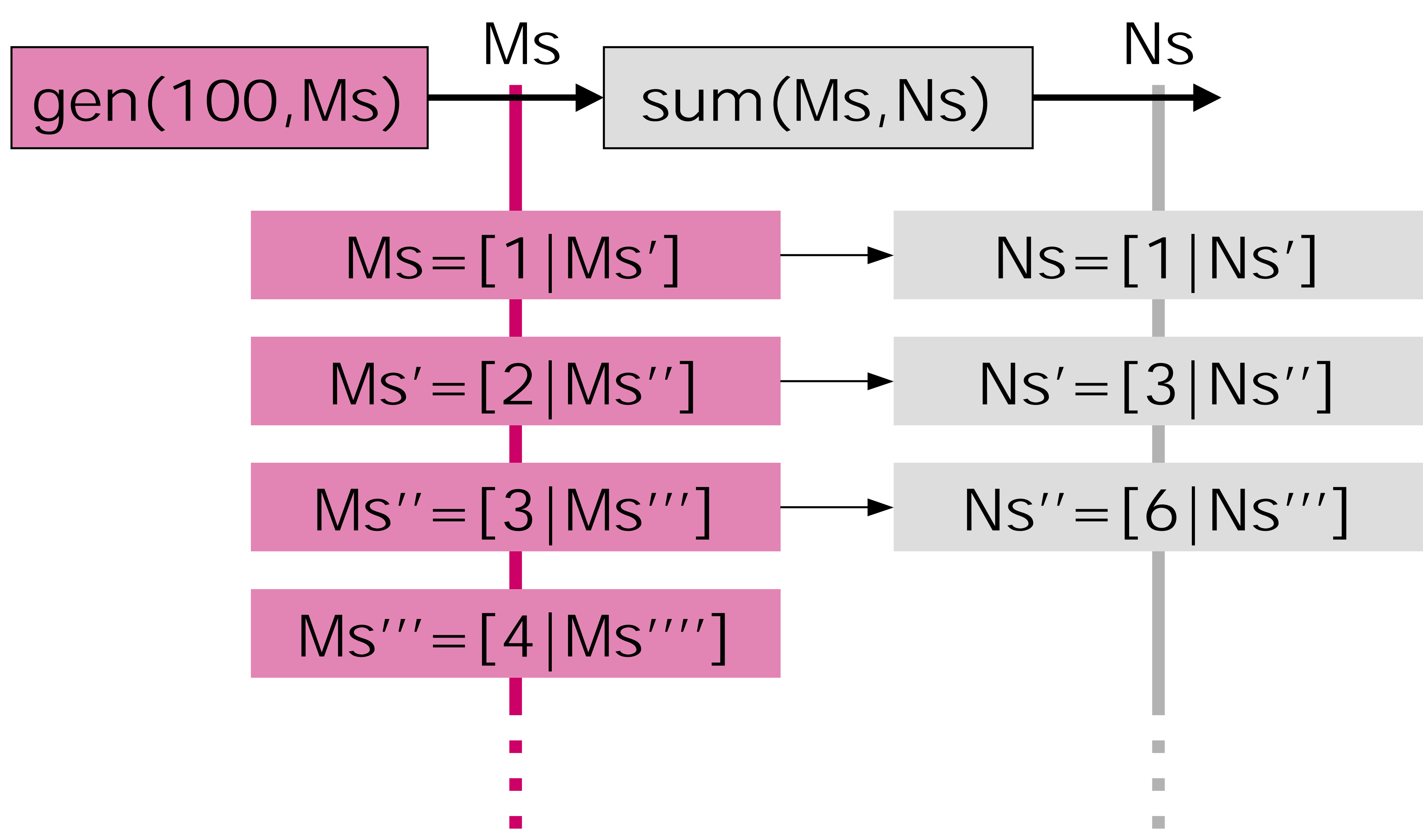
Prefix Sum



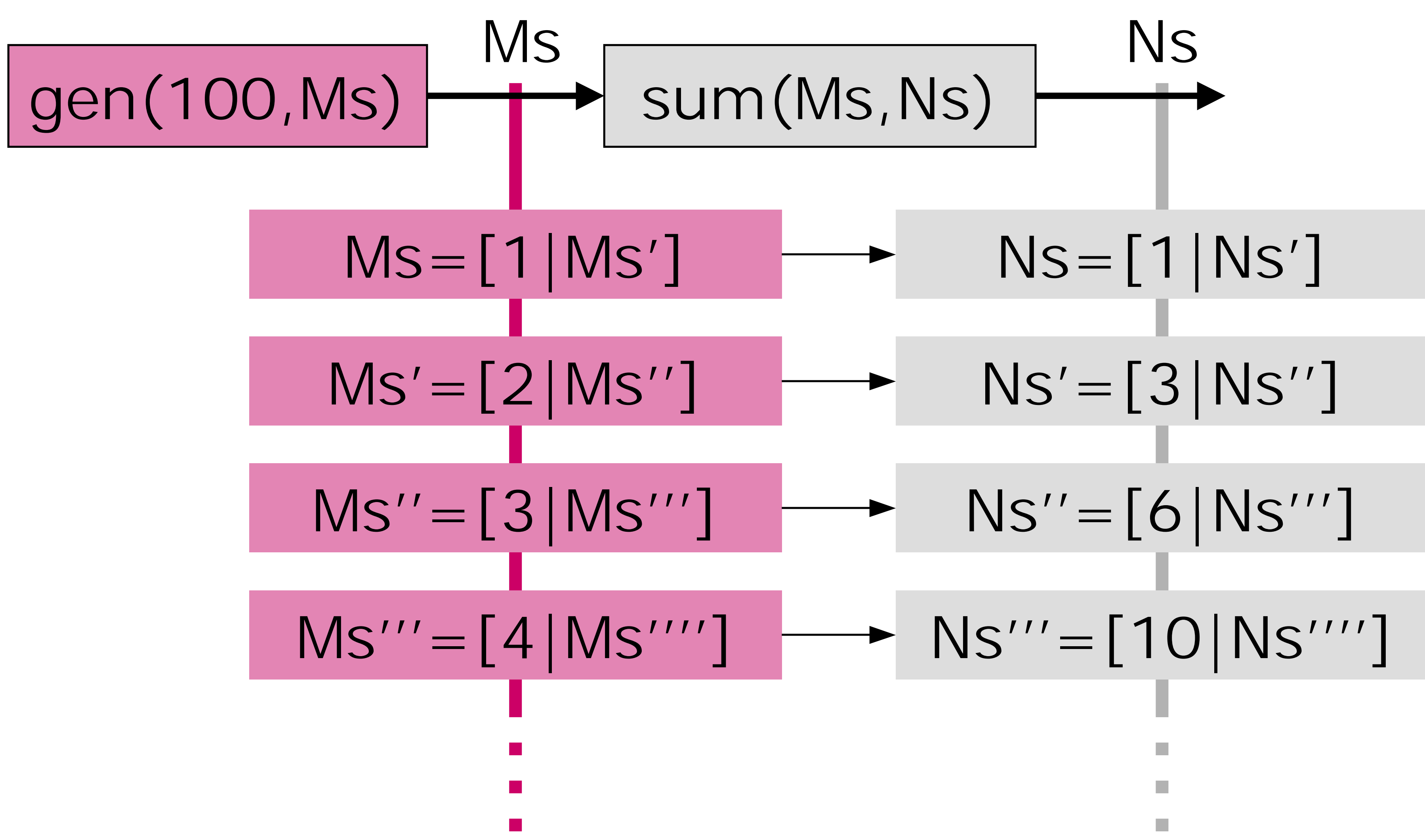
Prefix Sum



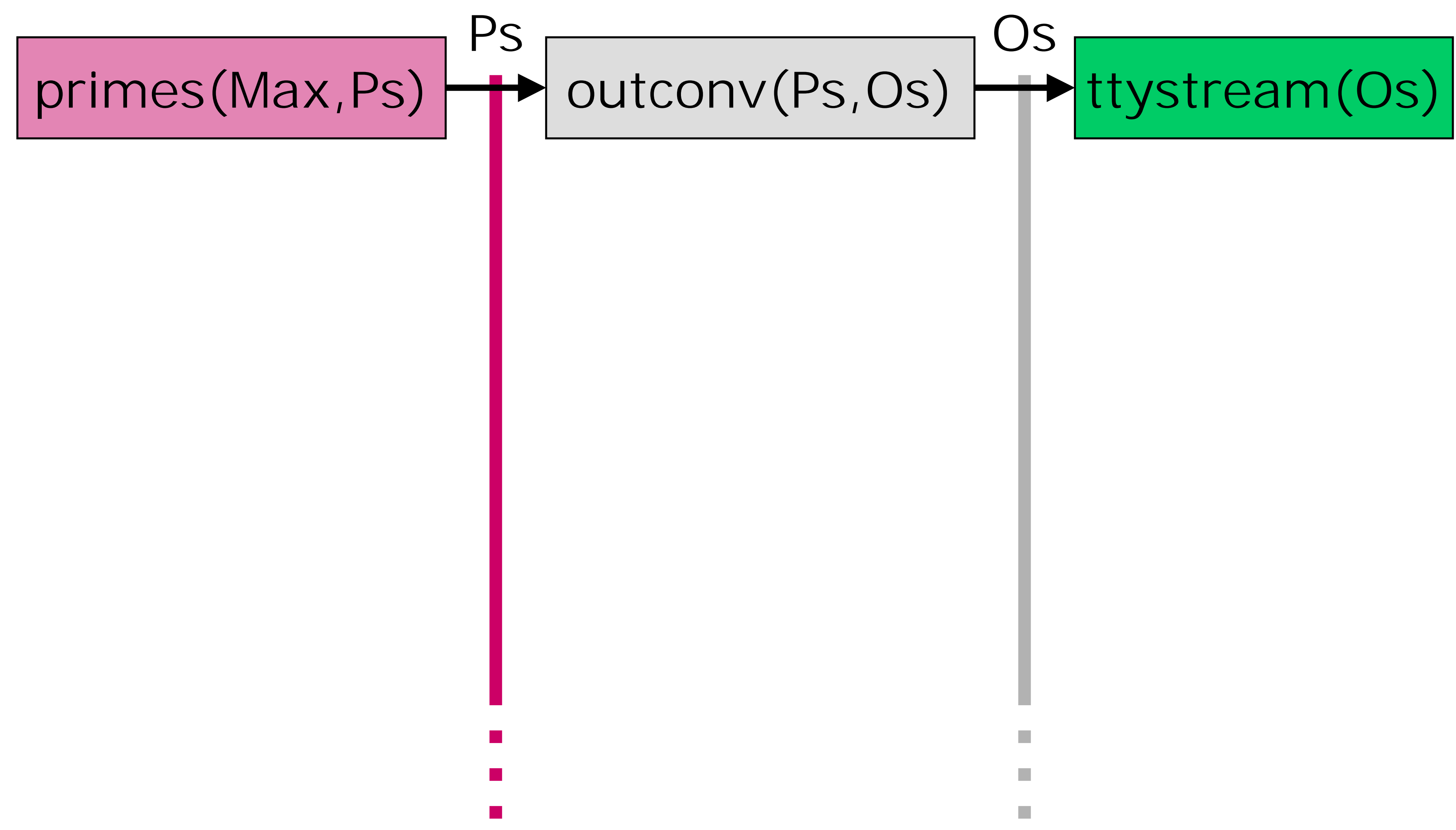
Prefix Sum



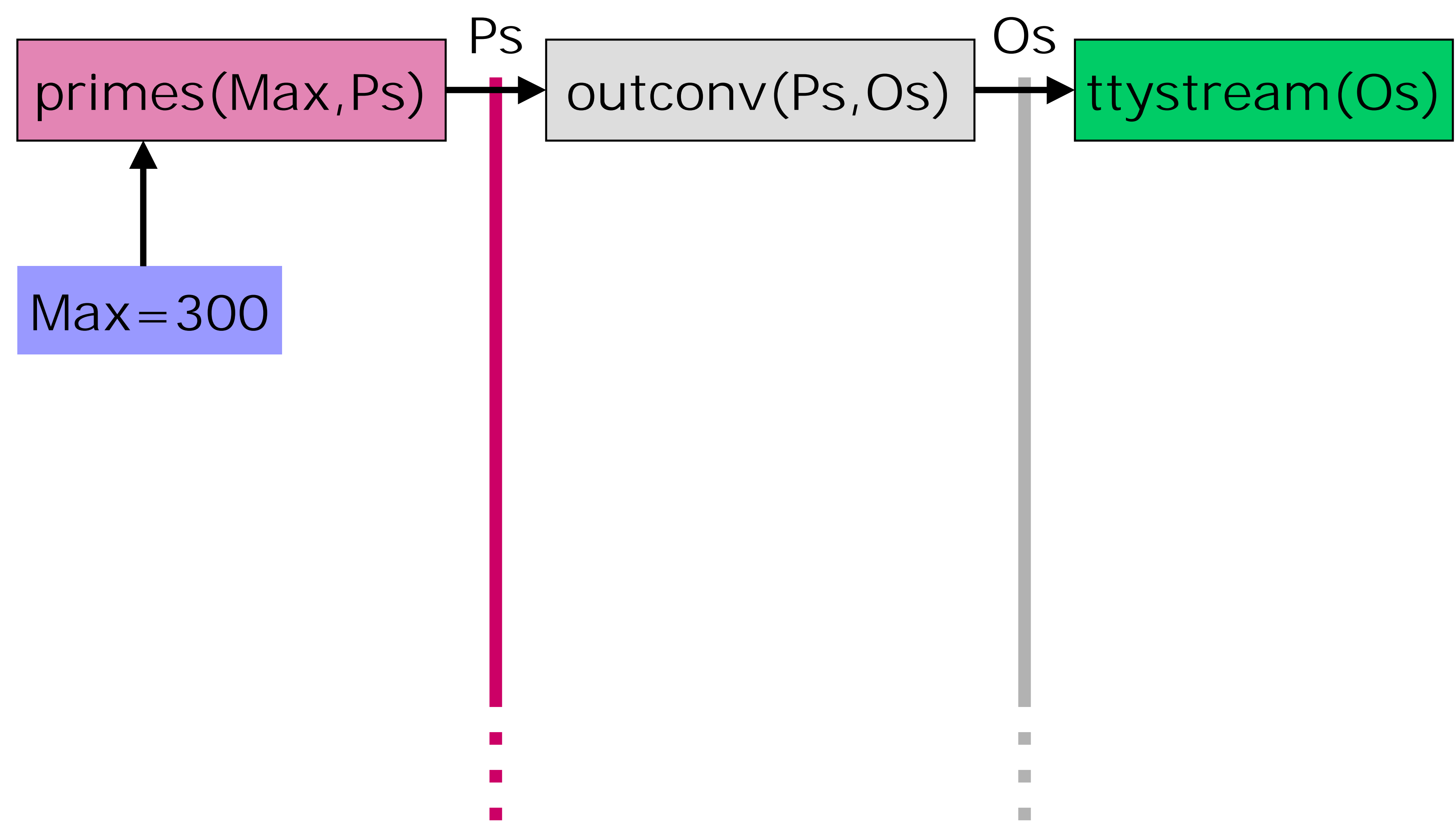
Prefix Sum



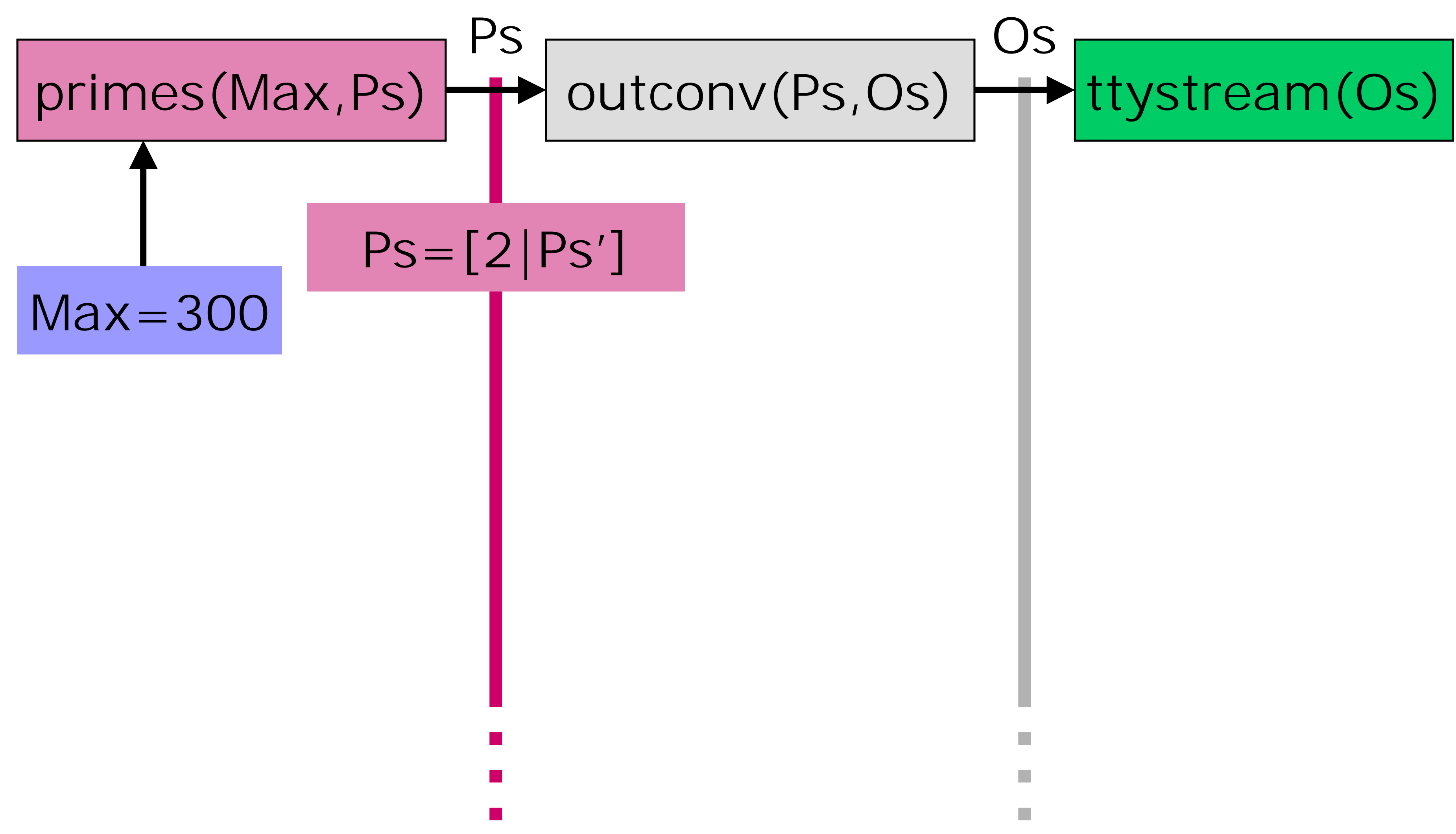
Printing Prime Numbers



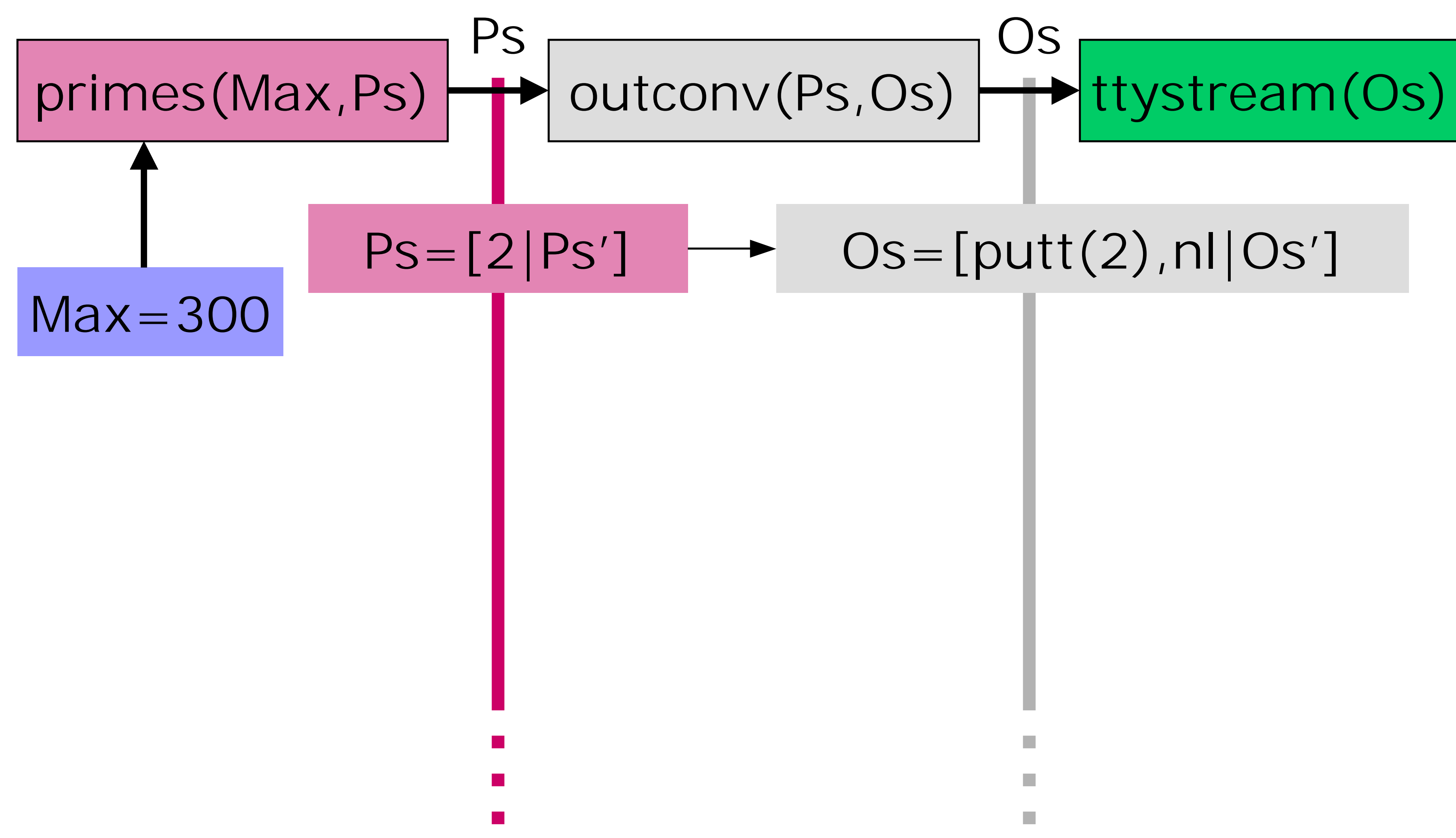
Printing Prime Numbers



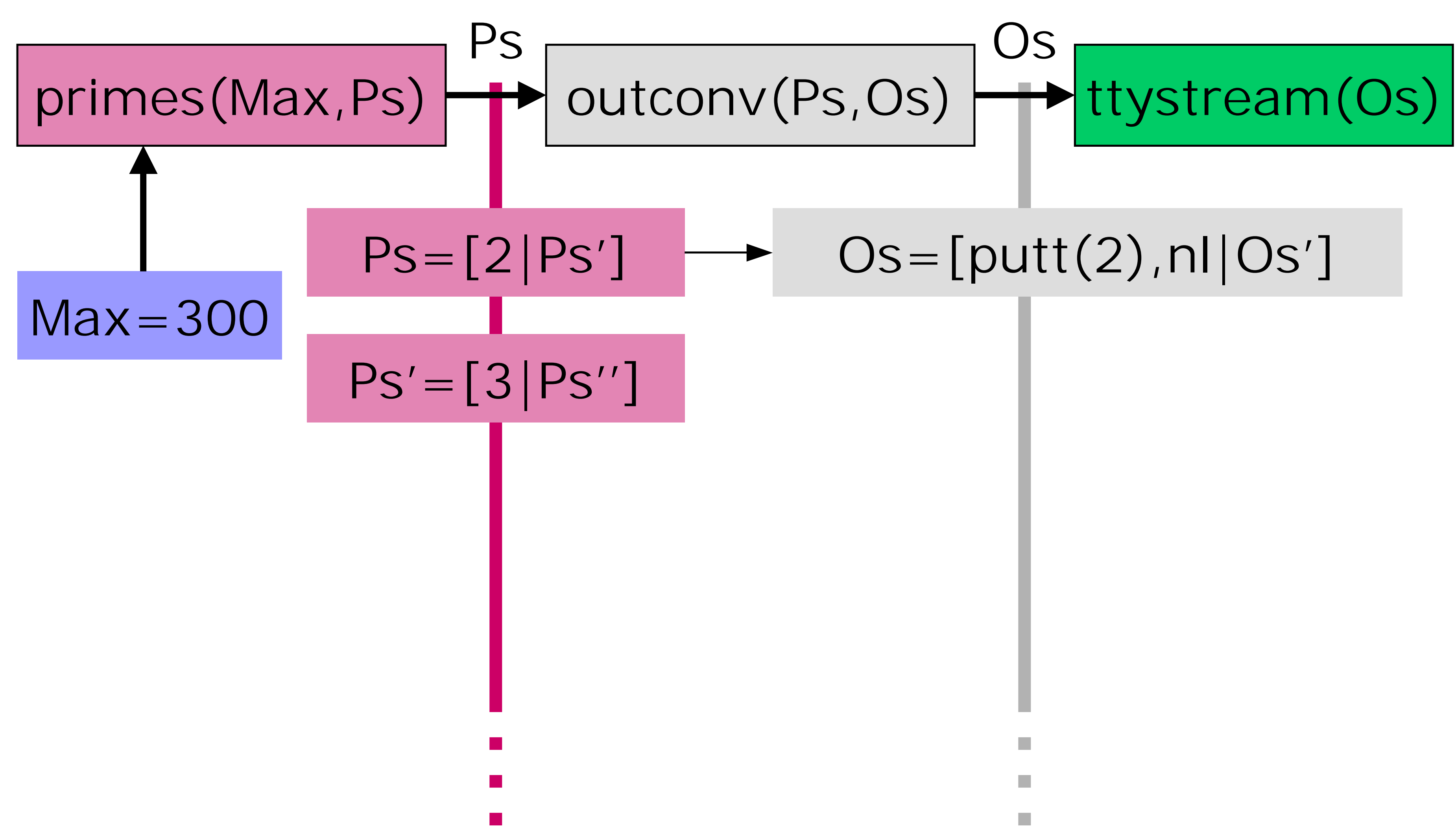
Printing Prime Numbers



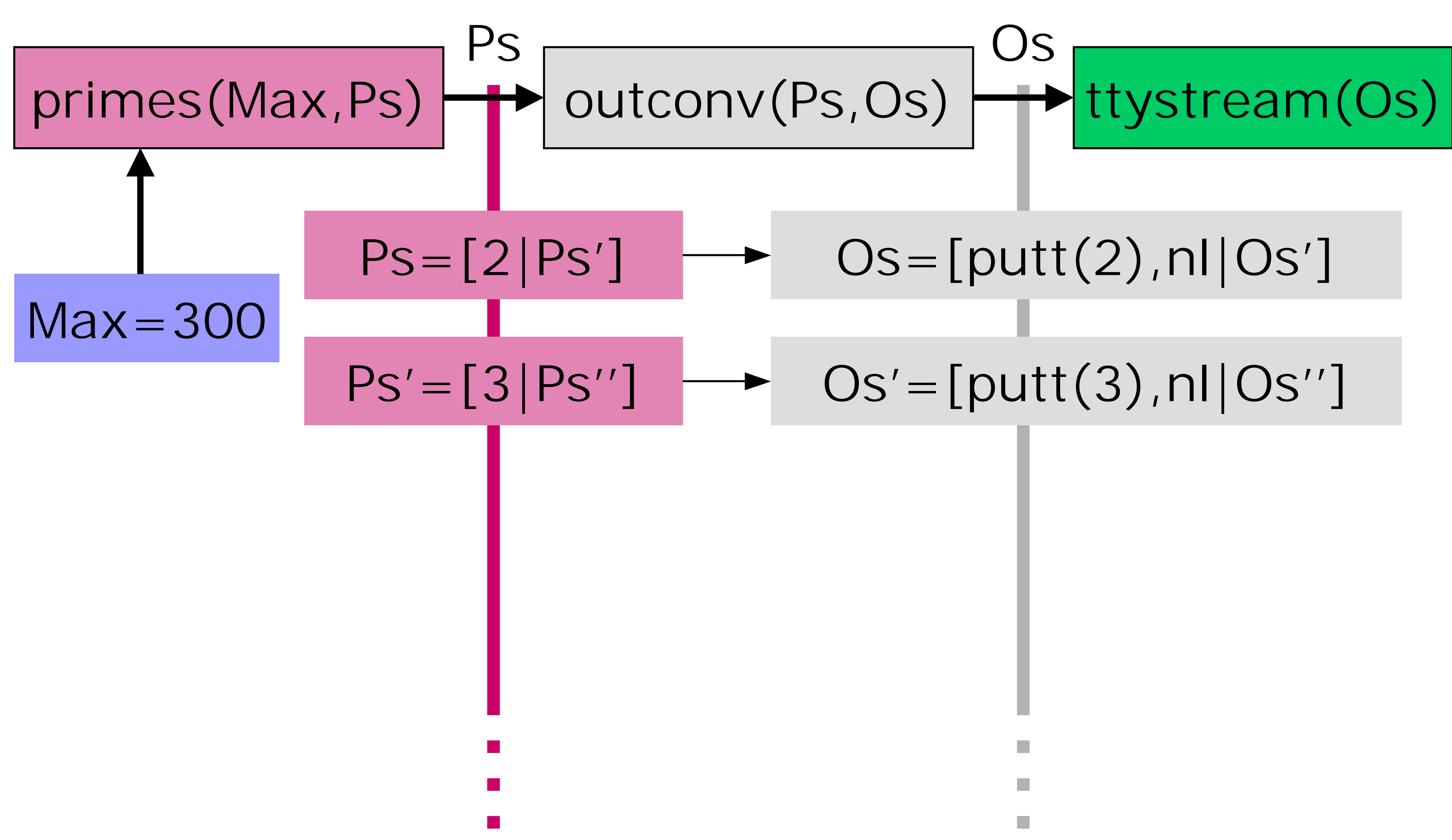
Printing Prime Numbers



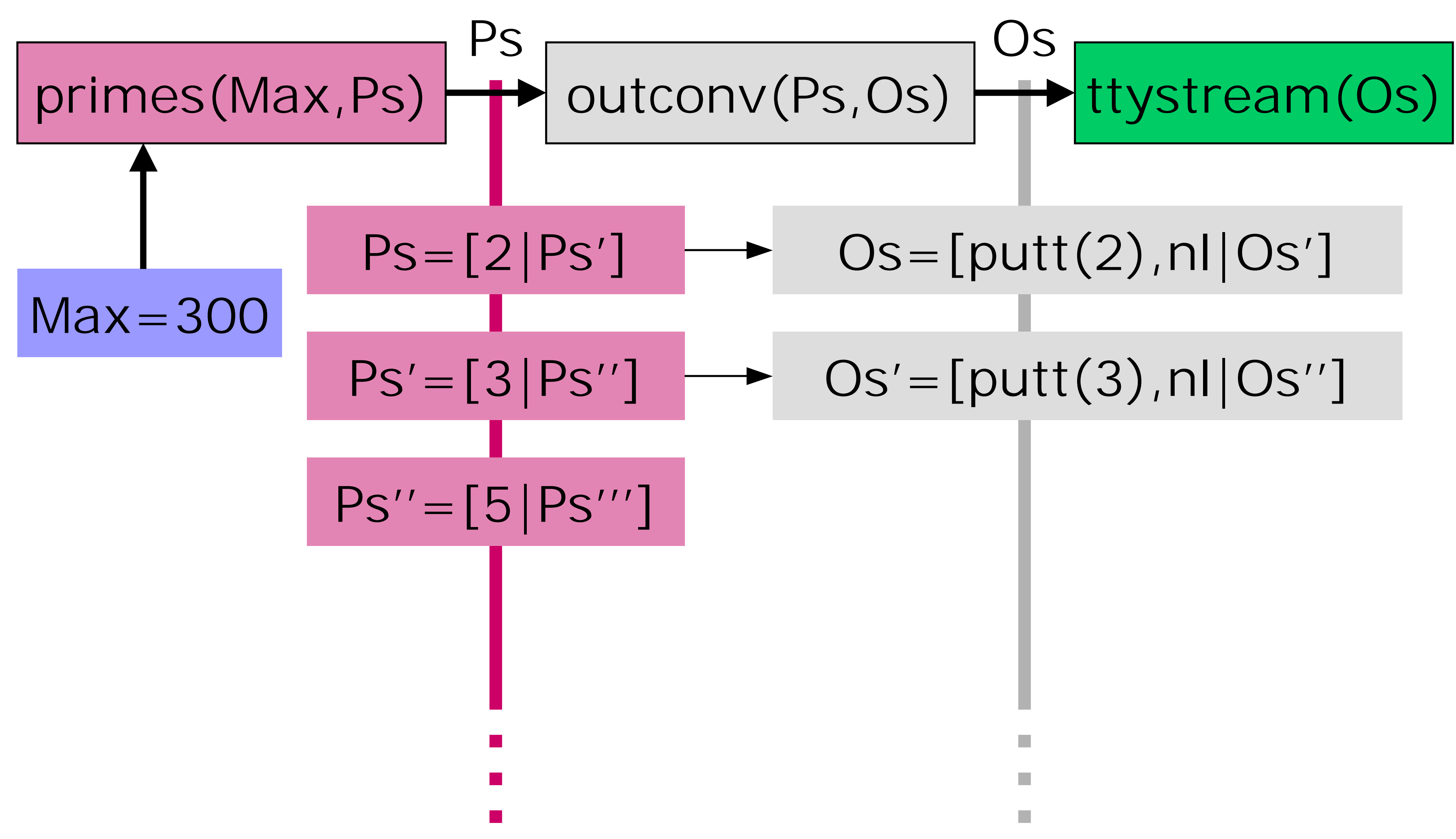
Printing Prime Numbers



