A **Lattice-Based** Approach to **Deterministic Parallelism**

Lindsey Kuper and Ryan R. Newton
Indiana University

POPL 2013 Student Session
25 January 2013

What does this program evaluate to?

$$
\begin{aligned}
&\text{let } \_ = \text{put } l\ 3 \text{ in} \\
&\quad \text{let par } v = \text{get } l \\
&\qquad\qquad\quad\ \ \_ = \text{put } l\ 4 \\
&\quad\ \ \text{in } v
\end{aligned}
$$

Disallow multiple writes?

$$
\begin{array}{l}
\text{let } \_ = \text{put } l\ 3 \text{ in} \\
\quad \text{let par } v = \text{get } l \\
\qquad\qquad\quad\ \ \_ = \text{put } l\ 4 \\
\quad\ \ \text{in } v
\end{array}
$$

Disallow multiple writes?

let _ = put l 3 in

  let par v = get l

    _ = put l 4 ✗

  in v

Tesler and Enea, 1968

Arvind *et al.*, 1989

“IVars”

Deterministic programs that single-assignment forbids

$$
\begin{aligned}
&\text{let } \_ = \text{put } l\ 3 \text{ in} \\
&\quad \text{let par } v = \text{get } l \\
&\qquad\qquad\quad \_ = \text{put } l\ 3 \\
&\quad\ \ \text{in } v
\end{aligned}
$$

Deterministic programs that single-assignment forbids

$$
\begin{array}{l}
\text{let } \_ = \text{put } l \ 3 \text{ in} \\
\quad \text{let par } v = \text{get } l \\
\qquad\qquad \_ = \text{put } l \ 3 \\
\quad\quad \text{in } v
\end{array}
$$

$$
\begin{array}{l}
\text{let par } \_ = \text{put } l \ (4, \bot) \\
\qquad\qquad \_ = \text{put } l \ (\bot, 3) \\
\quad \text{in get } l
\end{array}
$$

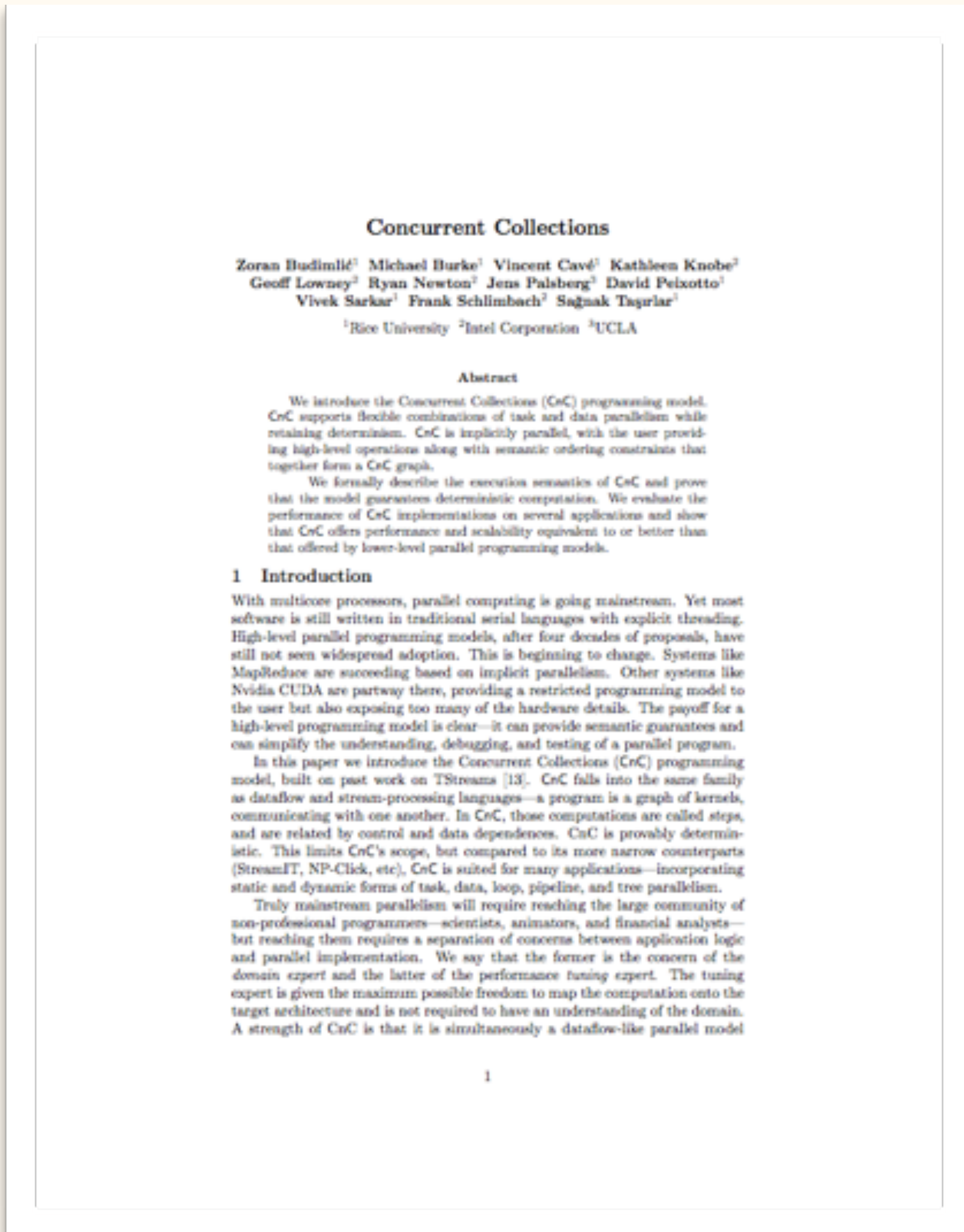
Deterministic programs that single-assignment forbids

$$
\begin{aligned}
&\text{let } \_ = \text{put } l\ 3 \text{ in} \\
&\quad \text{let par } v = \text{get } l \\
&\qquad\qquad \_ = \text{put } l\ 3 \\
&\quad\ \text{in } v
\end{aligned}
$$

$$
\begin{aligned}
&\text{let par } \_ = \text{put } l\ (4, \bot) \\
&\qquad\qquad \_ = \text{put } l\ (\bot, 3) \\
&\quad \text{in get } l
\end{aligned}
$$

$$
\begin{aligned}
&\text{let par } \_ = \text{insert } l\ ''1111'' \\
&\qquad\qquad \_ = \text{insert } l\ ''1100'' \\
&\quad \text{in get } l
\end{aligned}
$$

