Monadic Composition for Deterministic, Parallel Batch Processing

Ryan Scott\textsuperscript{1}  
Ryan Newton\textsuperscript{1}  
Omar Navarro Leija\textsuperscript{2}  
Joe Devietti\textsuperscript{2}

\textsuperscript{1}Indiana University  
\textsuperscript{2}University of Pennsylvania

rgscott@indiana.edu  
github.com/RyanGlScott
Unintended nondeterminism sucks.
Same commit, different results #770

Closed amitaibu opened this issue on Nov 19, 2012 · 2 comments

amitaibu commented on Nov 19, 2012

I have a build that was successful, but after "rebuilding" it fails.

Success VS Fail -- so I assume it's related to the environment?
Can't reproduce examples #21

Can't reproduce nested grid method from Examples #621

Can't Reproduce the Accuracy for "Pre-trained word embeddings in Keras" Example #5826

Same commit, different results #770

amitaibu commented on Nov 19, 2012

I have a build that was successful, but after "rebuilding" it fails.

Success VS Fail -- so I assume it's related to the environment?
all: create-bindir install-exec-local

DESTDIR=foo
bindir=bar

install-exec-local:
    cd $(DESTDIR)/$(bindir) && ls

create-bindir:
    mkdir -p $(DESTDIR)/$(bindir)
ryanglscott at T450-Linux in ~/.../sandbox/testing/123
$ make -j1
mkdir -p foo/bar
cd foo/bar && ls
ryanglscott at T450-Linux in ~/.../sandbox/testing/123
$
ryanglscott at T450-Linux in ~/.../sandbox/testing/123
$ make -j2
mkdir -p foo/bar
cd foo/bar && ls
ryanglscott at T450-Linux in ~/.../sandbox/testing/123
$
ryanglscott at T450-Linux in ~/.../sandbox/testing/123
$ make -j2
mkdir -p foo/bar
cd foo/bar && ls
/bin/sh: 1: cd: can't cd to foo/bar
Makefile:7: recipe for target 'install-exec-local' failed
make: *** [install-exec-local] Error 2
make: *** Waiting for unfinished jobs....
ryanglscott at T450-Linux in ~/.../sandbox/testing/123
$
all: create-bindir install-exec-local

DESTDIR=foo
bindir=bar

install-exec-local:
  cd $(DESTDIR)/$(bindir) && ls
create-bindir:
  mkdir -p $(DESTDIR)/$(bindir)
Race condition!

dir: create-bindir install-exec-local

DESTDIR=foo
bindir=bar

install-exec-local:
  cd $(DESTDIR)/$(bindir) && ls

create-bindir:
  mkdir -p $(DESTDIR)/$(bindir)
The detflow guarantee

If a program is invoked under detflow twice with identical inputs, and given sufficient machine resources to complete, then both invocations will produce the same output.
Quasideterminism

Arguments
Input files

Output files
1011001

detflow program
Quasideterminism

Arguments

Input files

detflow program

Arguments

Input files

Exn 1
Quasideterminism

```
Arguments
Input files

detflow program

Arguments
Input files

Exn 1
Exn 2
```
Nondeterministic program

Arguments

Input files

Output files v1

Output files v2

...
Nondeterministic program

Arguments
Input files
Thread scheduling
Environment leaks
Nondet. system calls/CPU instructions

Output files v1
Output files v2
...
Arguments
Input files

Thread scheduling
Environment leaks
Nondet. system calls/CPU instructions
detflow program

Arguments
Input files

Thread scheduling
Environment leaks
Nondet. system calls/CPU instructions

Arbitrary executable (untrusted)

Output files
detflow
detflow

Haskell
Fork-join parallelism

detflow

Haskell
detflow

Fork-join parallelism

Haskell

LVars
detflow

- Fork-join parallelism
- LVars

OS process 1

...  

OS process n
detflow

Fork-join parallelism

Haskell

LVars

libdet

OS process 1

...

OS process n
detflow

Fork-join parallelism

Haskell

LVars

libdet

OS process 1

... 

