LECTURE 24

Theory and Design of PL (CS 538)

April 20, 2020

NEWS

HW5 WRAPUP

- An enormous pain in the ass
- Why go through all of this trouble?
  - Ownership: rule out most memory leaks
  - Aliasing: make totally unsafe stuff safe

BUILD A TREE THAT...

- Gives clients pointers to internal tree memory
- Lets client write whenever and whatever they want
    - With no runtime checks (direct memory write)
    - While never segfaulting or breaking the tree
- Also build an iterator handing out pointers all over

```rust
impl<K, V> TreeMap<K, V> {
    // Wildly unsafe
    pub fn get_mut(&mut self, key: &K) -> Option<&mut V> { ... }
}
```

HW5: FEEDBACK?

HW6 OUT

- Concurrency: making things go faster
- We give you: a slow, single-threaded version
- You make: multi-threaded versions in two ways
- Should be much less grappling with borrow checker
  - But still a bit (The Rust Programming Language)

Start early, especially if you haven't tried writing concurrent code before

MODELING CONCURRENCY

MANY ASPECTS

- Parallelism and simultaneous execution
- Message-passing and channels
- Shared memory and locking
- Threads blocking/waiting

PROCESS CALCULUS

- Mathematical model of message passing
- Many flavors, developed in 1970s and 1980s
  - Communicating sequential processes (CSP)
  - Communicating concurrent systems (CCS)
  - Pi-calculus
- By Tony Hoare, Robin Milner, and many others

OUR VERSION

1. Simple arithemtic expressions
2. *Channels*: named pipes for communication
3. *Processes*: send/receive along channels

PROCESS CALCULUS: GRAMMAR

ARITHMETIC EXPRESSIONS

```
num = "0" | "1" | "2" | ...

var = "x" | "y" | "z" | ...

exp = num | var | exp "+" exp | exp "*" exp | ...
```

- Arithmetic expressions with variables
- Examples of arithmetic expressions:
  - 42
  - 5 * 3
  - 2 + 0 + z

CHANNELS

```
chn = "A" | "B" | "C" | ...
```

- Addresses to send to/receive from
- Different names = different addresses
- We'll use two special channels:
  - `I`: input channel into program
  - `O`: output channel from program

PROCESSES

```
prc = "done"                                  (* do nothing         *)
    | "make" chn "in" prc                     (* make new channel   *)
    | "send" exp "->" chn "then" prc          (* send a message     *)
    | "recv" var "<-" chn "then" prc          (* receive a message *)
    | "[" exp "<" exp "]" prc                 (* run if guard true *)
    | prc "+" prc                             (* run this or that  *)
    | prc "|" prc                             (* run in parallel   *)
```

- Make new channel, send and receive along channel
- Combine several processes together
  - Select between different processes
  - Run processes in parallel

EXAMPLES (BLACKBOARD)

OPERATIONAL SEMANTICS

MAIN SETUP

- Define how processes step $P \rightarrow Q$
- New addition: each transition may have a *label*
- Labels model sending and receiving
  - (A, n): send num n along channel A
  - $(\bar{A}, n)$: receive num n from channel A
- Other steps: no label (silent transitions)

BLACKBOARD (OR WRG)

EXTENSION: RECURSION

WHY RECURSION?

- So far: finite number of steps
- Some processes live forever (e.g., servers)
- Extend the language with *recursive processes*

SYNTAX

```
name = "P1" | "P2" | "P3" | ...   (* names of processes *)

prc = ...
    | name                        (* process could be a name *)

def = name "=" prc                (* definition of processes *)
```

- Add *process names* and *recursive definitions*

EXAMPLES

OPERATIONAL SEMANTICS

- Just add one more rule to unfold definitions…

A TINY GLIMPSE OF ERLANG

JOE ARMSTRONG

- Passed away in 2019 :(
- Invented Erlang while working for Ericsson
- Hugely influential views on computing
  - Take a look at his thesis
  - Or check out some of his talks

PRINCIPLE 1: PROCESSES

- Take idea of process from OS
  - Not threads: no shared memory space!
  - Separate program into several processes
- Erlang: processes are cheap
  - Can make millions of processes
  - So-called “green threads”
- Rust: heavier, OS threads (can’t have so many)
  - Used to have green threads, taken out

PRINCIPLE 2: ISOLATION

- Communicate only by message passing
- A fault in one process should be contained
- Share nothing concurrency
  - Also known as the Actor model

PRINCIPLE 3: LET IT CRASH

- Will never be able to eliminate all faults
- Instead: plan for faults to happen
- If a process hits an error, just crash it
  - Don’t make things worse
- Let someone other process fix/restart

THE ERLANG LANGUAGE

- Designed for telecom applications
  - Soft real-time, highly reliable
- Designed for processes that live forever
  - Can swap in code updates live
- At the core: processes, messages, isolation

BIG IMPACT

- Runs Ericsson telecom switches
  - Handles estimated 50% of all cell traffic
  - OTP libraries, Open Telecom Protocol
- Runs Whatsapp and FB chat (previously)
  - Whatsapp: 50 employees for 900M users (2015)
- Many successful applications
  - CouchDB, Riak, Elixir, …

SPAWNING PROCESSES

```
my_proc = fun() -> 2 + 2 end.
p_id = spawn(my_proc).
```

- Just like in Rust: pass it a closure

SENDING MESSAGES

```
p_id ! hello.
self() ! there.
```

- Asynchronous channels: send never blocks
- Send directly to process, not to specific channel

RECEIVING MESSAGES

```
dolphin() ->
  receive
    do_a_flip ->
      io:format("How about no?~n");
    fish ->
      io:format("So long and thanks for all the fish!~n");
    _ ->
      io:format("Heh, we're smarter than you humans.~n")
  end.
```

- Do a case analysis on received message
- Each process has one incoming queue, like a mailbox

JOE'S FAVORITE PROGRAM

- Universal Server: can turn into any another process

```
universal_server() ->
  receive
    {become, New_proc} ->
      New_proc()
  end.
```

- Lowercase match on string, uppercase variable

FACTORIAL SERVER

- Wait for message, respond with factorial

```
factorial_server() ->
  receive
    {From, N} ->
      From ! factorial(N),
      factorial_server()
  end.

factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).
```

BECOMING FACTORIAL

- Turn a universal server into a factorial server

```
main() ->
  univ_pid = spawn(fun universal_server/0),
  univ_pid ! {become, fun factorial_server/0},
  univ_pid ! {self(), 50},
  receive
    Response -> Response
  end.
```

- `/0` means zero arguments (Erlang dynamically typed)