From *Concurrent Collections*…



Concurrent Collections

Zoran Budimlić[1] Michael Burke[1] Vincent Cavé[1] Kathleen Knobe[2]
Geoff Lowney[2] Ryan Newton[2] Jens Palsberg[3] David Peixotto[1]
Vivek Sarkar[1] Frank Schlimbach[2] Sağnak Taşırlar[1]

[1]Rice University [2]Intel Corporation [3]UCLA

Abstract

We introduce the Concurrent Collections (CnC) programming model. CnC supports flexible combinations of task and data parallelism while retaining determinism. CnC is implicitly parallel, with the user providing high-level operations along with semantic ordering constraints that together form a CnC graph.

We formally describe the execution semantics of CnC and prove that the model guarantees deterministic computation. We evaluate the performance of CnC implementations on several applications and show that CnC offers performance and scalability equivalent to or better than that offered by lower-level parallel programming models.

1 Introduction

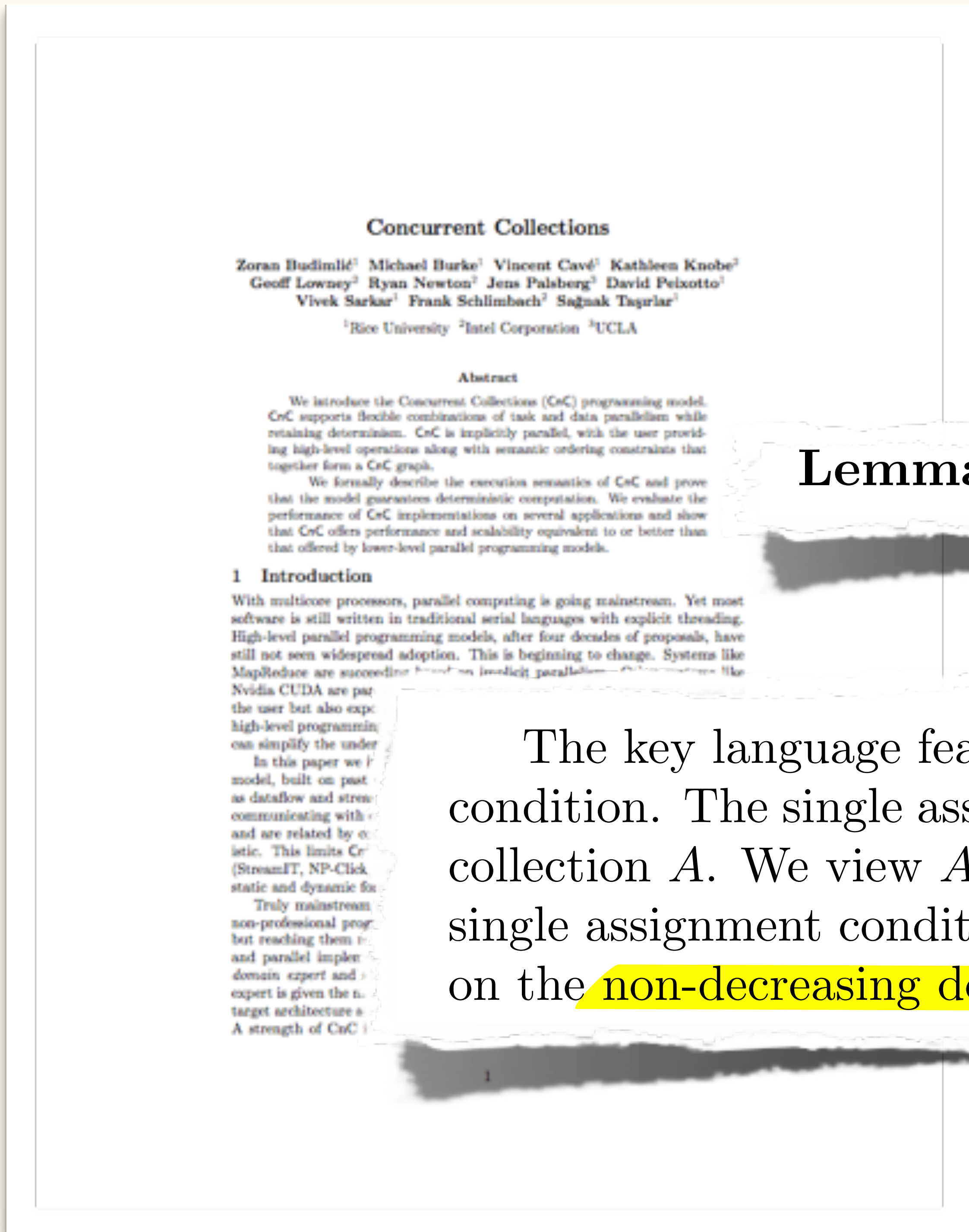
With multicore processors, parallel computing is going mainstream. Yet most software is still written in traditional serial languages with explicit threading. High-level parallel programming models, after four decades of proposals, have still not seen widespread adoption. This is beginning to change. Systems like MapReduce are succeeding based on implicit parallelism. Other systems like Nvidia CUDA are partway there, providing a restricted programming model to the user but also exposing too many of the hardware details. The payoff for a high-level programming model is clear—it can provide semantic guarantees and can simplify the understanding, debugging, and testing of a parallel program.