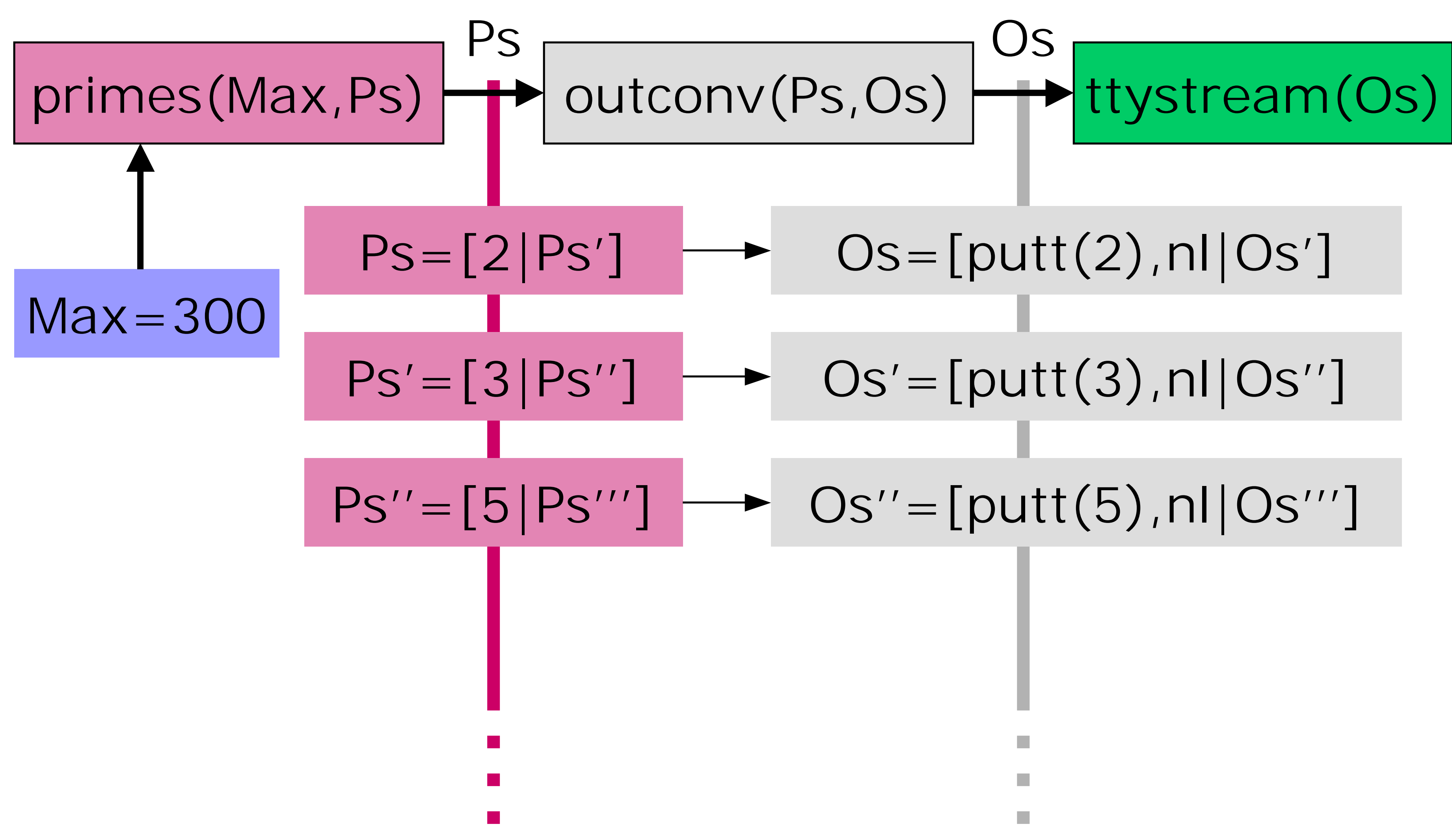
Printing Prime Numbers



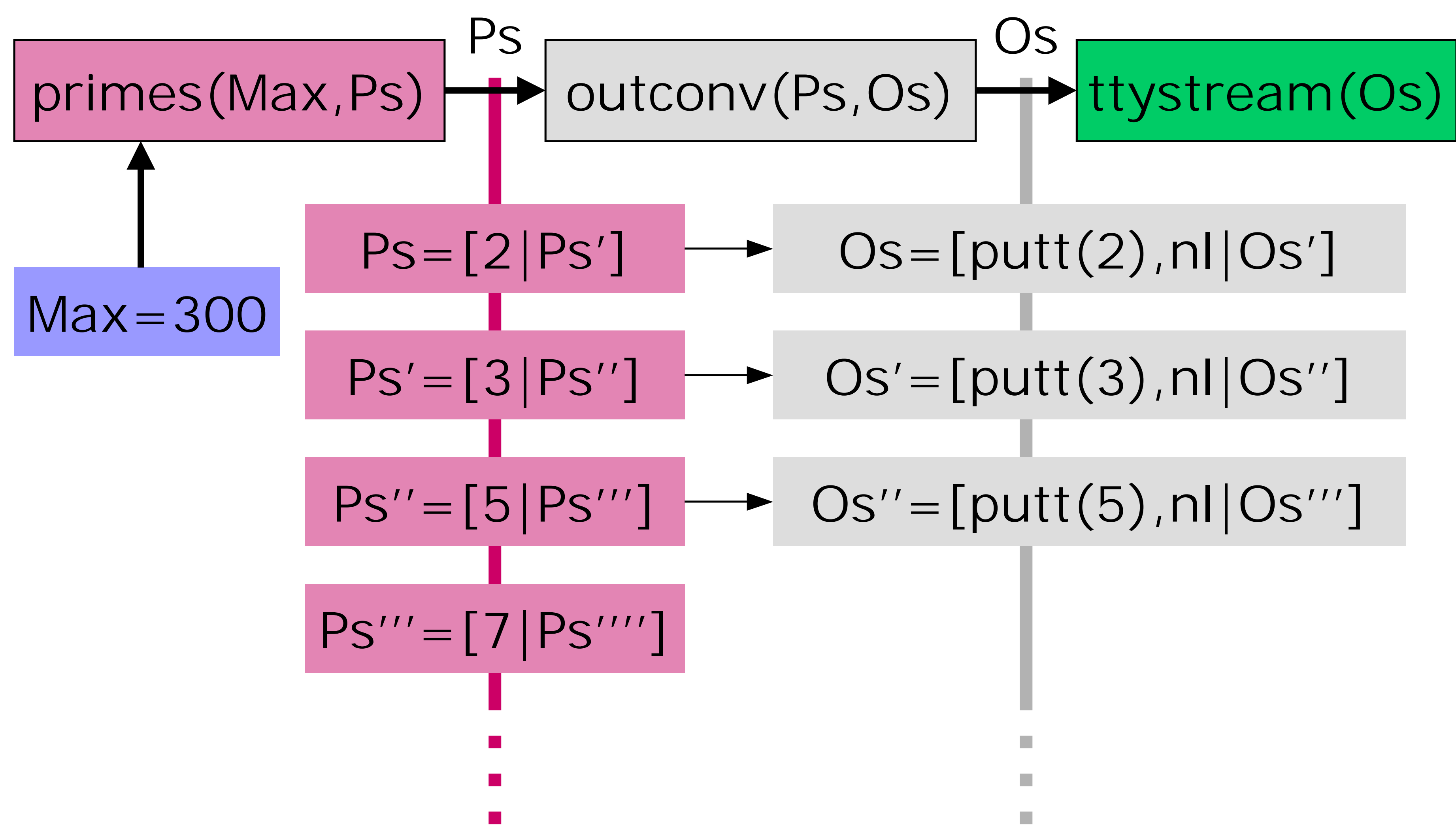
Printing Prime Numbers



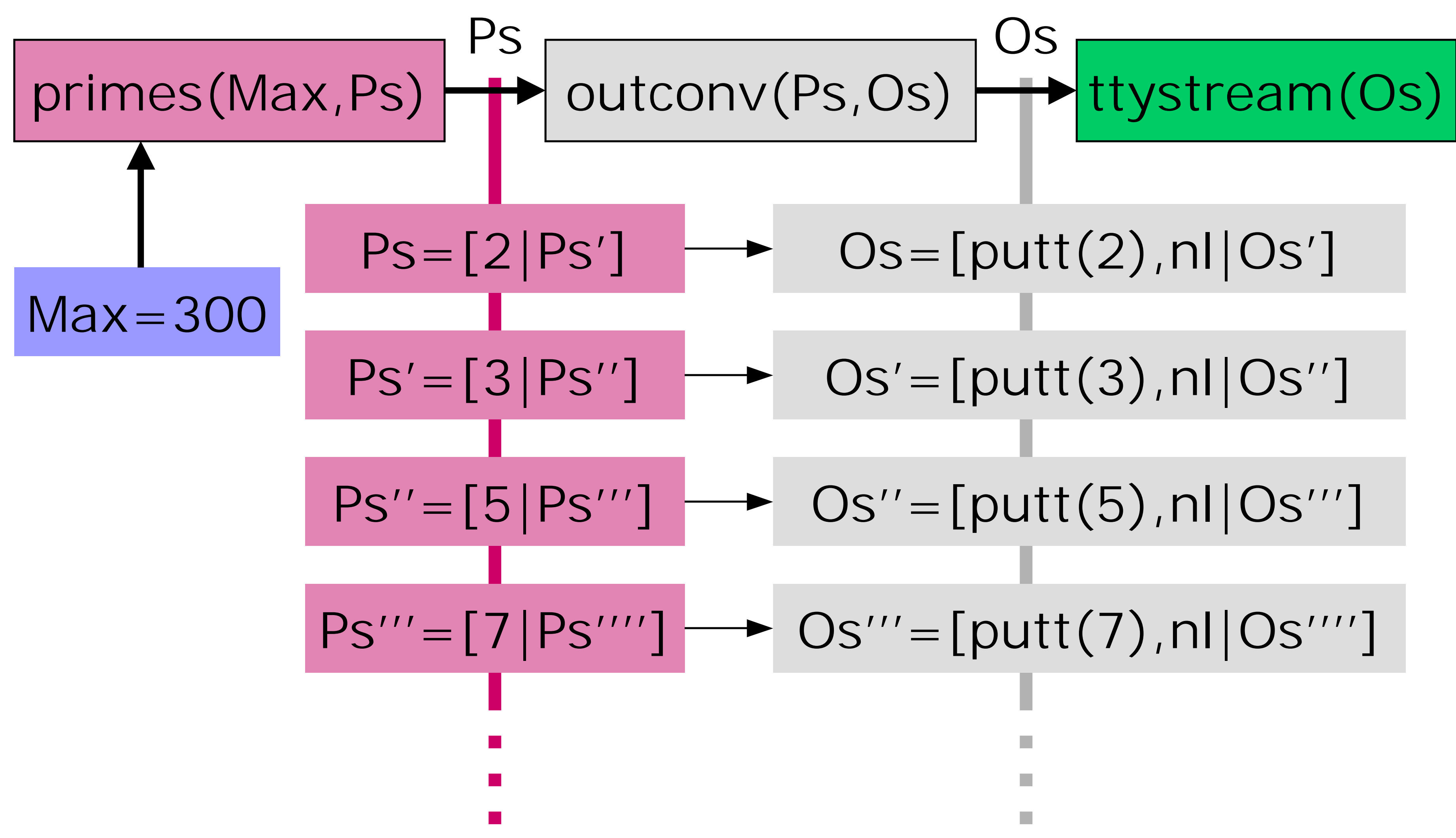
Printing Prime Numbers



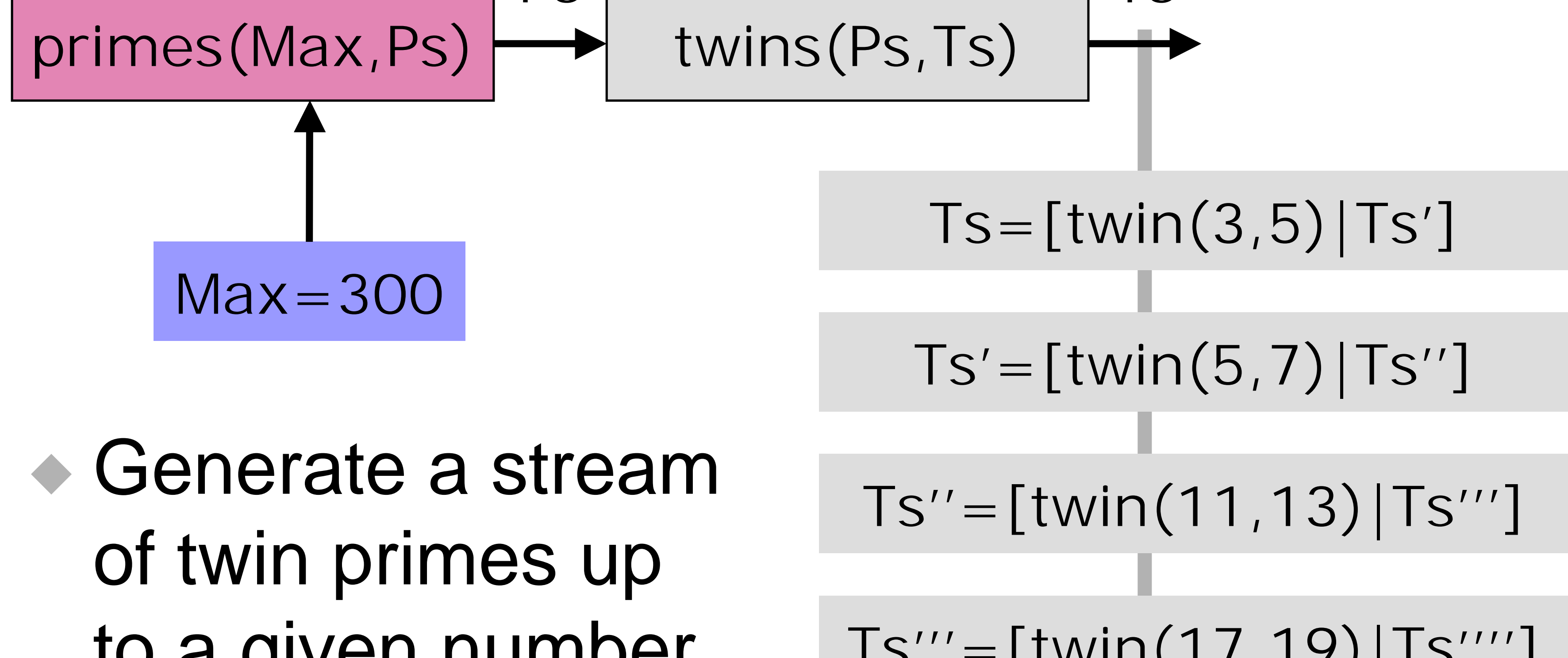
Printing Prime Numbers



Printing Prime Numbers



Twin Primes



- Generate a stream of twin primes up to a given number. (cf. differential programming)

Hamming's Problem

- Generate an ascending sequence of natural numbers (≤ 1000) of the form $2^l \times 3^m \times 5^n$ ($l, m, n \geq 0$) . Do not use integer division.

計算モデルとしてのGHC

