Principles of Programming Languages Lecture 5: Exploring non-determinism in K. Threads.

Andrei Arusoaie[1]

[1]Department of Computer Science

October 31, 2017

Outline

Exploring non-determinism in K

Outline

Non-determinism

- What non-determinism means?

Non-determinism

- What non-determinism means?
- In programs: multiple possible executions

Non-determinism

- What non-determinism means?
- In programs: multiple possible executions
- Examples?

Non-determinism

- What non-determinism means?
- In programs: multiple possible executions
- Examples?
  - In IMP: Binary expressions combined with side-effects

Non-determinism

- What non-determinism means?
- In programs: multiple possible executions
- Examples?
  - In IMP: Binary expressions combined with side-effects
- Example: y = `++x / (++x / x) ;`

Non-determinism

- What non-determinism means?
- In programs: multiple possible executions
- Examples?
  - In IMP: Binary expressions combined with side-effects
- Example: y = `++x / (++x / x) ;`
  - Exercise: what's the value of `y` if `x = 1`?

Non-determinism

- What non-determinism means?
- In programs: multiple possible executions
- Examples?
  - In IMP: Binary expressions combined with side-effects
- Example: y = `++x / (++x / x) ;`
  - Exercise: what's the value of `y` if `x = 1`?
- DEMO: `krun`

In K

- By default, `krun` shows only *one* solution
- Reason: exploring all executions might take too long

In K

- By default, `krun` shows only *one* solution
- Reason: exploring all executions might take too long
- In fact, `kompile` generates an interpreter which picks one possible execution path

In K

- By default, `krun` shows only *one* solution
- Reason: exploring all executions might take too long
- In fact, `kompile` generates an interpreter which picks one possible execution path
- Enable non-determinism exploration:

```
kompile <file> --transition <tag>
```

Example

Recall y = `++x / (++x / x);`

Example

Recall y = `++x / (++x / x) ;`

Steps:

1. Tag the division syntax production:

```
syntax AExp ::= AExp "/" AExp [left, strict, division]
```

Example

Recall y = `++x / (++x / x) ;`

Steps:

1. Tag the division syntax production:

```
syntax AExp ::= AExp "/" AExp [left, strict, division]
```

2. Compile the defintion:

```
:~$ kompile imp.k --transition division
```

Example

Recall y = `++x / (++x / x) ;`

Steps:

1. Tag the division syntax production:

```
syntax AExp ::= AExp "/" AExp [left, strict, division]
```

2. Compile the defintion:

```
:~$ kompile imp.k --transition division
```

3. Run:

```
:~$ krun test.imp --search
```

Example

Recall y = `++x / (++x / x) ;`

Steps:

1. Tag the division syntax production:

```
syntax AExp ::= AExp "/" AExp [left, strict, division]
```

2. Compile the defintion:

```
:~$ kompile imp.k --transition division
```

3. Run:

```
:~$ krun test.imp --search
```

DEMO

Threads in IMP

- What are threads?

Threads in IMP

- What are threads?
- The smallest sequence of instructions that are managed by the OS's scheduler

Threads in IMP

- What are threads?
- The smallest sequence of instructions that are managed by the OS's scheduler
- Thread vs. Process:
  - multiple threads can exist in one process
  - threads run in a shared memory space
  - processes run in separate memory spaces

Threads in IMP

- What are threads?
- The smallest sequence of instructions that are managed by the OS's scheduler
- Thread vs. Process:
  - multiple threads can exist in one process
  - threads run in a shared memory space
  - processes run in separate memory spaces
- Recall `fork`: creates a new child process
- Both the caller and the child will execute the instruction right after `fork` system call

Threads in IMP

- What are threads?
- The smallest sequence of instructions that are managed by the OS's scheduler
- Thread vs. Process:
  - multiple threads can exist in one process
  - threads run in a shared memory space
  - processes run in separate memory spaces
- Recall `fork`: creates a new child process
- Both the caller and the child will execute the instruction right after `fork` system call
- In our language we will use `spawn`:
  - it takes a statement and creates a new concurrent thread
  - the memory is shared with the parent thread

Threads in IMP

- What are threads?
- The smallest sequence of instructions that are managed by the OS's scheduler
- Thread vs. Process:
  - multiple threads can exist in one process
  - threads run in a shared memory space
  - processes run in separate memory spaces
- Recall `fork`: creates a new child process
- Both the caller and the child will execute the instruction right after `fork` system call
- In our language we will use `spawn`:
  - it takes a statement and creates a new concurrent thread
  - the memory is shared with the parent thread
- Demo: `syntax Stmt ::= "spawn" Stmt`
- Demo: write a program

spawn

- `spawn S`: create a new concurrent thread that executes `S`

`spawn`

- `spawn S`: create a new concurrent thread that executes `S`
- The new thread is being passed at creation time its parent's environment
- It shares with its parent the memory locations

`spawn`

- `spawn S`: create a new concurrent thread that executes `S`
- The new thread is being passed at creation time its parent's environment
- It shares with its parent the memory locations
- The parent and the child threads can evolve unrestricted
  - they can change their own environments
  - declare or hide variables
  - create new threads, etc.