In this paper we introduce the Concurrent Collections (CnC) programming model, built on past work on TStreams [13]. CnC falls into the same family as dataflow and stream-processing languages—a program is a graph of kernels, communicating with one another. In CnC, those computations are called *steps*, and are related by control and data dependences. CnC is provably deterministic. This limits CnC's scope, but compared to its more narrow counterparts (StreamIT, NP-Click, etc), CnC is suited for many applications—incorporating static and dynamic forms of task, data, loop, pipeline, and tree parallelism.

Truly mainstream parallelism will require reaching the large community of non-professional programmers—scientists, animators, and financial analysts—but reaching them requires a separation of concerns between application logic and parallel implementation. We say that the former is the concern of the *domain expert* and the latter of the performance *tuning expert*. The tuning expert is given the maximum possible freedom to map the computation onto the target architecture and is not required to have an understanding of the domain. A strength of CnC is that it is simultaneously a dataflow-like parallel model

Budimlić *et al.*, 2010

From *Concurrent Collections*...


Concurrent Collections

Zoran Budimlić[1] Michael Burke[1] Vincent Cavé[1] Kathleen Knobe[2]
Geoff Lowney[2] Ryan Newton[2] Jens Palsberg[3] David Peixotto[1]
Vivek Sarkar[1] Frank Schlimbach[2] Sağnak Taşırlar[1]
[1]Rice University [2]Intel Corporation [3]UCLA

Abstract

We introduce the Concurrent Collections (CnC) programming model. CnC supports flexible combinations of task and data parallelism while retaining determinism. CnC is implicitly parallel, with the user providing high-level operations along with semantic ordering constraints that together form a CnC graph.

We formally describe the execution semantics of CnC and prove that the model guarantees deterministic computation. We evaluate the performance of CnC implementations on several applications and show that CnC offers performance and scalability equivalent to or better than that offered by lower-level parallel programming models.

1 Introduction

With multicore processors, parallel computing is going mainstream. Yet most software is still written in traditional serial languages with explicit threading. High-level parallel programming models, after four decades of proposals, have still not seen widespread adoption. This is beginning to change. Systems like

Lemma 3.2. **(Monotonicity)** *If* $\sigma \rightarrow \sigma'$*, then* $\sigma \leq \sigma'$*.*

The key language feature that enables determinism is the single assignment condition. The single assignment condition guarantees monotonicity of the data collection A. We view A as a partial function from integers to integers and the single assignment condition guarantees that we can establish an ordering based on the non-decreasing domain of A.

Budimlić *et al.*, 2010

...to KPNs

INFORMATION PROCESSING 74 – NORTH-HOLLAND PUBLISHING COMPANY (1974)

THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

Gilles KAHN

IRIA-Laboria, Domaine de Voluceau, 78150
Rocquencourt, France

and

Commissariat à l'Energie Atomique, France

In this paper, we describe a simple language for parallel programming. Its semantics is studied thoroughly. The desirable properties of this language and its deficiencies are exhibited by this theoretical study. Basic results on parallel program schemata are given. We hope in this way to make a case for a more formal (i.e. mathematical) approach to the design of languages for systems programming and the design of operating systems.

There is a wide disagreement among systems designers as to what are the best primitives for writing systems programs. In this paper, we describe a simple language for parallel programming and study its mathematical properties.

1. A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING.

The features of our mini-language are exhibited on the sample program S on fig.1. The conventions are close to Algol and we only insist upon the new features. The program S consists of a set of declarations and a body. Variables of type *integer channel* are declared at line (1), and for any simple type σ (boolean, real, etc...) we could have declared a σ *channel*. Then processes f, g and h are declared, much like procedures. Aside from usual parameters (passed by value in this example, like INIT at line (3)), we can declare in the heading of the process how it is linked to other processes : at line (2) f is stated to communicate via two input lines that can carry integers, and one similar output line.
The body of a process is an usual Algol program except for invocation of *wait* on an input line (e.g. at (4)) or *send* a variable *on* a line of compatible type (e.g. at (5)). The process stays blocked on a *wait* until something is being sent on this line by another process, but nothing can prevent a process from performing a *send* on a line.
In other words, processes communicate via first-in first-out (fifo) queues.
Calling instances of the processes is done in the body of the main program at line (6) where the actual names of the channels are bound to the formal parameters of the processes. The infix operator *par* initiates the concurrent activation of the processes. Such a style of programming is close to may systems using EVENT mechanisms ([1],[2],[3],[4]). A pictorial representation of the program is the schema P on fig.2., where the nodes represent processes and the arcs communication channels between these processes.
What sort of things would we like to prove on a program like S ? Firstly, that all processes in S run forever. Secondly, more precisely, that S prints out (at line (7)) an alternating sequence of 0's and 1's forever. Third, that if one of the processes were to stop at some time for an extraneous reason, the whole system would stop.
The ability to state formally this kind of property of a parallel program and to prove them within a formal logical framework is the central motivation for the theoretical study of the next sections.

