LVars:
Lattice-based Data Structures for Deterministic Parallelism

Lindsey Kuper and Ryan Newton
Indiana University

FHPC ’13, Boston, MA, USA
September 23, 2013
What does this program evaluate to?

\[
p = \begin{array}{l}
\text{do}
\begin{array}{l}
\text{num} <- \text{newEmptyMVar}
\text{forkIO} \$ \text{do putMVar num 3}
\text{forkIO} \$ \text{do putMVar num 4}
\text{v} <- \text{takeMVar num}
\text{return v}
\end{array}
\end{array}
\end{array}
\]
landin:ivar-examples lkuper$ make data-race-example
ghc -O2 data-race-example.hs -rtsopts -threaded
Linking data-race-example ...
while true; do ./data-race-example +RTS -N2; done
Disallow multiple writes?

\[
p = \text{do}
\begin{align*}
    \text{num} & \leftarrow \text{newEmptyMVar} \\
    \text{forkIO} & \hat{\text{do}} \quad \text{putMVar num} \ 3 \\
    \text{forkIO} & \hat{\text{do}} \quad \text{putMVar num} \ 4 \\
    \text{v} & \leftarrow \text{takeMVar num} \\
    \text{return v}
\end{align*}
\]
Disallow multiple writes?

\[ \begin{align*}
\text{\texttt{p} } = & \quad \texttt{do} \\
& \quad \text{\texttt{num} } \leftarrow \texttt{newEmptyMVar} \\
& \quad \texttt{forkIO} \quad \texttt{do} \quad \texttt{putMVar num 3} \\
& \quad \texttt{forkIO} \quad \texttt{do} \quad \texttt{putMVar num 4} \\
& \quad \texttt{v} \leftarrow \texttt{takeMVar num} \\
& \quad \texttt{return v}
\end{align*} \]
Disallow multiple writes?

\[
\begin{align*}
p &= \textbf{do} \\
    &\quad \text{num} \leftarrow \text{newEmptyMVar} \\
    &\quad \text{forkIO} \; \textbf{$ do$} \; \text{putMVar num 3} \\
    &\quad \text{forkIO} \; \textbf{$ do$} \; \text{putMVar num 4} \\
    &\quad \text{v} \leftarrow \text{takeMVar num} \\
    &\quad \text{return v}
\end{align*}
\]

Tesler and Enea, 1968
Arvind et al., 1989

IVars
Disallow multiple writes?

```haskell
p :: Par Int
p = do
    num <- new
    fork $ do
        put num 3
    fork $ do
        put num 4
    v <- get num
    return v
```

Tesler and Enea, 1968
Arvind et al., 1989
Disallow multiple writes?

\[
p :: \text{Par Int}
\]
\[
p = \text{do}
\]
\[
\text{num} \leftarrow \text{new}
\]
\[
\text{fork} \quad \text{do} \quad \text{put} \quad \text{num} \quad 3
\]
\[
\text{fork} \quad \text{do} \quad \text{put} \quad \text{num} \quad 4
\]
\[
v \leftarrow \text{get} \quad \text{num}
\]
\[
\text{return} \quad v
\]

Tesler and Enea, 1968
Arvind et al., 1989

IVars

`./ivar-example +RTS -N2`
`ivar-example: multiple put`
Deterministic programs that single-assignment forbids

\[
p :: \text{ Par Int}
\]
\[
p = \text{ do}
\]
\[
\text{num} \leftarrow \text{new}
\]
\[
\text{fork} \; $ \; \text{do} \; \text{put} \; \text{num} \; 4
\]
\[
\text{fork} \; $ \; \text{do} \; \text{put} \; \text{num} \; 4
\]
\[
\text{v} \leftarrow \text{get} \; \text{num}
\]
\[
\text{return} \; \text{v}
\]
Deterministic programs that single-assignment forbids

```haskell
p :: Par Int
p = do
  num <- new
  fork $ do
    put num 4
  fork $ do
    put num 4
  v <- get num
  return v
```

```bash
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```
Deterministic programs that single-assignment forbids