- 計算の目的は外界との通信
- 計算の主体はプロセスの集まり
- プロセスは情報（制約）の観測 (ask) と生成 (tell) によって通信する
- 情報とは、変数値のとりうる値への制約
- 情報（制約）は共有変数を通じて伝達
- 同期とは変数値が（判断に必要なだけ）具体化するまで待つこと
  – 送り手は待たされない（非同期通信）

計算モデルとしてのGHC

- プロセスは、ゴールの集合として表現
- ゴールの挙動は書換え規則によって定義
- 外界も一つのプロセス
- プロセスには決定的なものと非決定的なものとがある

Prolog vs. GHC

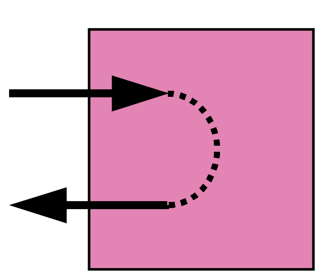
	Prolog (LP)	GHC (Concurrent LP)
制御	ゴールの順序付け	データフロー
	逐次（＋並行）	並行
探索	あり	なし
双方向性	あり	なし

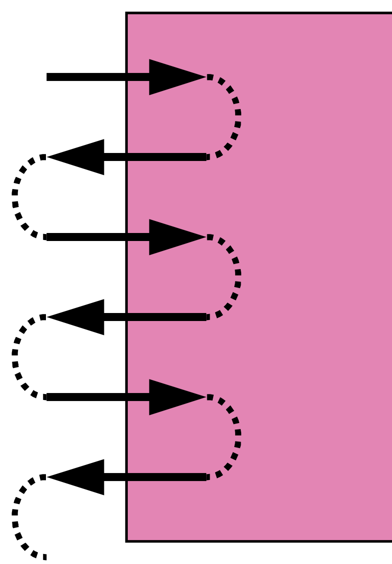
Tranformational vs. Reactive

- 関数型言語は、deterministic かつ transformational な言語
- Prolog は、nondeterministic な探索機能をもつ transformational な言語