2. PARALLEL COMPUTATION.

Informally speaking, a parallel computation is organized in the following way : some autonomous computing stations are connected to each other in a network by communication lines. Computing stations exchange information through these lines. A given station computes on data coming along its input lines,

```
    Begin
(1) Integer channel X, Y, Z, T1, T2 ;
(2) Process f(integer in U,V; integer out W) ;
      Begin integer I ; logical B ;
            B := true ;
            Repeat Begin
(4)             I := if B then wait(U) else wait(V) ;
(7)             print (I) ;
(5)             send I on W ;
                B := ¬B ;
                end ;
      End ;
    Process g(integer in U ; integer out V, W) ;
      Begin integer I ; logical B ;
        B := true ;
        Repeat Begin
          I := wait (U) ;
          if B then send I on V else send I on W ;
          B := ¬B ;
          End ;
      End ;
(3) Process h(integer in U;integer out V; integer INIT);
      Begin integer I ;
        send INIT on V ;
        Repeat Begin
          I := wait(U) ;
          send I on V ;
          End ;
      End ;
   Comment : body of mainprogram ;
(6) f(Y,Z,X) par g(X,T1,T2) par h(T1,Y,0) par h(T2,Z,1
   End ;
```

Fig.1. Sample parallel program S.

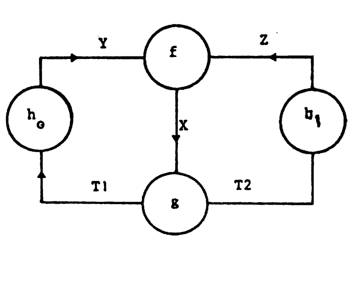


Fig.2. The schema P for the program S.

Kahn, 1974

...to KPNs

INFORMATION PROCESSING 74 – NORTH-HOLLAND PUBLISHING COMPANY (1974)

THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

Gilles KAHN

IRIA-Laboria, Domaine de Voluceau, 78150
Rocquencourt, France

and

Commissariat à l'Energie Atomique, France

<u>Monotonicity</u> means that receiving more input at a computing station can only provoke it to send more output. Indeed this a crucial property since it allows parallel operation : <u>a machine need not have all of its input to start computing, since future input concerns only future output.</u>

The kind of parallel programming we have studied in this paper is severely limited : it can produce only <u>determinate</u> programs.

2. PARALLEL COMPUTATION.

Informally speaking, a parallel computation is organized in the following way : some autonomous computing stations are connected to each other in a network by communication lines. Computing stations exchange information through these lines. A given station computes on data coming along its input lines,

Fig.1. Sample parallel program S.

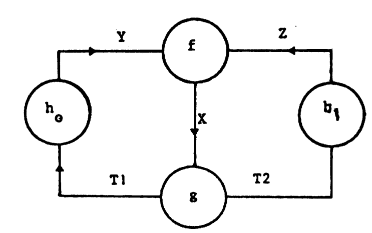
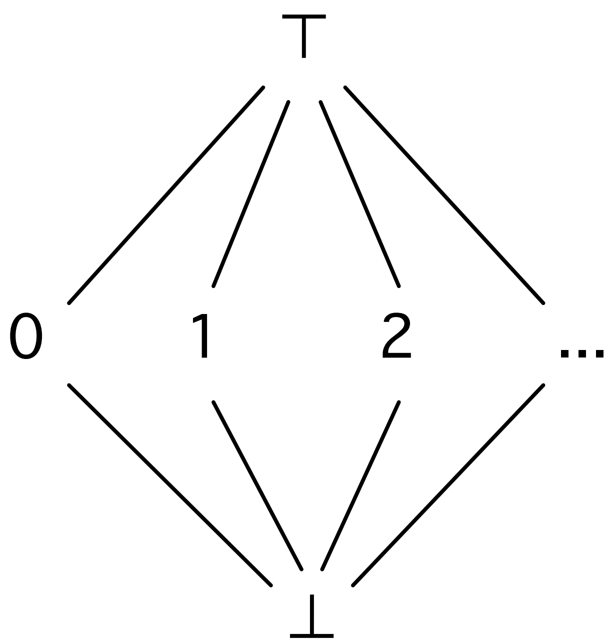


Fig.2. The schema P for the program S.