```hs
p :: Par Int
p = do
  num <- new
  fork $ do
    put num 4
  fork $ do
    put num 4
  v <- get num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```
Deterministic programs that single-assignment forbids

```haskell
p :: Par Int
p = do
  num <- new
  fork $ do
    put num 4
  fork $ do
    put num 4
  v <- get num
  return v
```

```
./repeated-4-ivar +RTS -N2
repeated-4-ivar: multiple put
```
Deterministic programs that single-assignment forbids

```
\begin{verbatim}
\textbf{p :: Par Int}
p = do
  num <- new
  fork $ do put num 4  \textcolor{red}{\textcircled{2}}
  fork $ do put num 4  \textcolor{red}{\textcircled{2}}
  v <- get num
  return v
\end{verbatim}
```

```
\texttt{./repeated-4-ivar +RTS -N2 repeated-4-ivar: multiple put}
```

```
\begin{verbatim}
\textbf{do}
  fork $ do insert t "0"
  fork $ do insert t "1100"
  fork $ do insert t "1111"
  v <- get t
  return v
\end{verbatim}
```

```
\begin{tikzpicture}
  \node (root) at (0,0) {1};
  \node (1) at (-1,-1) {1};
  \node (0) at (1,-1) {0};
  \node (10) at (-2,-2) {1}
  \node (11) at (0,-2) {1}
  \node (110) at (2,-2) {0}
  \node (1100) at (-1,-3) {1}
  \node (1101) at (1,-3) {0}
  \draw (root) -- (1);
  \draw (root) -- (0);
  \draw (1) -- (10);
  \draw (1) -- (11);
  \draw (0) -- (110);
  \draw (0) -- (1100);
  \draw (10) -- (1100);
  \draw (11) -- (1100);
\end{tikzpicture}
```
Deterministic programs that single-assignment forbids

\[
p :: \text{Par Int}
p = \text{do}
\text{num} \leftarrow \text{new}
\text{fork} \ _\$ \ do \ \text{put} \ \text{num} 4
\text{fork} \ _\$ \ do \ \text{put} \ \text{num} 4
\text{v} \leftarrow \text{get} \ \text{num}
\text{return} \ v
\]

\[
./\text{repeated-4-ivar} \ +\text{RTS} \ -N2
\text{repeated-4-ivar: multiple put}
\]

\[
\text{do}
\text{fork} \ _\$ \ do \ \text{insert} \ t \ "0"
\text{fork} \ _\$ \ do \ \text{insert} \ t \ "1100"
\text{fork} \ _\$ \ do \ \text{insert} \ t \ "1111"
\text{v} \leftarrow \text{get} \ t
\text{return} \ v
\]
LVars: Multiple least-upper-bound writes

Raises an error, since $3 \sqcup 4 = \top$

```plaintext
do
  fork $ do$ put num 3
  fork $ do$ put num 4
```

Works fine, since $4 \sqcup 4 = 4$

```plaintext
do
  fork $ do$ put num 4
  fork $ do$ put num 4
```
LVars: Threshold reads

```plaintext
do
    nn <- newPair
    fork $ do putFst nn 0
    fork $ do putSnd nn 1
    v <- getSnd nn
    return v -- returns 1
```

Diagram:

```
\( nn \)
```

```
(0, 0) (0, 1) ... (1, 0) (1, 1) ...
```

```
(⊥, 0) (⊥, 1) ... (0, ⊥) (1, ⊥) ...
```

```
getSnd "tripwire"
```
LVars: Threshold reads

nn

\[
\begin{array}{c}
(0, 0) \quad (0, 1) \quad \ldots \quad (1, 0) \quad (1, 1) \\

(\bot, 0) \quad (\bot, 1) \quad \ldots \\
(0, \bot) \quad (1, \bot) \\
\end{array}
\]

\[
\text{do}
\begin{align*}
nn & \leftarrow \text{newPair} \\
\text{fork} & \quad \text{do} \quad \text{putFst} \nn 0 \\
\text{fork} & \quad \text{do} \quad \text{putSnd} \nn 1 \\
v & \leftarrow \text{getSnd} \nn \\
\text{return} & \quad v \quad -- \quad \text{returns} \quad 1
\end{align*}
\]
LVars: Threshold reads

```
do
    nn <- newPair
    fork $ do putFst nn 0
    fork $ do putSnd nn 1
    v <- getSnd nn
    return v  -- returns 1
```
LVars: Threshold reads

\[
\text{getSnd} \quad "\text{tripwire}"
\]

\[
\text{do}
\]

\[
\begin{align*}
nn & \leftarrow \text{newPair} \\
\text{fork} & \; \text{do} \; \text{putFst} \; \text{nn} \; 0 \\
\text{fork} & \; \text{do} \; \text{putSnd} \; \text{nn} \; 1 \\
v & \leftarrow \text{getSnd} \; \text{nn} \\
\text{return} & \; v \quad -- \quad \text{returns} \; 1
\end{align*}
\]
LVars: Threshold reads

do
  nn <- newPair
  fork $ do putFst nn 0
  fork $ do putSnd nn 1
  v <- getSnd nn
  return v -- returns 1
LVars: Threshold reads

```
getSnd "tripwire"

do
 nn <- newPair
 fork $ do putFst nn 0
 fork $ do putSnd nn 1
 v <- getSnd nn
 return v -- returns 1
```
LVars: Threshold reads