- GHC / KL1 は並行処理機能 と indeterminacy をもつ reactive な言語

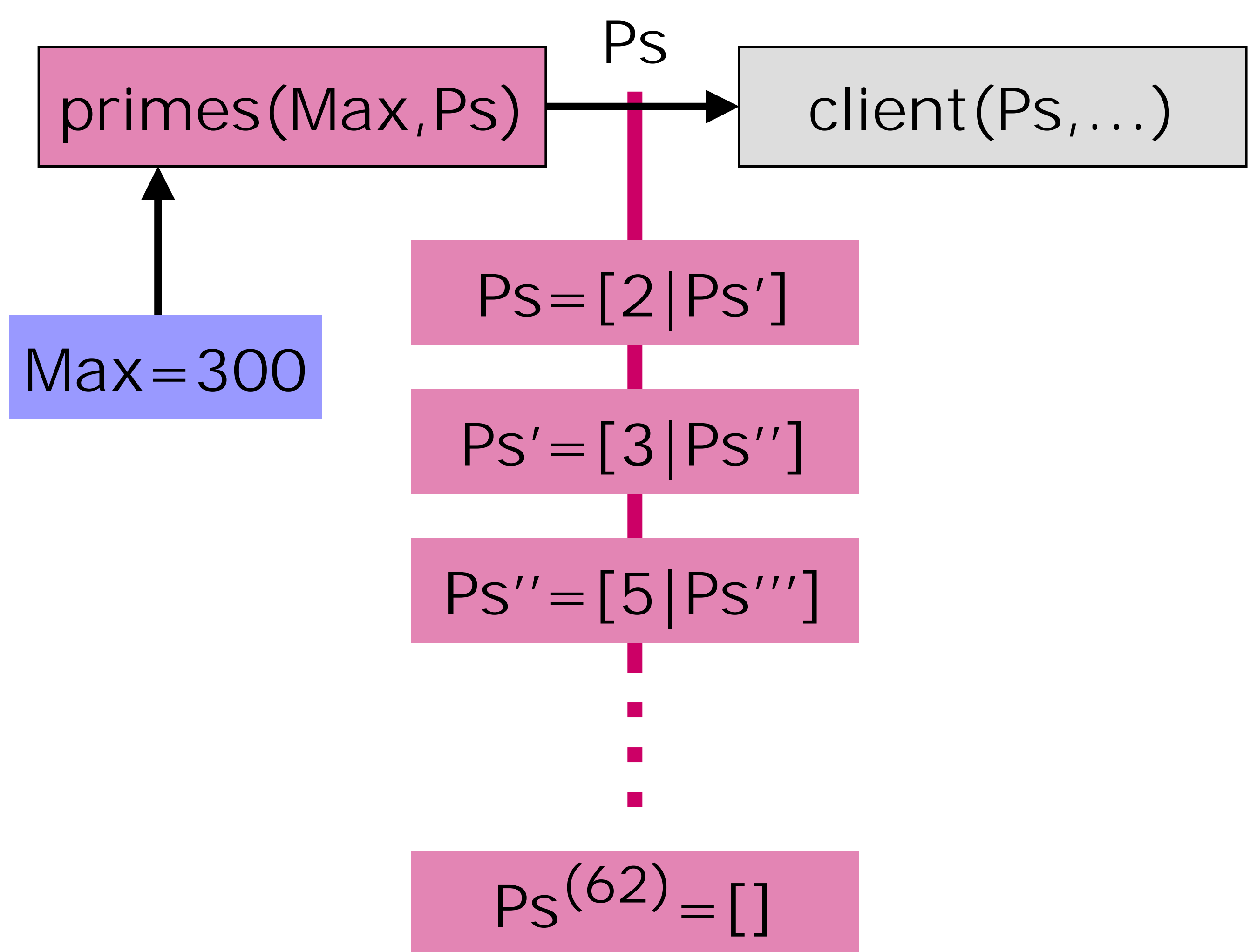
Tranformational vs. Reactive

- Transformational — one transaction
- Reactive — possibly many tranactions
  - input may depend on output

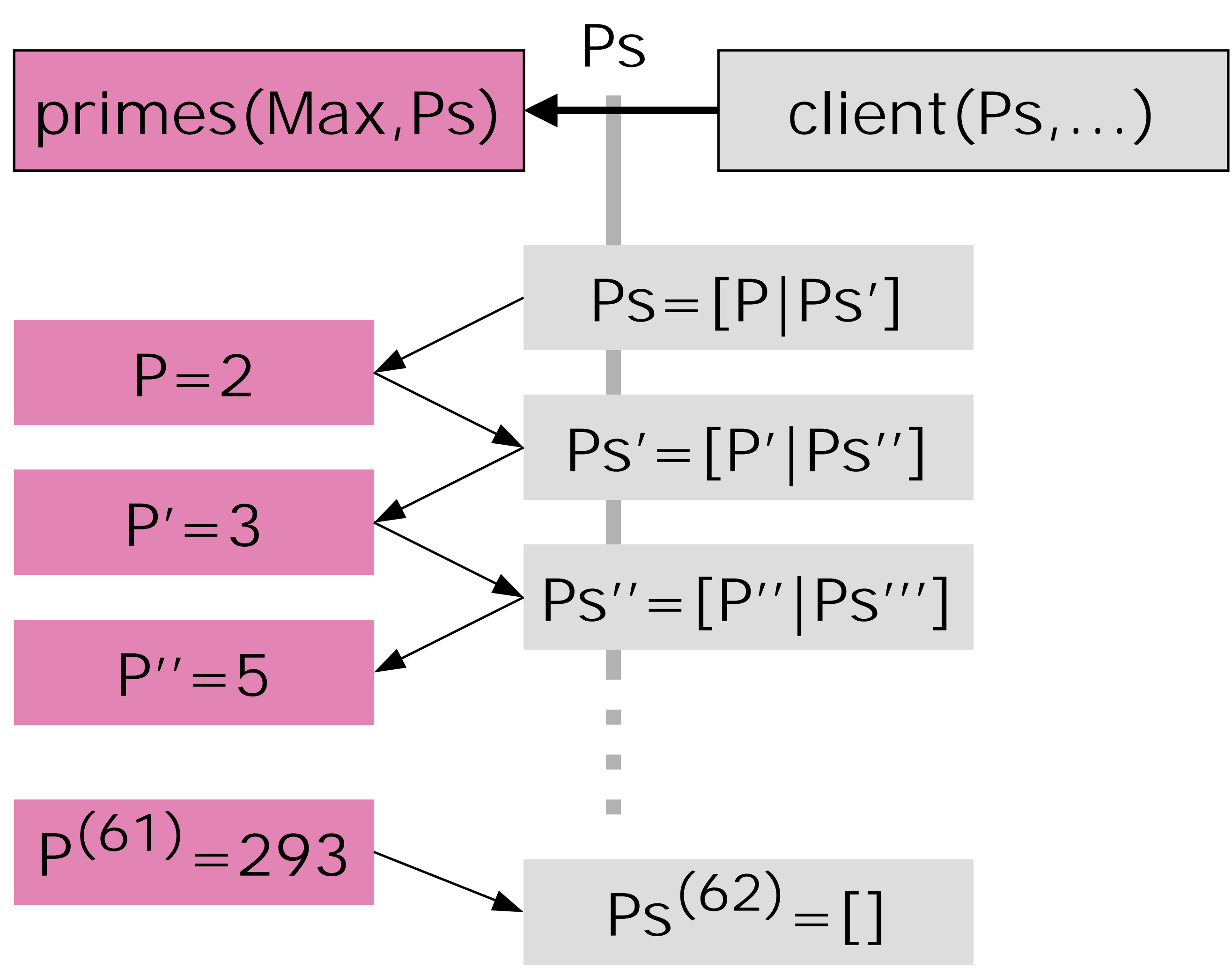




データ駆動計算



要求駆動計算



要求駆動計算

- 要求駆動計算では、ストリームの骨格は要素の消費者が生成して生産者に送る
- データ駆動と要求駆動の利害得失
- 例：フィボナッチ数列 [0,1,1,2,3,5,8, ...] の要求駆動生成

```
fiblazy(Ns) :- true | fiblazy(    ,     ,Ns).
fiblazy(_,  _,  []) :- true | true.
fiblazy(N1,N2,[N3|Ns1]) :- true |
    N3:=N1+N2, fiblazy(N2,N3,Ns1).
```