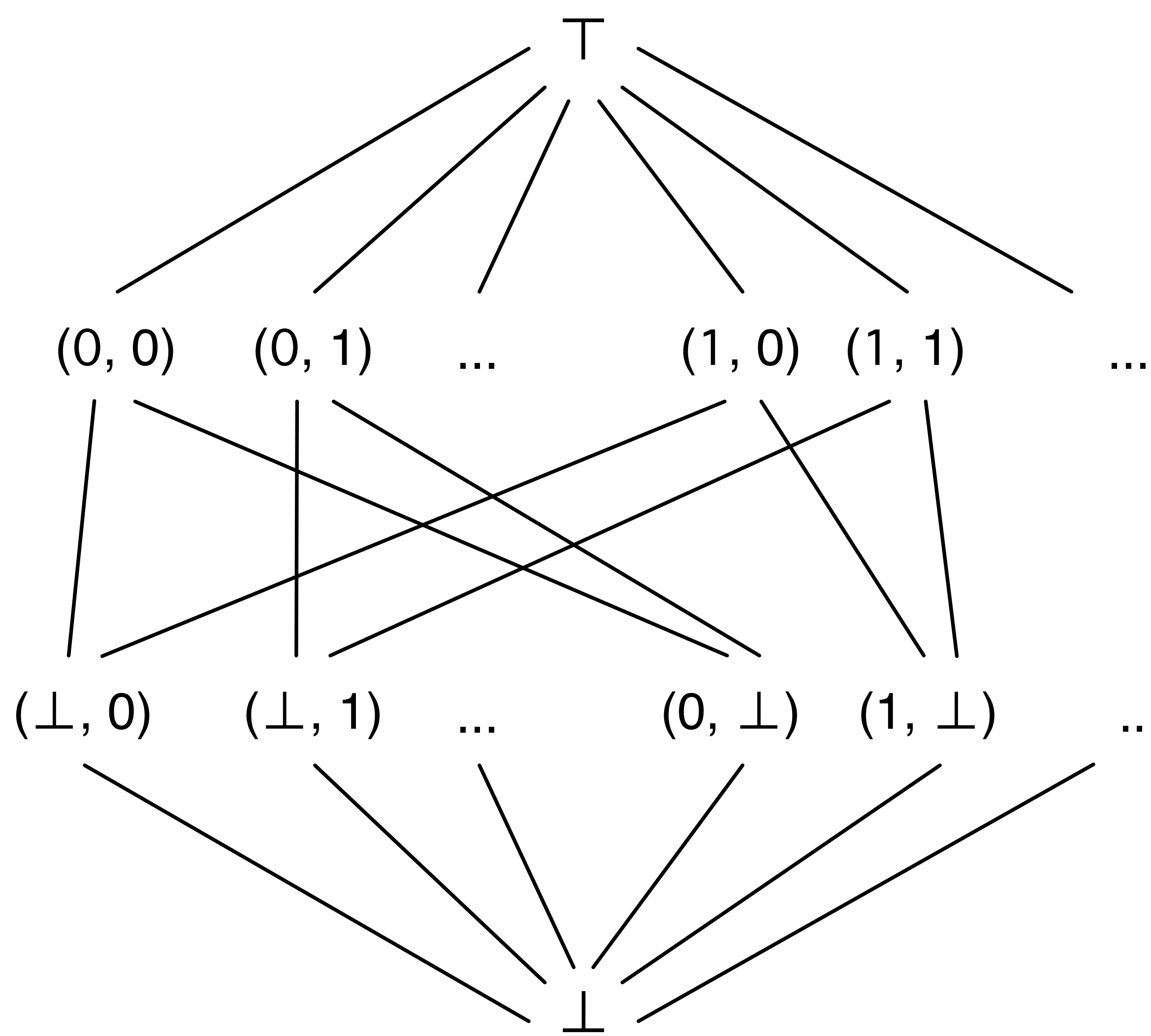
Kahn, 1974

Monotonicity causes deterministic parallelism!

