LVars for Parallel Programming

Isaiah Weating and Lindsey Kuper

School of Informatics and Computing, Indiana University

SCHOOL OF INFORMATICS AND COMPUTING
INDIANA UNIVERSITY
Bloomington

Introduction

One of the most challenging parts of parallel programming is dealing with shared state between computations. A program could behave differently on different runs, depending on the ordering of state-changing operations. This is called nondeterminism. Our goal is to create a language that will allow a program to run various operations in parallel while still remaining deterministic.

In this project, we worked on extending lambdaLVar, a small programming language that uses LVars. An LVar is a variable that can take on various states that form a lattice. Figure 1 shows some examples of typical LVars. The lambdaLVar language is actually not just one language, but a family of languages, depending on the user-specified lattice.

In this project we looked at some ways to instantiate lambdaLVar with new lattices. To instantiate lambdaLVar with a lattice, one has to specify a "least upper bound" (lub) operation that specifies how two lattice elements should be combined together. This lub operation corresponds to how, when two threads write to a variable, the writes done by the two threads are merged together after the two threads finish running.

For some lattices, the least upper bound operation was a simple built-in function like taking the maximum of two numbers; for other lattices it was more sophisticated and required us to write our own function.

We instantiated lambdaLVar in a variety of ways, and for each instantiation, we created test suites to check if the programs we wrote were working the way we intended.

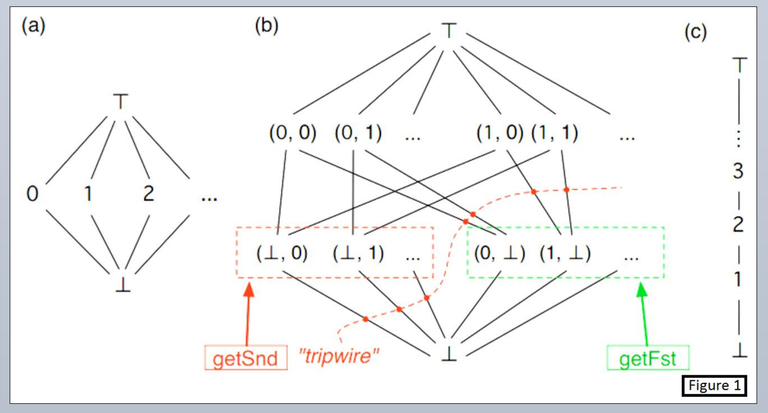


Figure 1

Lattices and LVars

The counter LVar, figure 1c, keeps track of the max value that has been entered. If the user were to put in values x and y into the LVar, it would keep track of max(x, y). If the user were to then use the get function on any value up to max(x, y), it would return that value. If they were to call get with a value greater than max(x, y), the call would block the program until the value asked for is reached.

The pair LVar, figure 1b, allows for pairs of single-assignment values. This LVar only allows for one write to each index. The order in which the program puts in the values isn't important, allowing for writing both values in parallel while still maintaining determinism.

State and Nondeterminism

```
let par a = ...
            let par x = ...
                        y = ...
             in ...
        b = ...
in ...
```

```
let par a = ...
            let par x = ... Put(1)
...
                        y = ...
             in ...
        b = ... Get(1) ...
in ...
```

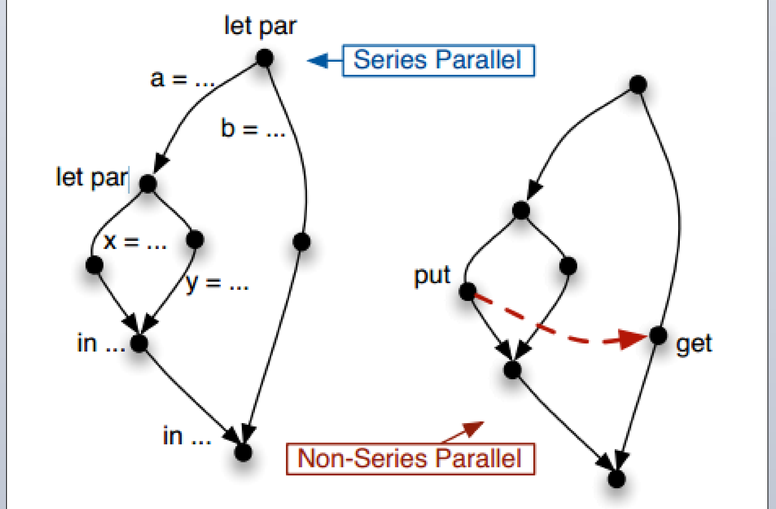
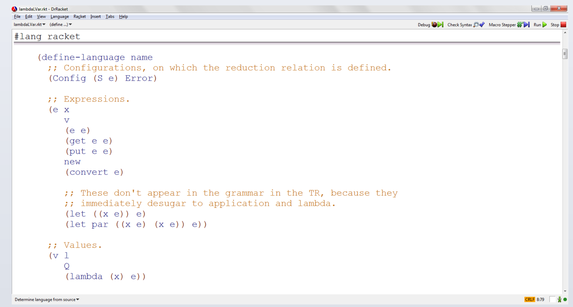


Figure 2

Figure 2 shows two parallel program dependence graphs. The one on the left will always be deterministic since it doesn't have any put or get operations. The one on the right, though, will be nondeterministic since the put and get could evaluate simultaneously or in an arbitrary order, causing nondeterminism. LVars address this problem by insuring the program behaves the same way regardless of the order of the put and get.

Tools Used

We developed our language in Redex. Redex is a framework for creating programming languages which is embedded in the Racket language. Our development software of choice was DrRacket, shown below.



```
#lang racket

(define-language name
  ;; Configurations, on which the reduction relation is defined.
  (Config (S e) Error)

  ;; Expressions.
  (e x
     v
     (e e)
     (get e e)
     (put e e)
     new
     (convert e)

     ;; These don't appear in the grammar in the TR, because they
     ;; immediately desugar to application and lambda.
     (let ((x e)) e)
     (let par ((x e) (x e)) e))

  ;; Values.
  (v l
     Q
     (lambda (x) e))
```

Future Work

Future work on lambdaLVar will include:

- Automatic test generation: Currently we manually write tests for lambdaLVar. In the future, we will exploit the random test generation features of Redex to automate some of this work for us.
- Decoupling lambdaLVar from Redex: lambdaLVar programs can be cumbersome to work with because one has to write them inside the Redex framework. In the future, we will write lambdaLVar programs independently from Redex.

Future work on LVars in general will include:

- There is still much work to be done on effective ways of programming with LVars beyond the special-purpose lambdaLVar language, and eventually, we want to build libraries of efficient concurrent data structures on top of LVars. (Ask Lindsey for more details on the current roadmap for LVars!)

Contacts

Isaiah Weating --- iweating@indiana.edu
Lindsey Kuper – lkuper@cs.indiana.edu