Configuration Abstraction - again

Configuration Abstraction - again

- The above suggests that a thread should have its own `<k>`, `<env>`, and `<stack>` cells

Configuration Abstraction - again

- The above suggests that a thread should have its own `<k>`, `<env>`, and `<stack>` cells
- Configuration refinement:

```
<T>
  <threads>
   <thread>
     <k> $PGM:Stmt </k>
     <env> .Map </env>
     <stack> .List </stack>
   </thread>
  </threads>
  <store> .Map </store>
  <in stream="stdin"> .List </in>
  <out stream="stdout"> .List </out>
</T>
```

Configuration Abstraction - again

- The above suggests that a thread should have its own `<k>`, `<env>`, and `<stack>` cells
- Configuration refinement:

```
<T>
  <threads>
   <thread>
     <k> $PGM:Stmt </k>
     <env> .Map </env>
     <stack> .List </stack>
   </thread>
  </threads>
  <store> .Map </store>
  <in stream="stdin"> .List </in>
  <out stream="stdout"> .List </out>
</T>
```

- What if we run the semantics with the new configuration?

Configuration Abstraction - again

- The above suggests that a thread should have its own `<k>`, `<env>`, and `<stack>` cells
- Configuration refinement:

```
<T>
  <threads>
   <thread>
     <k> $PGM:Stmt </k>
     <env> .Map </env>
     <stack> .List </stack>
   </thread>
  </threads>
  <store> .Map </store>
  <in stream="stdin"> .List </in>
  <out stream="stdout"> .List </out>
</T>
```

- What if we run the semantics with the new configuration?
- Demo: look at the rules and show how they should be written.

Multiplicity

Multiplicity

- We should be able to create multiple threads

Multiplicity

- We should be able to create multiple threads
- Thus, we specify this in the configuration:

```
<T>
  <threads>
   <thread multiplicity="*">
     <k> $PGM:Stmt </k>
     <env> .Map </env>
     <stack> .List </stack>
    </thread>
  </threads>
  <store> .Map </store>
  <in stream="stdin"> .List </in>
  <out stream="stdout"> .List </out>
</T>
```

Multiplicity

- We should be able to create multiple threads
- Thus, we specify this in the configuration:

```
<T>
  <threads>
   <thread multiplicity="*">
     <k> $PGM:Stmt </k>
     <env> .Map </env>
     <stack> .List </stack>
    </thread>
  </threads>
  <store> .Map </store>
  <in stream="stdin"> .List </in>
  <out stream="stdout"> .List </out>
</T>
```

- Now, we can start giving semantics to `spawn`.

Sematics of `spawn`

- First, the rule for `spawn`:

```
// spawn
rule <k> (spawn S => .) ...</k> <env> Rho </env>
     (.Bag => <thread>...
                    <k> S </k> <env> Rho </env>
                ...</thread>)
```

Sematics of `spawn`

- First, the rule for `spawn`:

```
// spawn
rule <k> (spawn S => .) ...</k> <env> Rho </env>
     (.Bag => <thread>...
                    <k> S </k> <env> Rho </env>
                ...</thread>)
```

- Note that the configuration abstraction algorithm fills the missing cells!

Execute programs with threads

- Append rule tags: DEMO.

Execute programs with threads

- Append rule tags: DEMO.

```
:-$ kompile <file> --transition "<tags>"
```

Execute programs with threads

- Append rule tags: DEMO.

```
:-$ kompile <file> --transition "<tags>"
```

```
:-$ krun <file> --search
```

Thread termination

- Thread termination rule:

Thread termination

- Thread termination rule:

```
rule (<thread> <k> . </k> ...</thread> => .Bag)
```

Thread termination

- Thread termination rule:

```
rule (<thread> <k> . </k> ...</thread> => .Bag)
```

- DEMO

Challenge/Exercise:

`halt`: a statement that ends the execution immediately

Lab this week

- Continue your work on your language definition.