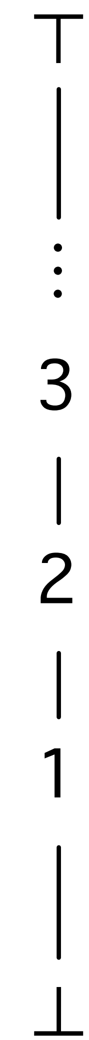
Parameterizing our language: LVars



IVar

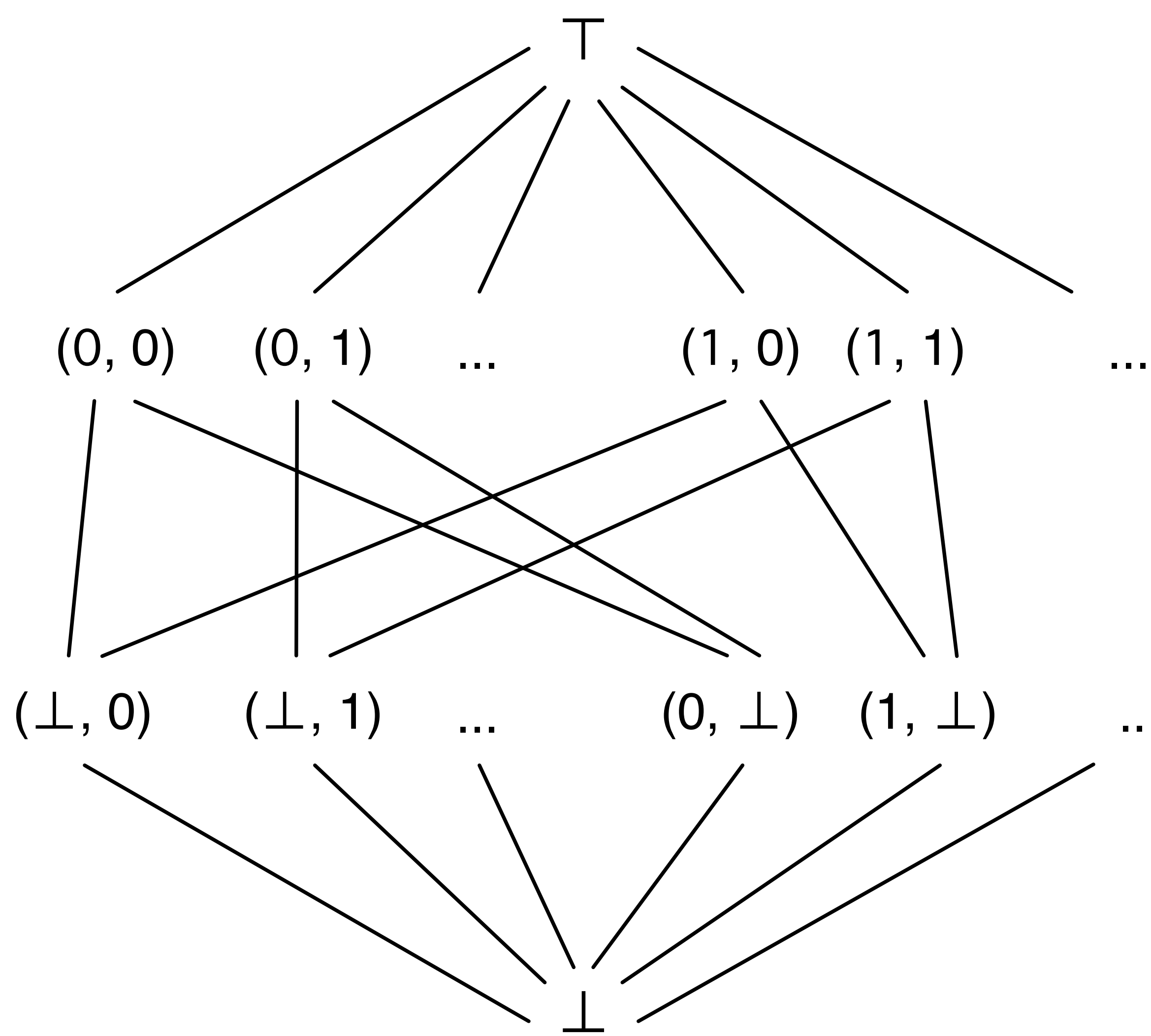


Pair of IVars



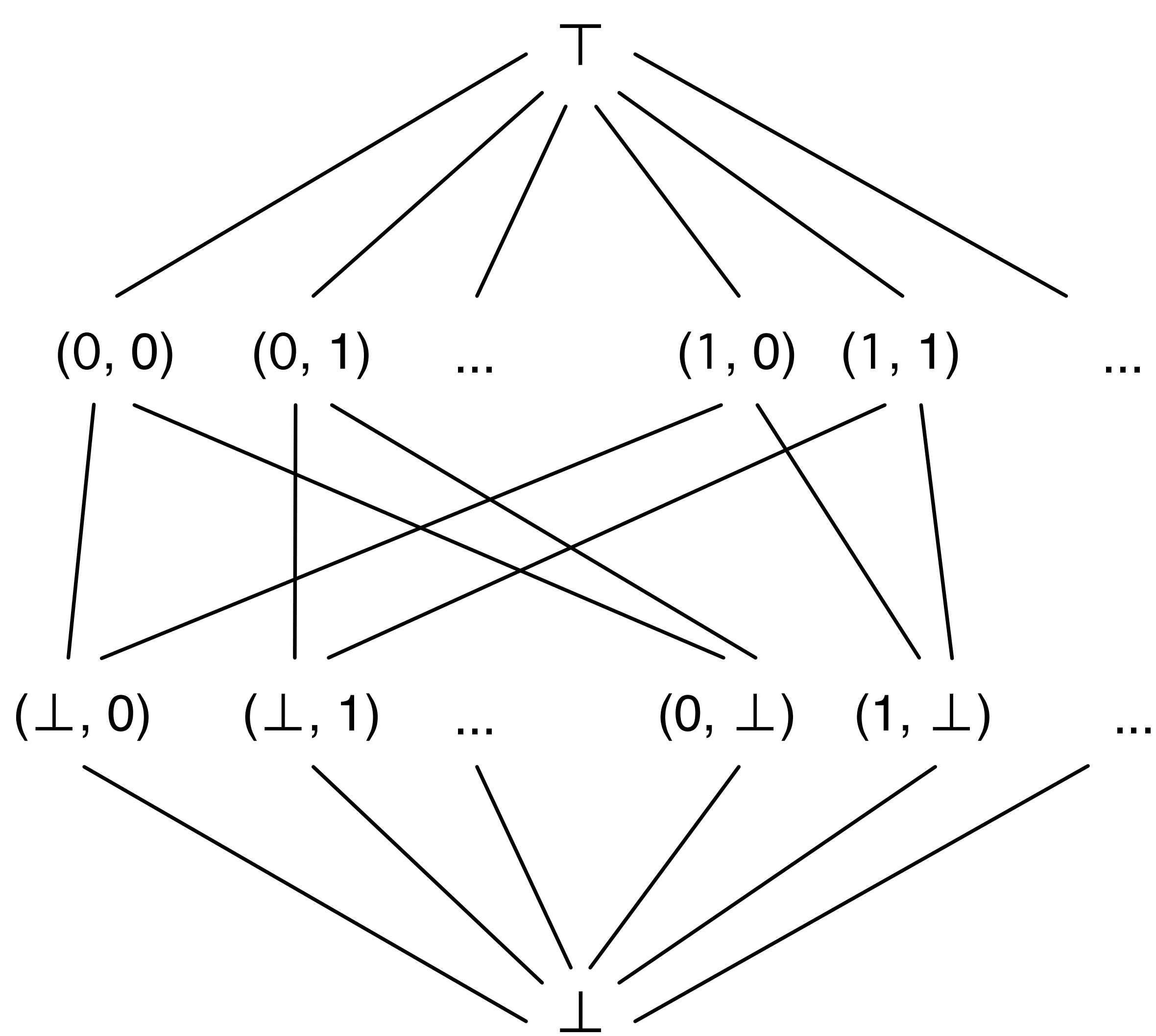
Counter

Parameterizing our language: LVars



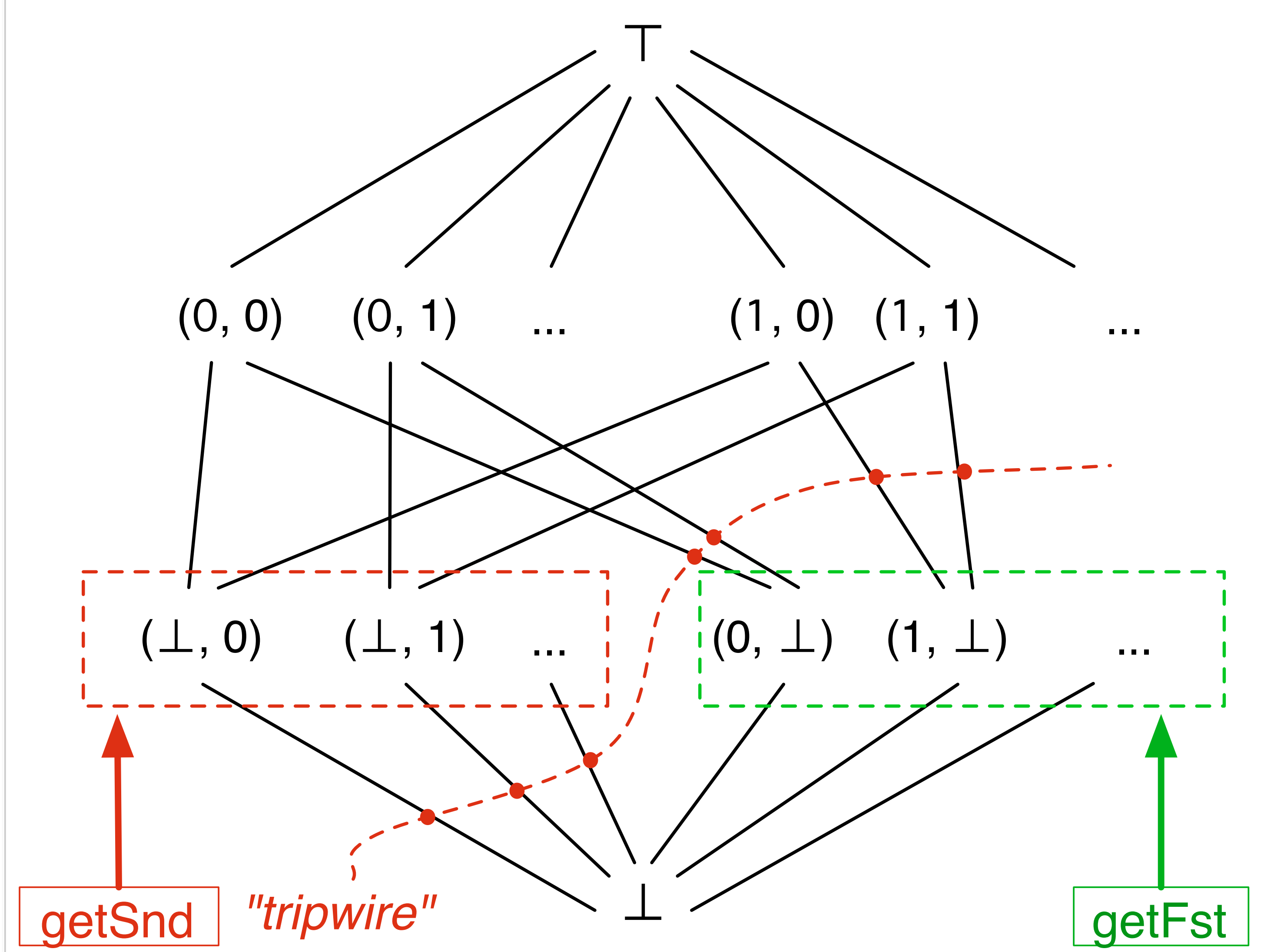
Pair of IVars

Parameterizing our language: LVars



Pair of IVars

Parameterizing our language: LVars



Pair of IVars

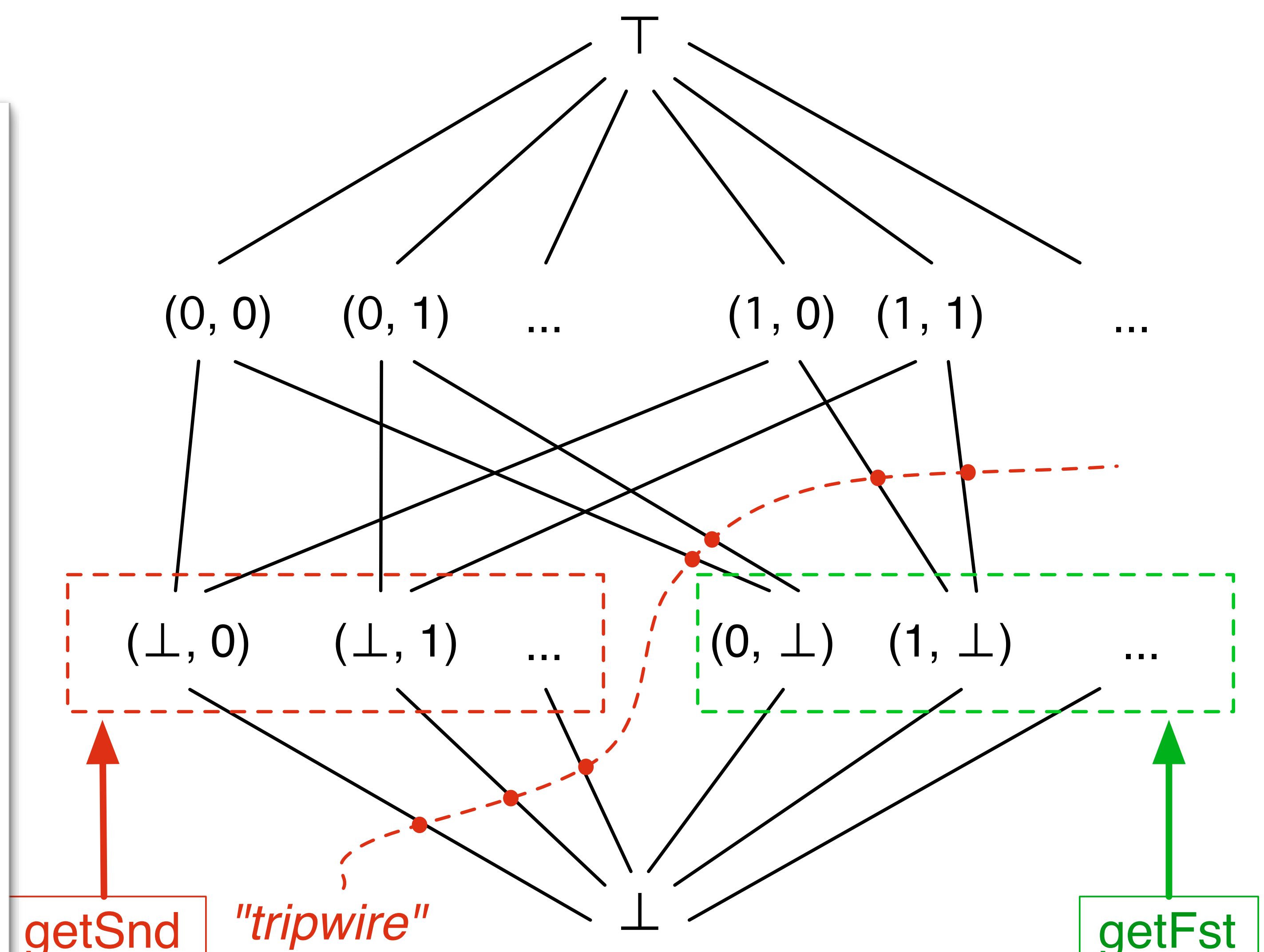
Parameterizing our language: LVars