サーバ

- クライアントからの要求に応じて情報や資源を供給するプロセス。一般に履歴依存性あり
- 例：カウンタプロセス

```
counter(S) :- true | counter(S,0).
counter([],C) :- true | true.
counter([up(N)|S],C0) :- true |
    C:=C0+N, counter(S,C).
counter([reset|S],C) :- true | counter(S,0).
counter([show(V)|S],C) :- true |
    V:=C, counter(S,C).
```

未完成メッセージ

- up(N): N はクライアントが供給
- show(V): V はサーバが供給してメッセージを「完成」させる
  - １本のストリームを用いた双方向通信

入出力サーバ

◆ サーバ tty: ttystream(S) の挙動

メッセージ	操作
putc(C)	文字 C を出力
nl	改行
putt(T)	項 T を出力
fwrite(S)	文字列 S を出力
getc(C)	文字 C を入力
unget(C)	文字 C を戻す
gett(T)	項 T を入力

多対一通信

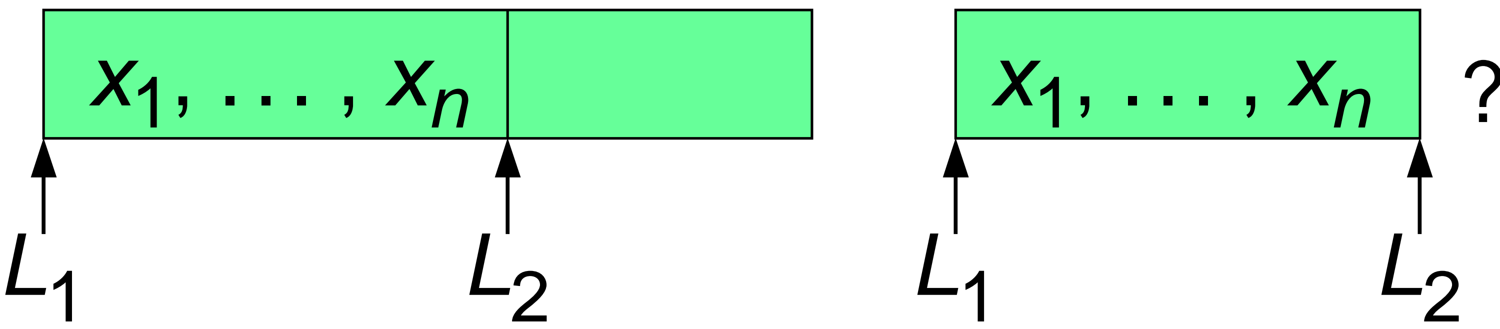
- 多対一のストリーム通信には、ストリーム併合器を用いる

```
merge([],        Ys,       Zs) :- true | Zs=Ys.
merge(Xs,        [],       Zs) :- true | Zs=Xs.
merge([A|Xs],Ys,           Zs0) :- true |
        Zs0=[A|Zs], merge(Xs,Ys,Zs).
merge(Xs,        [A|Ys],Zs0) :- true |
        Zs0=[A|Zs], merge(Xs,Ys,Zs).
```

差分リスト (difference lists)

- 論理変数を利用した強力なプログラミング技法
- リスト L_1 から要素 $x_1, \ldots, x_n$ ($n \geq 0$) を除去するとリスト L_2 が得られるとき、L_1 と L_2 の対を差分リストという

cf. time vs. duration, position vs. displacement



差分リスト (difference lists)

- L_2 が不定ならば、(L_1,L_2) と他の差分リストを定数時間で連結できる
- 差分リスト技法は、リストの各部を任意の順序で（あるいは並列に）作成することを可能にする
- 後で他のリストと連結する予定のあるリストは差分リストとして作っておく
- 差分リストをリストに変換するのは容易。その逆は手間がかかる

Quicksort

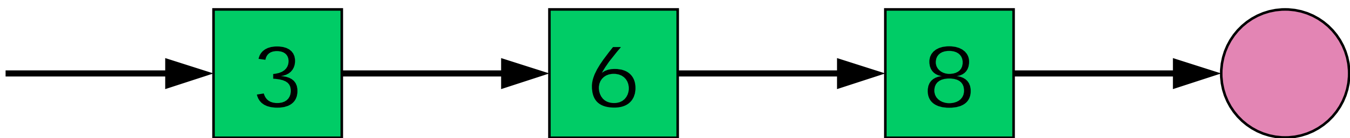
```
qsort(Xs,Ys) :- true | qsort(Xs,Ys,[]).
qsort([],      Ys0,Ys  ) :- true | Ys=Ys0.
qsort([X|Xs],Ys0,Ys3) :- true |
    part(X,Xs,S,L),
    qsort(S,Ys0,[X|Ys2]), qsort(L,Ys2,Ys3).

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).
```