\[
\text{do} \quad \\
\begin{align*}
nn &\leftarrow \text{newPair} \\
\text{fork} &\quad \text{do} \quad \text{putFst} \nn 0 \\
\text{fork} &\quad \text{do} \quad \text{putSnd} \nn 1 \\
v &\leftarrow \text{getSnd} \nn \\
\text{return} &\quad v \quad \quad \text{-- returns 1}
\end{align*}
\]
LVars: Threshold reads

```
do
  nn <- newPair
  fork $ do putFst nn 0
  fork $ do putSnd nn 1
  v <- getSnd nn
  return v -- returns 1
```
LVars: Threshold reads

The threshold set must be pairwise incompatible.

```
\texttt{do}
\begin{align*}
\texttt{nn} & \leftarrow \texttt{newPair} \\
\texttt{fork} & \; \texttt{do} \; \texttt{putFst nn 0} \\
\texttt{fork} & \; \texttt{do} \; \texttt{putSnd nn 1} \\
\texttt{v} & \leftarrow \texttt{getSnd nn} \\
\texttt{return} & \; \texttt{v} -- \texttt{returns 1}
\end{align*}
```
Overlapping writes are no problem

```do
fork $ do insert t "0"
fork $ do insert t "1100"
fork $ do insert t "1111"
v <- get t
return v
```
Monotonicity enables deterministic parallelism

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 schemes 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 system programming and the design of operating systems.

There is a wide disagreement among system designers as to what are the best primitives for writing system 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 in 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 Integar or channel are declared at line (1), and for any simple type T, a channel is declared at line (2). Two processes f and g are simulated, much like procedures. Aside from usual parameters (passed by value in this example, like OUT at line (3)), we can declare in the heading of the process how it is linked to other processes: at line (4) f is stated to communicate via two input lines that can carry integers, and one similar output line. The body of the process is an usual Algol program except for invocation of unit input lines (e.g., at (6)) or send a variable on a line of compatible type (e.g., at (7)). The process starts blocked on a unit input line, and remains so until it receives a signal from 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 (8), where the actual names of the channels are bound to the formal parameters of the processes. The info operator permits the concurrent activation of the processes. Such a style of programming is close to any system using EVENT mechanisms (11), (12), (13), (14). A pictorial representation of the program is the schema P in Fig. 2, where the nodes represent processes and the arcs communication channels between these processes.

The sort of things we would like to prove on a program line S is: Firstly, that all processes in S run forever. Secondly, more precisely, that S prints out bit 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 means of communication lines, and they exchange information through these lines. A given station computes on data coming along its input lines,

Kahn, 1974
Monotonicity enables deterministic parallelism

$f$ is monotonic iff, for a given $\leq$,

$$x \leq y \implies f(x) \leq f(y)$$
Monotonicity enables deterministic parallelism

Monotonicity 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: a machine need not have all of its input to start computing, since future input concerns only future output.

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

Kahn, 1974
Challenge problem

In a directed graph:

- find the connected component of all nodes within $k$ hops of a vertex $v$
- and compute a function $analyze$ over each vertex in that component
- making the set of results available asynchronously to other computations
Challenge problem

- We compared two implementations:
  - Control.Parallel.Strategies
  - Our prototype LVar library (tracking visited nodes in an LVar)
- Level-sync breadth-first traversal, $k = 10$
- Random graph; 320K edges; 40K nodes
- Varying:
  - number of cores
  - amount of work done by analyze
Challenge problem: Strategies vs. LVars
(lower is better)
Challenge problem: Strategies vs. LVars

(lower is better)
Challenge problem: Strategies vs. LVars

Monotonicity 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: a machine need not have all of its input to start computing, since future input concerns only future output.

- Average time from start of program to first invocation of analyze:
  - Strategies version: 64.64 ms
  - LVVar version: 0.18 ms
Lots more stuff in the paper and TR

- $\lambda_{\text{LVar}}$, a core calculus for LVars
- Proof of determinism for $\lambda_{\text{LVar}}$
  - Independence Lemma is a frame property!
- Details on our Haskell library
- Proposed extension: quasi-determinism
  - Ask me about our new paper on this!
  - Cool applications: $k$-CFA, phylogenetics, ...
Thank you!

Email: lkuper@cs.indiana.edu
Research blog: composition.al
Project repo: github.com/iu-parfunc/lvars
Code from this talk: github.com/lkuper/lvar-examples

Photo by kakadu on Flickr. Thanks!