let $\_ = \text{put } p \ \{(\bot, 4)\}$ in

  let par $v_1 = \text{getFst } p$

    $\_ = \text{put } p \ \{(3, 4)\}$

  in $\ldots v_1 \ldots$



Pair of IVars

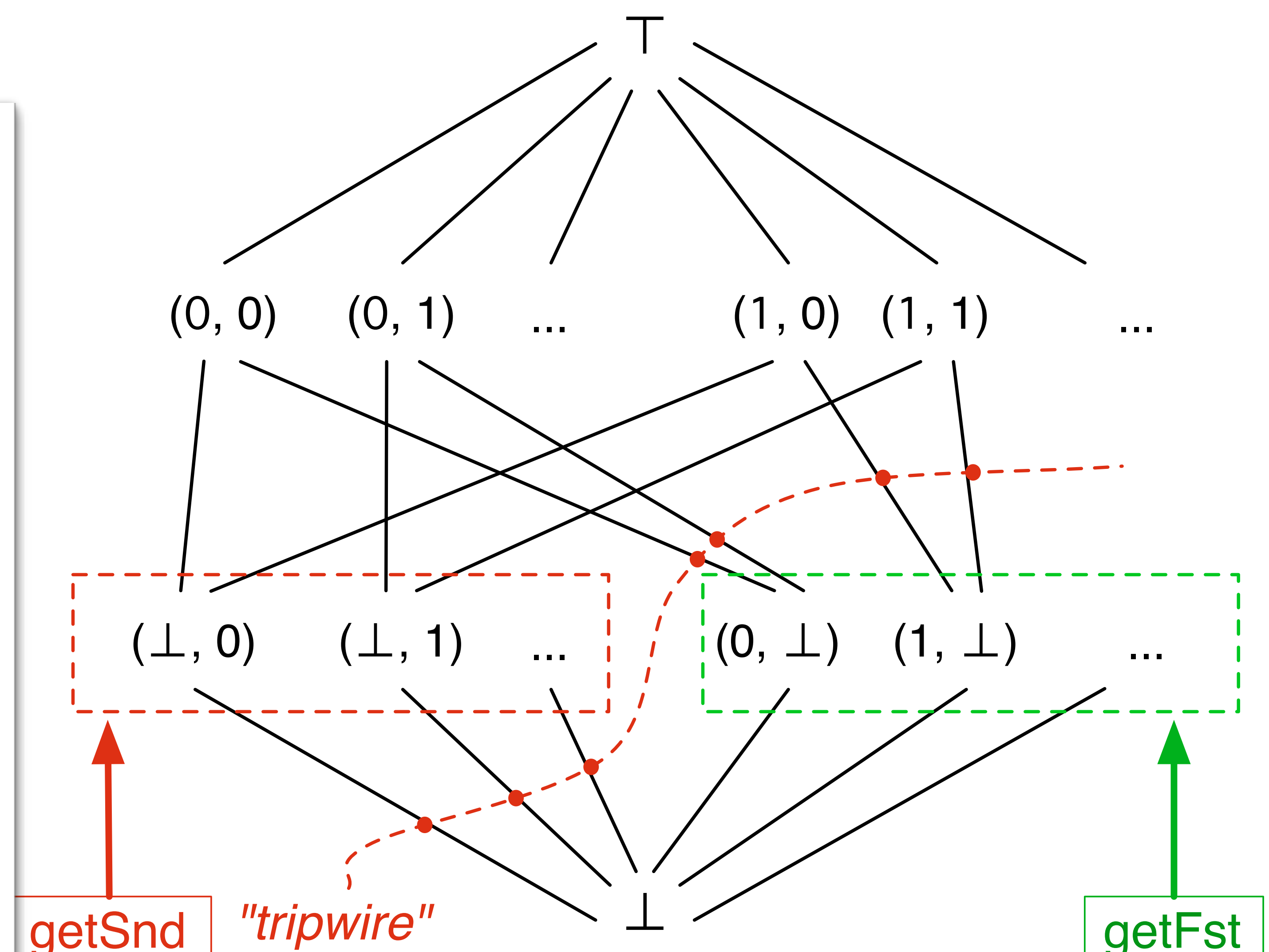
Parameterizing our language: LVars

let $_ = $ put p $\{(\bot, 4)\}$ in

let par $v_1 =$ getFst p

$\_ \;=$ put p $\{(3, 4)\}$

in $\ldots v_1 \ldots$



Pair of IVars

getFst $p \triangleq$ get p $\{(n, \bot) \mid n \in \mathbb{N}\}$

Two take-aways

Monotonicity causes deterministic parallelism

Monotonically increasing writes
+ threshold reads
= deterministic parallelism

More in our TR

- Complete syntax and semantics
- Proof of determinism
  - A frame property!
  - Location renaming is surprisingly tricky!
- Subsuming existing models
  - KPNs, CnC, monad-par
- Support for controlled nondeterminism
  - “probation” state

Grazie!

Email: lkuper@cs.indiana.edu
Twitter: @lindsey
Web: cs.indiana.edu/~lkuper
Research group: lambda.cs.indiana.edu