動的プロセス構造

- 例：疎な集合のリスト表現（cf. Hoare 1978）

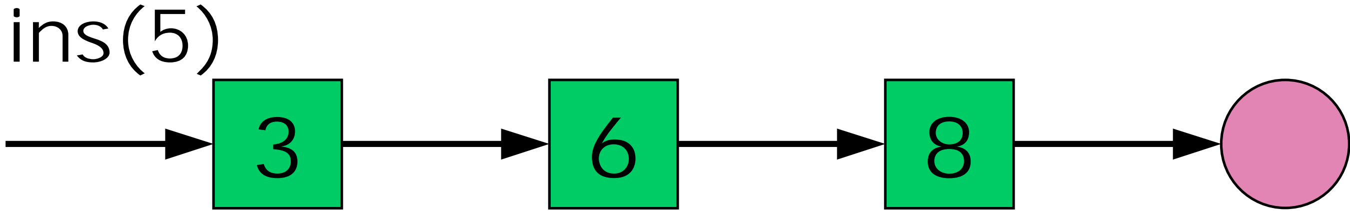
動的プロセス構造

- 例：疎な集合のリスト表現（cf. Hoare 1978）



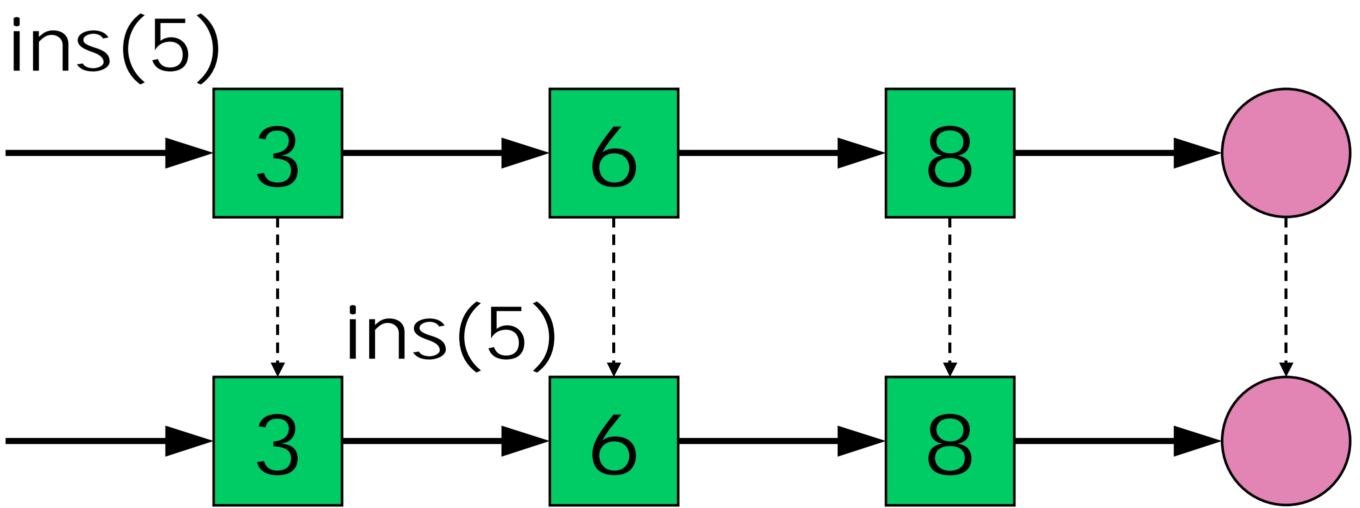
動的プロセス構造

- 例：疎な集合のリスト表現（cf. Hoare 1978）



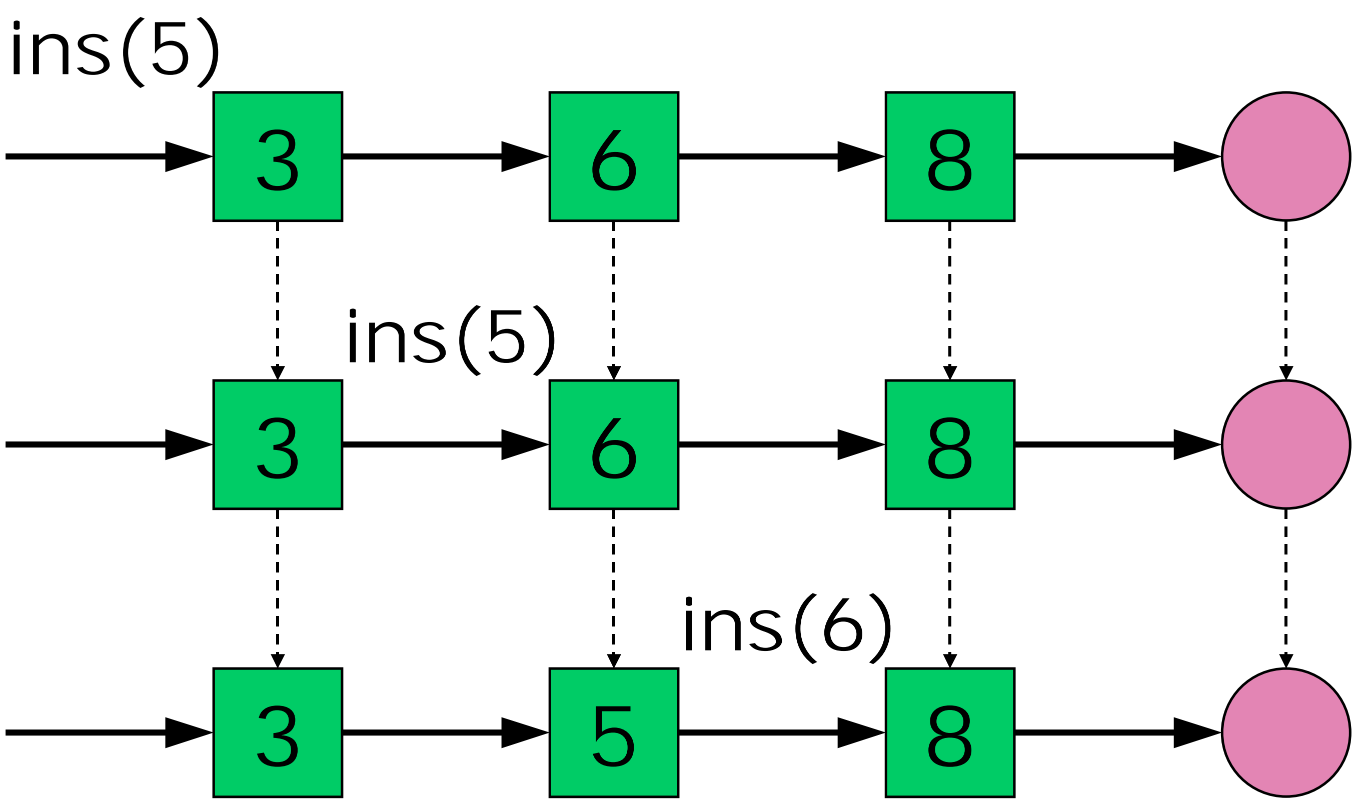
動的プロセス構造

- 例：疎な集合のリスト表現（cf. Hoare 1978）



動的プロセス構造

- 例：疎な集合のリスト表現（cf. Hoare 1978）

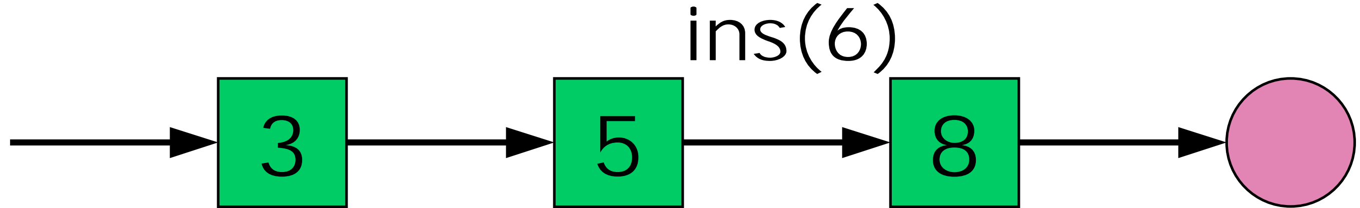


動的プロセス構造

- 例：疎な集合のリスト表現（cf. Hoare 1978）

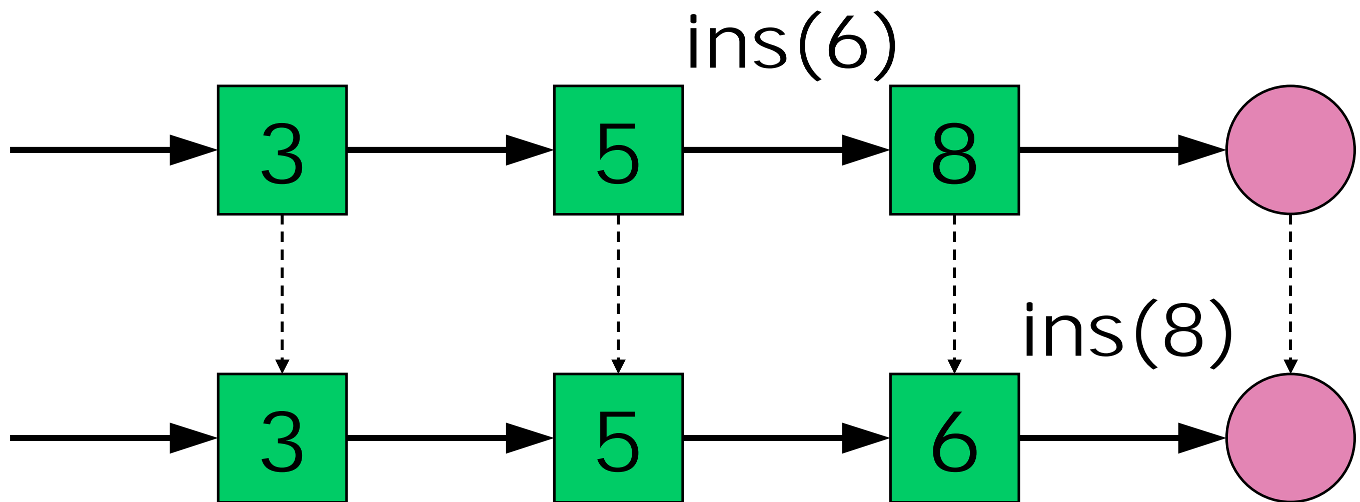
動的プロセス構造

◆ 例：疎な集合のリスト表現（cf. Hoare 1978）



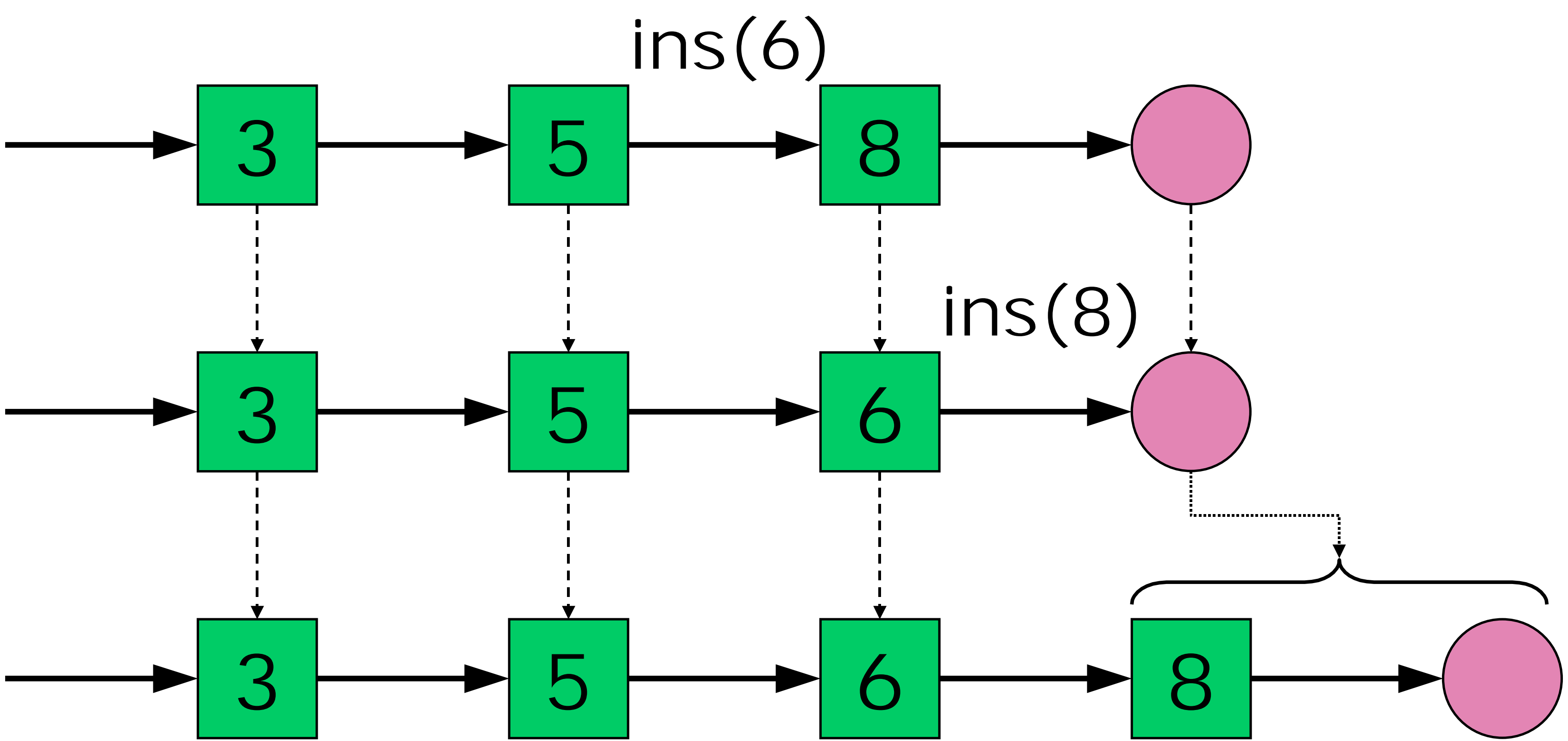
動的プロセス構造

- 例：疎な集合のリスト表現（cf. Hoare 1978）



動的プロセス構造

- 例：疎な集合のリスト表現（cf. Hoare 1978）



Element process: *n*

```
elem([],R,N) :- true | R=[ ].
elem([ins(M)|L],R0,N) :- M < N |
                    R0=[ins(N)|R],     elem(L,R,M).
elem([ins(M)|L],R,  N) :- M=:=N | elem(L,R,N).
elem([ins(M)|L],R0,N) :- M > N |
                    R0=[ins(M)|R],     elem(L,R,N).
elem([has(M,A)|L],R,  N) :- M < N |
                    A=false,           elem(L,R,N).
elem([has(M,A)|L],R,  N) :- M=:=N |
                    A=true,            elem(L,R,N).
elem([has(M,A)|L],R0,N) :- M > N |
                    R0=[has(M,A)|R], elem(L,R,N).
```

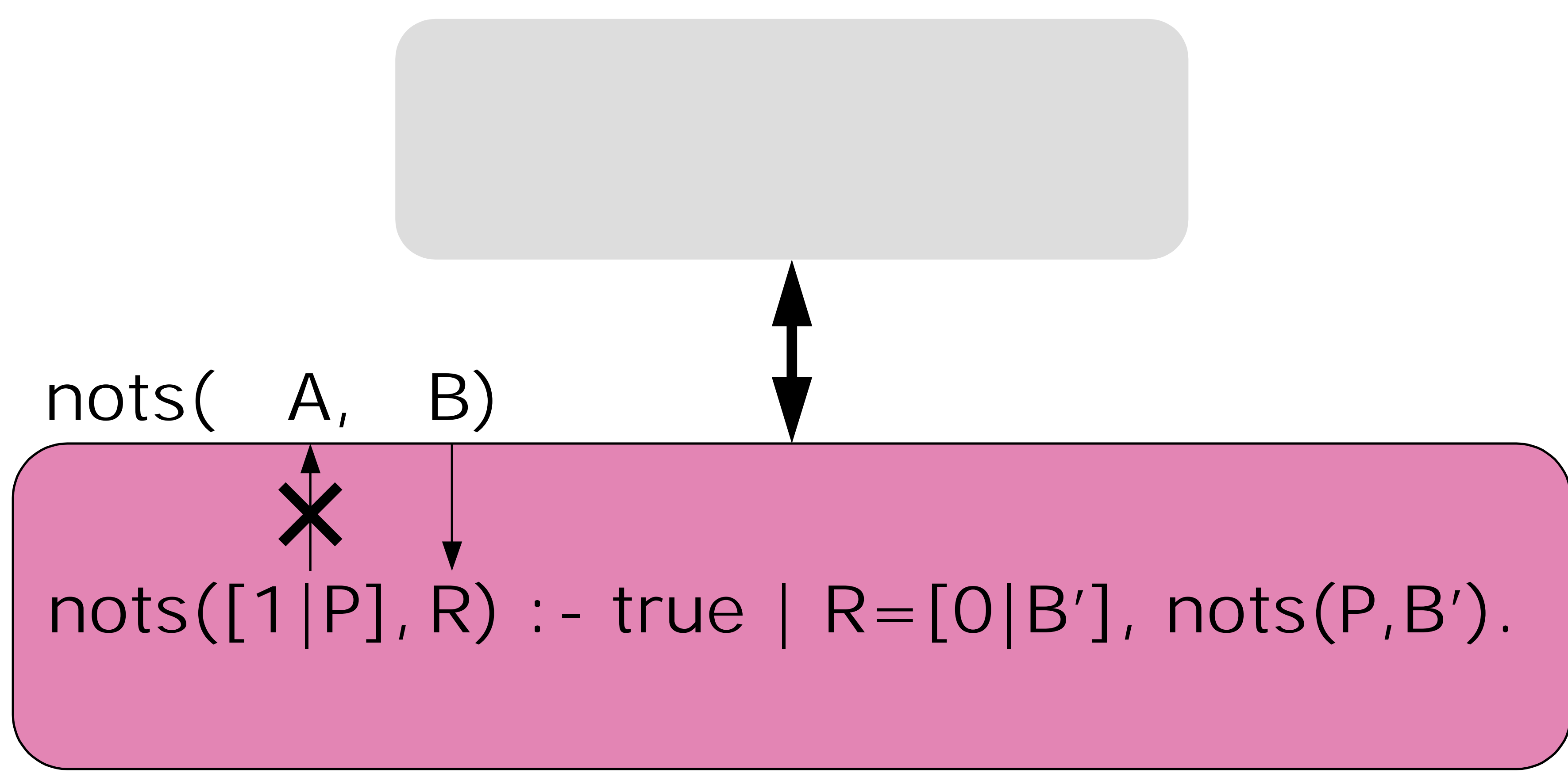
Terminator process:

```
emptyset([]) :- true | true.
emptyset([ins(M)|L]    ) :- true |
                       elem(L,R,M), emptyset(R).
emptyset([has(M,A)|L]) :- true |
                       A=false,      emptyset(L).
```

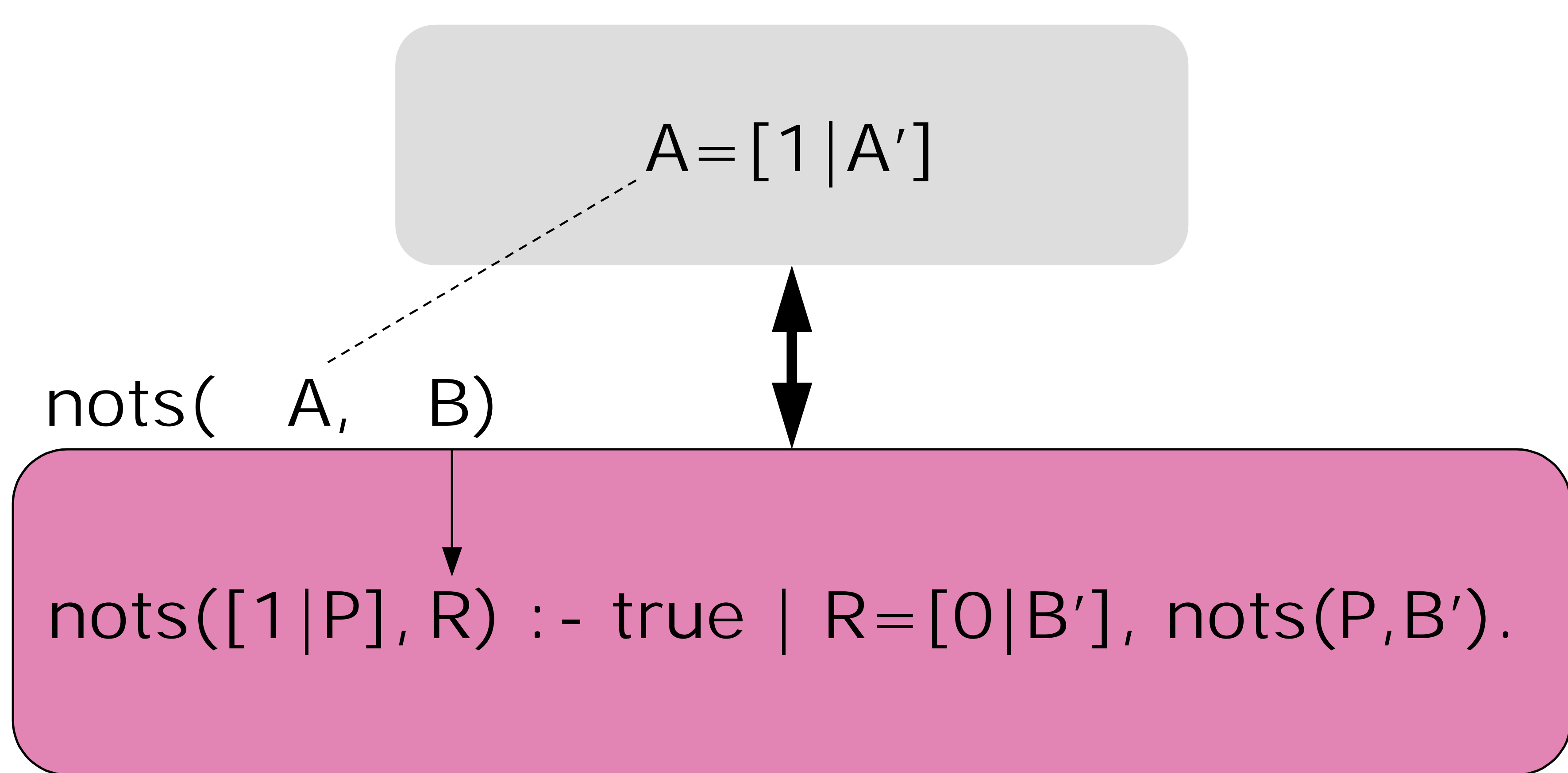
並行論理プログラミング

- プロセスが送受信する情報の実体は、変数のとりうる値への等号制約
- 記憶装置（store）は、各プロセスが生成した制約の集合（論理積）を保持
- 送信とは単一化ゴールを実行して等号制約を生成すること
- 受信とはゴール g（の現在値）と節頭部 h とのマッチングによって記憶装置から情報を獲得すること