OS process n
Traditional Haskell program

main :: IO ()
main :: IO ()
-- ^ Lots of ways to sneak in
-- nondeterminism!
detflow Haskell programs

main :: DetIO ()
data DetIO a -- Abstract
data DetIO a -- Abstract
-- Expose only deterministic API calls
getLine :: DetIO String
putStrLn :: String -> DetIO ()
-- etc.
data DetIO a -- Abstract
-- Expose only deterministic API calls
getLine :: DetIO String
putStrLn :: String -> DetIO ()
-- etc.

Key idea: Only expose deterministic operations that can be composed in a deterministic fashion
data DetIO a -- Abstract
-- Expose only deterministic API calls
getLine :: DetIO String
putStrLn :: String -> DetIO ()
-- etc.

main :: DetIO ()
main = do x <- getline
          system ("gcc -c " ++ x)
          putStrLn x
Parallel file access

- `detflow` uses the filesystem as a mutable, shared store
- Should this be allowed?

```haskell
readFile :: FilePath -> DetIO String
writeFile :: FilePath -> String
            -> DetIO String
fork      :: DetIO a -> DetIO (Thread a)
```
Problem: racing file access

Thread 1

do writeFile "foo.txt"
  "Hello, World"

Thread 2

do foo <- readFile "foo.txt"
  if foo == "Hello, World"
    then ... 
    else ...

Problem: racing file access

Thread 1

\[
\text{do writeFile "foo.txt" "Hello, World"}
\]

Thread 2

\[
\text{do foo <- readFile "foo.txt" }
\text{if foo == "Hello, World" }
\text{then ...}
\text{else ...}
\]
Solution: permissions

- Every thread holds separate permissions on system filepaths
Solution: permissions

- Every thread holds separate permissions on system filepaths

```
/abcdef/ghijkl/mnopqr
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 2</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R</td>
<td>R</td>
</tr>
</tbody>
</table>
Parallel file access, revisited

data Perm -- (R/RW) + path

forkWPerms :: [PathPerm] -> DetIO a
        -> DetIO (Thread a)

• readFile and writeFile must respect the permissions in a thread’s local state
system shell calls

system :: String -> DetIO ()

main :: DetIO ()
main = system "gcc foo.c -o foo"
libdet

libdet must intercept potential sources of nondeterminism at runtime.
libdet must intercept potential sources of nondeterminism at runtime.

Reading from “banned” directories

- /dev/urandom
- /proc
- etc.
libdet

libdet must intercept potential sources of nondeterminism at runtime.

Reading from “banned” directories

- /dev/urandom
- /proc
- etc.

Solution

- Intercept calls to `fopen()` (with `LD_PRELOAD`), error if they read anything blacklisted
Uncontrolled concurrency

- e.g., with pthreads
Uncontrolled concurrency

- e.g., with pthreads

Solution

- Intercept calls to `pthread_create()` (with `LD_PRELOAD`) to run everything sequentially
libdet must intercept potential sources of nondeterminism at runtime.

**Nondeterministic OS properties**

- e.g., reading addresses returned by `mmap()`
libdet must intercept potential sources of nondeterminism at runtime.

Nondeterministic OS properties

- e.g., reading addresses returned by `mmap()`

Solution

- Disable address-space layout randomization (ASLR)
libdet must intercept potential sources of nondeterminism at runtime.

Path operations with insufficient permissions

- e.g., reading /foo without read permissions on /foo
libdet must intercept potential sources of nondeterminism at runtime.

Path operations with insufficient permissions

- e.g., reading /foo without read permissions on /foo

Solution

- Inherit permissions from detflow!
Bioinfo. apps, parallel speedup

(Higher is better)
Future work

- Reach closer to catching *all* sources of nondeterminism in runtime
- Dynamic (at-runtime) checkout of permissions
- Make more programs feasible to determinize
detflow can be used to construct and run parallel batch processing jobs deterministically (including legacy binaries) with less than 5% overhead.

Approach:

- Statically-typed root process: allows multithreading
- Each thread may shell out to legacy binaries: internally sequentialized by sandbox
- Legacy binaries can create subprocesses: also sequentialized
- Each thread and subprocess holds distinct file system permissions to prevent races

https://github.com/iu-parfunc/detmonad