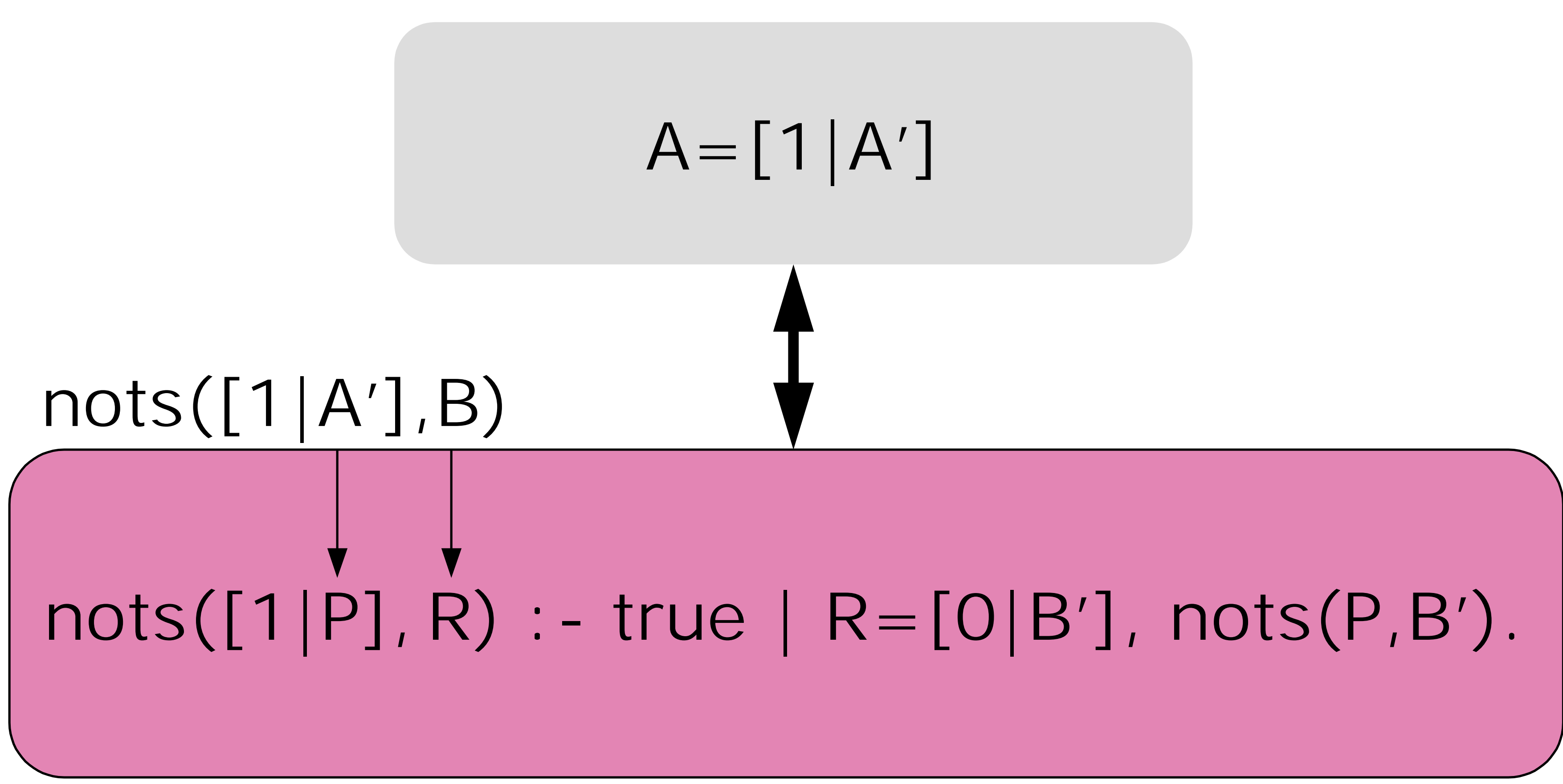
Reduction of an Inverter



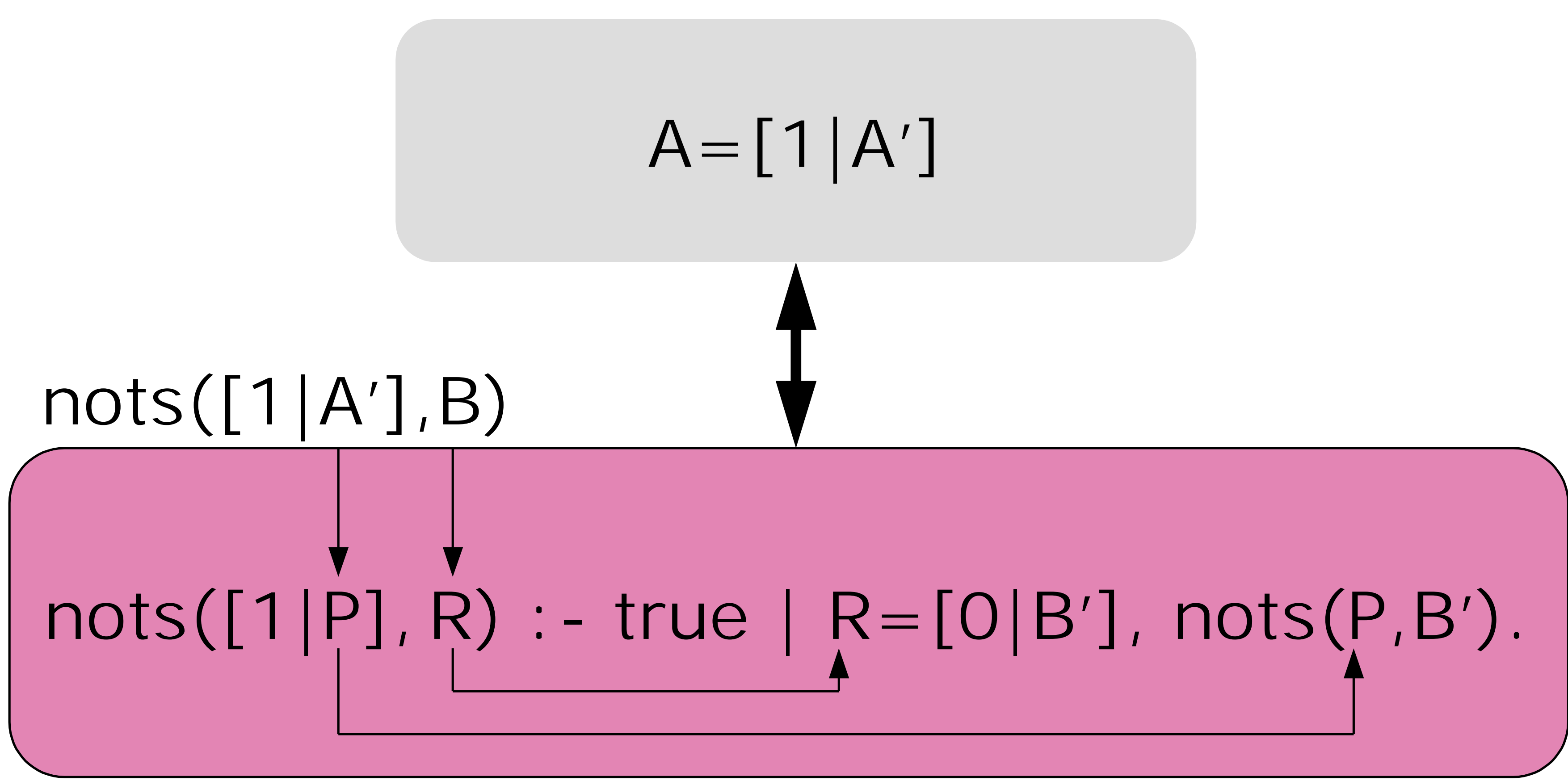
Reduction of an Inverter



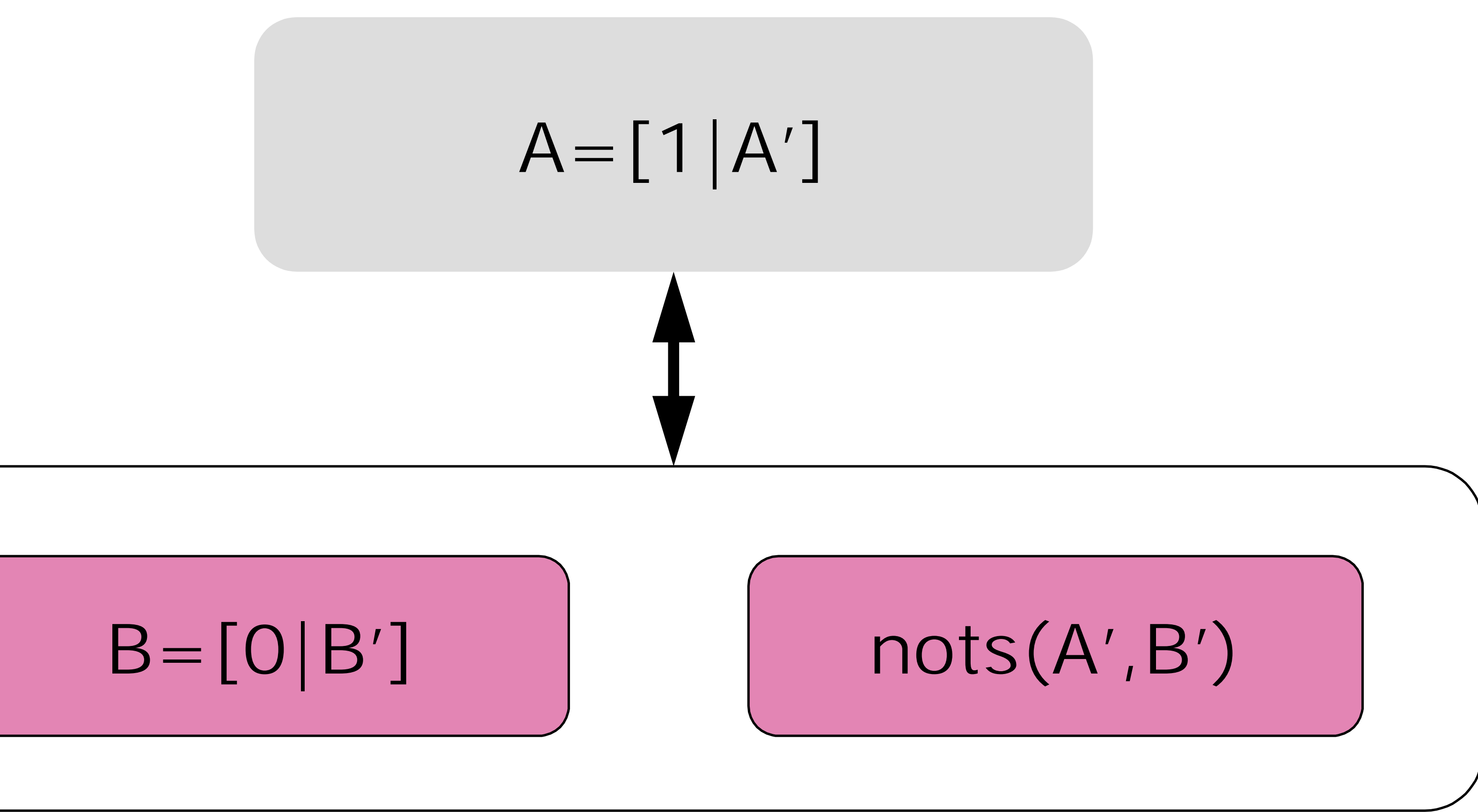
Reduction of an Inverter



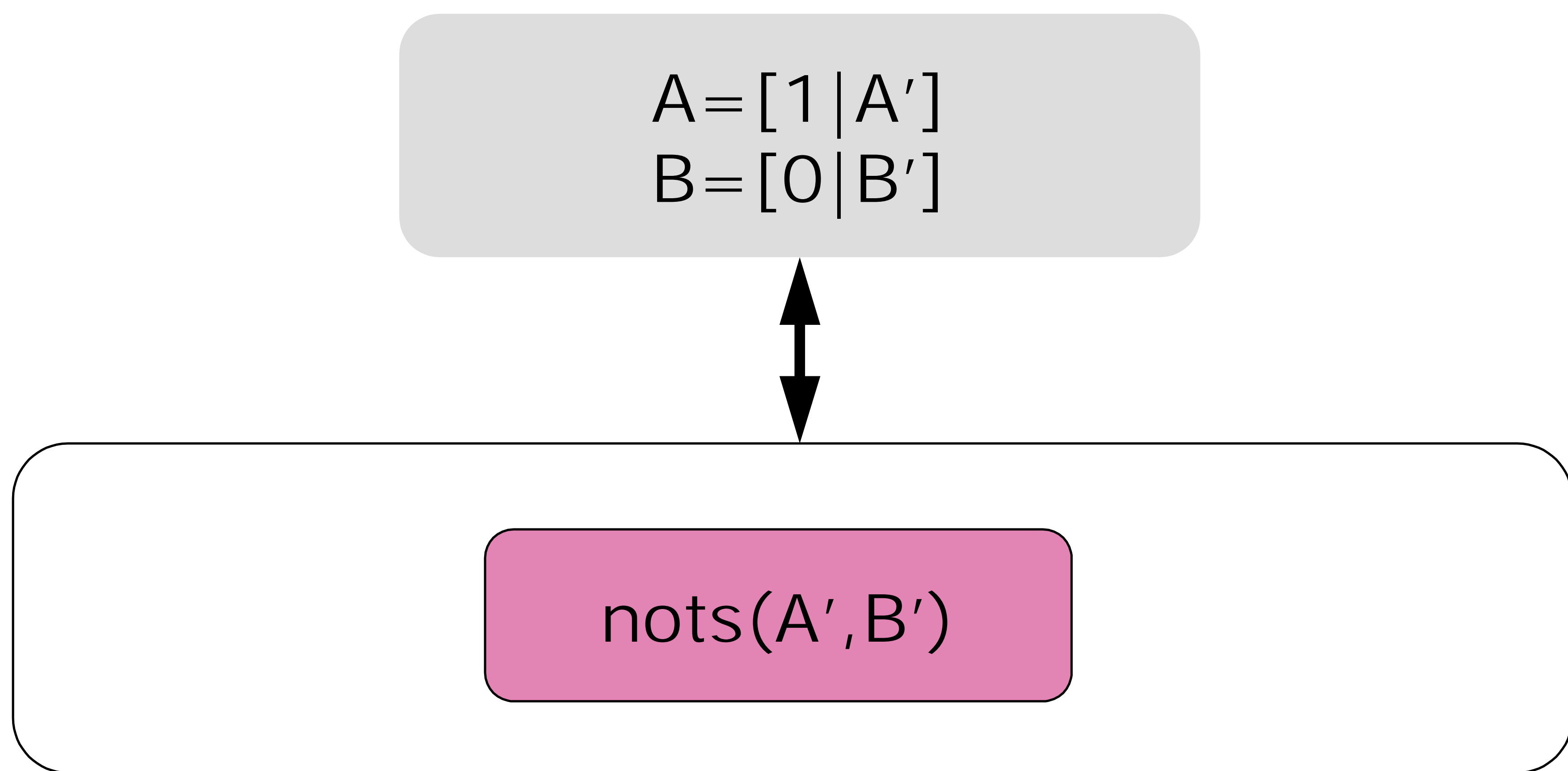
Reduction of an Inverter



Reduction of an Inverter



Reduction of an Inverter



受信の代数的解釈と論理的解釈

- 代数的解釈 ー ゴールの現在値 not(A,B)θ (θ ={A ← [1|A′]}) と節頭部 not([1|P],R) とのマッチング
- 論理的解釈 ー 適切な等号理論の下で、含意

  $\forall A \forall A' \forall B\ (A=[1|A'] \Rightarrow$

  $\exists P \exists R(\mathrm{not}(A,B)=\mathrm{not}([1|P],R)))$

  が成り立つかどうかの検査 (Maher, 1987)
  - より抽象的なとらえ方
  - 一般化可能 → 並行制約プログラミングへ

有限木の等号理論（1）

- **異なる関数・定数記号の対 f, g に対して**

$$\forall(\neg(f(X_1,\ldots,X_m)=g(Y_1,\ldots,Y_n)))$$

- X を含む項 t（X 自身は除く）に対して

$$\forall(\neg(t=X))$$

- 各 n 引数関数 f に対して

$$\forall\left(f(X_1,\ldots,X_n)=f(Y_1,\ldots,Y_n)\Rightarrow\bigcap_{i=1}^{n}(X_i=Y_i)\right)$$

- 各 n 引数関数 f に対して

$$\forall\left(\bigcap_{i=1}^{n}(X_i=Y_i)\Rightarrow f(X_1,\ldots,X_n)=f(Y_1,\ldots,Y_n)\right)$$

有限木の等号理論（2）

- 反射律 $\forall(X = X)$
- 対称律 $\forall(X = Y \Rightarrow Y = X)$
- 推移律 $\forall(X = Y \wedge Y = Z \Rightarrow X = Z)$

- 上記の集合を E とすると、E の下で記憶装置 S から制約 c が読み出せるとは

  $$E \models \forall(S \Rightarrow c)$$

  が成り立つことである

含意による読出し

- $E \models \forall(S \Rightarrow c)$ が成り立つことを確かめよ：

	S	c
(1)	X=s(0)	∃Y(X=s(Y))
(2)	X=s(Y) ∧ Y=0	X=s(0)
(3)	X=0 ∧ X=s(0)	任意の論理式

$E \models \forall(S \Rightarrow c)$ が成り立つならば、
$E \models \forall(S \wedge S' \Rightarrow c)$　も成り立つ（S' は任意）

含意による読出し

- c が読み出せない場合どうするか？

$\forall(S \Rightarrow c)$ \ $\forall(S \Rightarrow \neg c)$	false	true
false	maybe	no
true	yes	S is inconsistent

手続き型記憶装置

- 通常の（手続き型言語の）記憶装置
  - 場所（番地）から値への関数
  - 内容が非単調に変化

store(X,5) →

load(X) ←

store(X,6) →

load(X) ←

X＝5
Y＝8
Z＝3
⋮

論理型記憶装置

- 並行論理（制約）型言語における記憶装置
  - 制約の集合（論理積）
  - 単調に蓄積

tell(X=[2|Y]) →
ask(X=[ ]) ←
tell(Y=[3|Z]) →
ask(∃A(X=[2|A])) ←

X=[2|Y]
Y=[3|Z]
Z=[ ]
⋮

GHC サブセットの操作的意味論

- h :- true | B の形の節だけを許すGHCサブセットの構造的操作的意味論
  – cf. Plotkin, G. D., A Structural Approach to Operational Semantics. DAIMI FN-19, Computer Science Dept., Aarhus Univ., Denmark, 1981.

GHC サブセットの抽象構文

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

GHC プログラムの実行状態

- Configuration: $\langle B, C, V \rangle$
  - B : これから実行すべきゴールのマルチ集合
  - C : これまでに生成された等式の集合
  - V : これまでに使用した変数の集合
    - » cf. スタックポインタ
  - $\mathrm{vars}(B) \cup \mathrm{vars}(C) \subseteq V$
- $:- \ B_0$ の実行における initial configuration: $\langle B_0, \phi, \mathrm{vars}(B_0) \rangle$

構造的操作的意味論

- $P \vdash c \to c'$：与えられたプログラム P の下でのコンフィギュレーションの遷移関係
  - $P \vdash c \xrightarrow{*} c'$：反射推移閉包
- 次の形の推論規則で定義

$$\frac{P_1 \vdash c_1 \to c_1'}{P_2 \vdash c_2 \to c_2'} \text{ (if } Cond\text{)}$$

構造的操作的意味論

- 並行処理

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

- 単一化ゴールの実行 (tell)

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

構造的操作的意味論

- 非単一化ゴールの書換え (ask + reduction)

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

構造的操作的意味論

- 記憶装置の中の制約は、プログラムの実行につれて単調に増加
- 単調性から
  - 並行プロセスの実行は真に並列に実行可能
  - 情報の分割送信が可能

並列処理のモデル

- 一斉号令方式 — データ並列 (SIMD)
- 各人主体方式 — タスク並列 (MIMD)

  MIMD の方が汎用性が高い

  – SIMDをシミュレートできる。

- 並行論理プログラミングでは、ゴールの並列書換えが MIMD 並列処理。

並列処理における注意点

- プロセス（仕事）とプロセッサ（仕事をこなす主体）を区別しよう。
- 「マッピング」＝ プロセスを時空間上に配置すること
  - 空間：どこで（どのプロセッサが）実行するか？
  - 時間：いつ（どのような優先度で）実行するか？

並列処理における注意点

並列処理における注意点

- 計算における通信の重要性を認識しよう。

並列処理における注意点

- 計算における通信の重要性を認識しよう。
  - 計算機の処理内容のかなりの部分はデータの移動。

並列処理における注意点

- 計算における通信の重要性を認識しよう。
  – 計算機の処理内容のかなりの部分はデータの移動。
    » load, store

並列処理における注意点

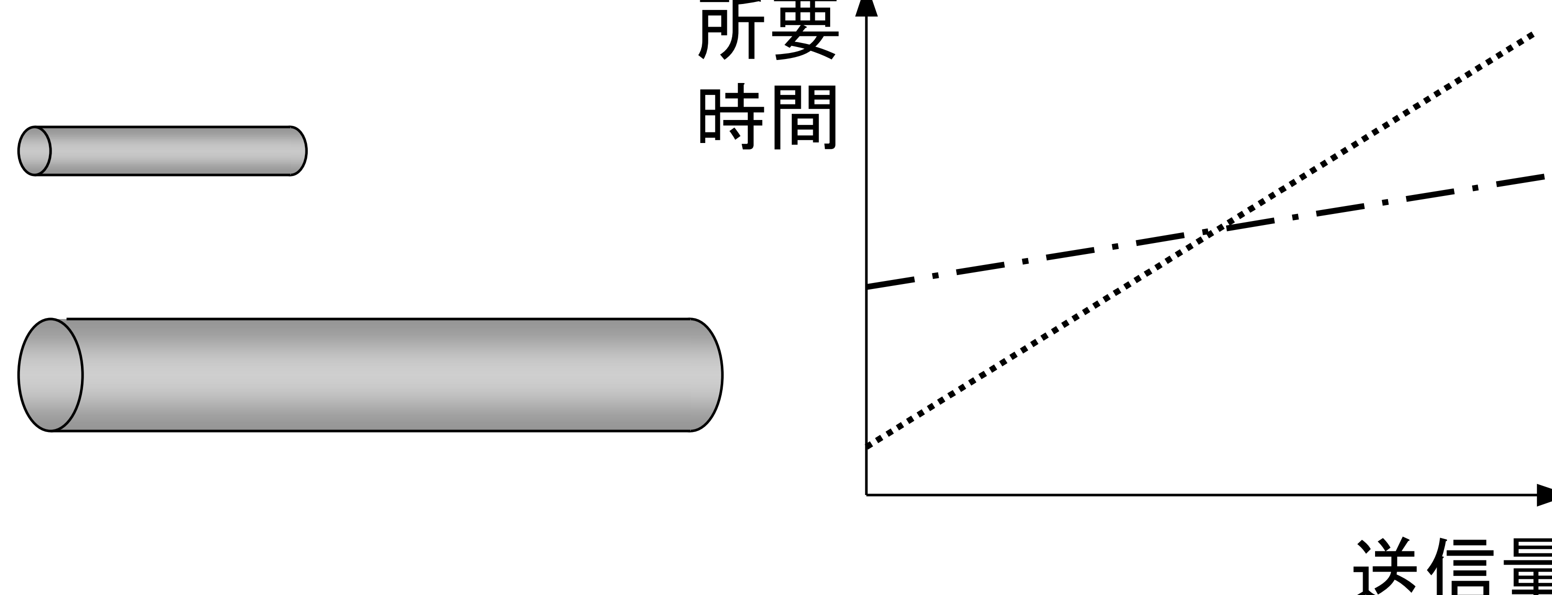
- 計算における通信の重要性を認識しよう。
  - 計算機の処理内容のかなりの部分はデータの移動。
    - load, store
    - 入出力

並列処理における注意点

- 計算における通信の重要性を認識しよう。
  - 計算機の処理内容のかなりの部分はデータの移動。
    - » load, store
    - » 入出力
    - » ネットワーク通信

並列処理における注意点

- 通信コストの（二つの独立な）指標
  - スループット
  - レイテンシ



並列処理における注意点

- 複数プロセス間の関係の定性的分類：
  - ほとんど無通信
    - 探索問題
    - 差分リスト連結
  - 密な単方向通信
    - データ駆動ストリーム計算
  - 密な双方向通信
    - 要求駆動ストリーム計算

並列処理における注意点

- 負荷は均等に分散すればいい（全員が平等に働いていればいい）とは限らない。
  - 通信の局所性
  - プロセス間の相互依存性
  - ボトルネック
  - 仕事分散の手間

スケーラビリティ (scalability)

- 並列アルゴリズムや並列アーキテクチャ設計の一つの基準
  - より多くのプロセッサを使えば性能が上がるか?
  - （アーキテクチャ）物理的に実現可能か？
- だが、並列処理にとって、並列度を上げることや、台数効果を上げることは、第一義的な目的ではない。

スケーラビリティ (scalability)

スケーラビリティ (scalability)

- **台数効果** (speedup)

スケーラビリティ (scalability)

- 台数効果 (speedup)
  - △ 設計した並列アルゴリズムを n 台で実行すると、１台で実行したときの何倍速いか？

スケーラビリティ (scalability)

- 台数効果 (speedup)
  - △ 設計した並列アルゴリズムを n 台で実行すると、１台で実行したときの何倍速いか？
  - ◯ 設計した並列アルゴリズムを n 台で実行すると、最良の逐次アルゴリズムの何倍速いか？

スケーラビリティ (scalability)

- 台数効果 (speedup)
  - △ 設計した並列アルゴリズムを *n* 台で実行すると、１台で実行したときの何倍速いか？
  - ◯ 設計した並列アルゴリズムを *n* 台で実行すると、最良の逐次アルゴリズムの何倍速いか？
- 効率 (efficiency)

スケーラビリティ (scalability)

- 台数効果 (speedup)
  - △ 設計した並列アルゴリズムを n 台で実行すると、１台で実行したときの何倍速いか？
  - ◯ 設計した並列アルゴリズムを n 台で実行すると、最良の逐次アルゴリズムの何倍速いか？
- 効率 (efficiency)
  - 台数効果／使用台数

スケーラビリティ (scalability)

- 台数効果 (speedup)
  - △ 設計した並列アルゴリズムを *n* 台で実行すると、１台で実行したときの何倍速いか？
  - ◯ 設計した並列アルゴリズムを *n* 台で実行すると、最良の逐次アルゴリズムの何倍速いか？
- 効率 (efficiency)
  - 台数効果／使用台数
  - １以上になるはずがないのだが . . .

スケーラビリティ (scalability)

- 台数効果 (speedup)
  - △ 設計した並列アルゴリズムを n 台で実行すると、１台で実行したときの何倍速いか？
  - ◯ 設計した並列アルゴリズムを n 台で実行すると、最良の逐次アルゴリズムの何倍速いか？
- 効率 (efficiency)
  - 台数効果／使用台数
  - １以上になるはずがないのだが . . .
  - super-linear speedup が観測されることがある。なぜだろう？

スケーラビリティ (scalability)

- Amdahl の法則
  - 総作業の中で、並列処理できない部分の割合が p を占めるならば、いかに分業しても高々 1／p 倍しか速くならない。
  - しかし p は、問題のサイズが大きくなれば小さくなることが多い。

AND型分業とOR型分業

AND型分業とOR型分業

- AND型分業 ＝ 共同作業
  - 全員の作業が結果に反映

AND型分業とOR型分業

- AND型分業 ＝ 共同作業
  - 全員の作業が結果に反映
- OR型分業 ＝ 競争
  - 一人の作業だけが結果に反映
  - ではなぜ多数人で実行するのか？
  - 探索問題において重要な手法

見込み計算（投機的計算）

見込み計算（投機的計算）

- 見込み計算 ＝ 必要になるかどうかわからないが、とりあえずやっておく計算のこと。

見込み計算（投機的計算）

- 見込み計算 ＝ 必要になるかどうかわからないが、とりあえずやっておく計算のこと。
  – cf. プロセッサの分岐予測

見込み計算（投機的計算）

- 見込み計算 ＝ 必要になるかどうかわからないが、とりあえずやっておく計算のこと。
  – cf. プロセッサの分岐予測
    入出力データのバッファリング

見込み計算（投機的計算）

- 見込み計算 ＝ 必要になるかどうかわからないが、とりあえずやっておく計算のこと。
  - cf. プロセッサの分岐予測
    入出力データのバッファリング
- 必要性が明らかでない計算は実行しないという方針は、多くの場合性能上不利。

見込み計算（投機的計算）

- 見込み計算 ＝ 必要になるかどうかわからないが、とりあえずやっておく計算のこと。
  - cf. プロセッサの分岐予測
    入出力データのバッファリング
- 必要性が明らかでない計算は実行しないという方針は、多くの場合性能上不利。
- ただし、投機的処理が必須処理を妨げてはいけない。

GHC と KL1 との関係

- GHC は並行プログラミング言語 ー 並列実行の可能性を最大限に示唆している（不必要な逐次性は一切導入していない）が、処理系がどう実行するかは任意
- KL1 は並行・並列プログラミング言語 ー 計算の空間的・時間的マッピング機能を提供

GHC と KL1 との関係

- 従来の言語におけるマッピング機能
  - 逐次実行の指定 ＝ 低レベル、高効率マッピング
  - 並列実行の指定 ＝ 高レベル、低効率マッピング（OSの介在）
- 「並列」のための言語要素は「並行」のための言語要素と異なり、処理系依存となりうる
  - KL1 では両者を分離

KL1 における優先度制御

- 優先度制御の目的
  - 空間効率の改善
  - 時間効率の改善
  - 通常の仕事と管理の仕事の差別化
    - 高優先度の管理業務
    - 低優先度の管理業務

KL1 における優先度制御

- 優先度制御の目的
  - 空間効率の改善（例：ストリームやゴールの過剰生産防止）
  - 時間効率の改善（例：OR型分業におけるヒューリスティクス）
  - 通常の仕事と管理の仕事の差別化
    - 高優先度の管理業務（例：枝刈り情報の伝達）
    - 低優先度の管理業務（例：データベース再編成）

KL1 における優先度制御

- KL1 における言語機能
  - @lower_priority : ゴール間の相対優先度
  - alternatively : プログラム節間の相対優先度
- 優先度制御機能はプログラムの論理的意味を変えない
  - 正当性に関わる事項（処理系独立）と効率にかかわる事項（処理系依存）の分離
- 優先度制御機能は処理系へのガイドライン
  - プラグマ (pragma)
  - 厳密に守られるとは限らない