Rust Lab - Register Machines and Godel Numberings

Intro to Rust Course

Jordan Hall - jh4020@ic.ac.uk

21/12/21

Summary

Register Machines are covered in the COMP50003 Models of Computation course. They are simple machines that only make use of 3 unique instructions, yet are Turing Complete. In this task, we will deal with programming several tools that automate common tasks.

Register Machines

Register Machines are an abstract class of machines. They are Turing Equivalent and as a result, are useful proxies for proving statements about computability. The Register Machines in the course have 3 unique instructions:

- $R_n^+ \to L_m$ Increment register R_n by one and jump to instruction L_m
- $R_n^- \to L_m, L_p$ If register $R_n > 0$ then decrement it by one and jump to instruction L_m else jump to instruction L_p
- HALT Stop the execution of the machine

It's worth noting that if a jump to an instruction that is undefined occurs, then it's considered a signal to halt the machine's execution.

Godel Numbering

Godel Numbering can be used to create bijections between Register Machine programs and natural numbers.

Namely, we use two different pair encodings: pub $\langle\langle x, y\rangle\rangle \stackrel{\text{def}}{=} 2^x(2y+1)$ which is a bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}^+$

$\langle x, y\rangle \stackrel{\text{def}}{=} 2^x(2y+1) - 1$ which is a bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$

It is worth noting that $2^x(2y+1)$ has x least-significant bits as zeros. This makes it easier to decode than other Godel Numbering systems that must make use of prime factorization.

We can then define a bijection between lists of naturals and natural numbers:

$\ulcorner [] \urcorner \stackrel{\text{def}}{=} 0$

$\ulcorner x :: l \urcorner \stackrel{\text{def}}{=} \langle\langle x, \ulcorner l \urcorner\rangle\rangle$

And we can define a bijection between Register Machine program instructions and natural numbers:

$\ulcorner R_i^+ \to L_j \urcorner \stackrel{\text{def}}{=} \langle\langle 2i, j\rangle\rangle$

$\ulcorner R_i^- \to L_j, L_k \urcorner \stackrel{\text{def}}{=} \langle\langle 2i+1, \langle j, k\rangle\rangle\rangle$

$\ulcorner \text{HALT} \urcorner \stackrel{\text{def}}{=} 0$

Combining the three numberings gives a method for representing any Register Machine program as a natural number.

What To Do

For this lab we want to automate the functionality described above to create a suite of useful tools.

Program Evaluation

Being able to evaluate programs on a certain set of inputs would be very useful and so we define representations of machine state and programs as such:

```rust
pub type Label = usize;
pub type Register = u128;
pub type State = (Label, HashMap<Register, u128>);

#[derive(Clone, Copy, Debug, PartialEq)]
pub enum Instruction {
    Add(Register, Label),
    Sub(Register, Label, Label),
    Halt
}
```

We rename primitive types such as usize and u128 in order to make them more readable. Note that this is just type aliasing.

As for state, or configuration, we use a tuple of the current label of the instruction being executed and a hashmap that maps registers to their value. We choose to use a hashmap as there's no guaranteed pattern of register numbering in a program (e.g. a program may use registers 1, 100, 100000, etc).

The Instruction enum is used to encode a choice between three unique instruction types. A wonderful thing about Rust is that enums can hold data much like a struct, and this data need not be the same for each enumeration. For Add, we store the Register we want to increment, and then the Label of the Instruction we jump to next. For Sub, we store the Register we want to decrement, and then the Label of the Instruction we jump to if that Register wasn't zero followed by the Label of the Instruction we want to jump to if that Register is zero.

Now we define a function that, given a list of Instructions and an initial State, returns State of the program when it halts (note that this evaluator will run indefinitely if given a non-terminating function).

As an aid, we give the function signature:

```rust
pub fn eval_program(program: &[Instruction], state: &State) -> State {
    unimplemented!();
}
```

Godel Numbering

Converting between the different representations of lists, programs, and pairs can be rather tedious. In this part, we will write some functions that convert between the representations in an easy-to-use way.
Firstly, encoding pairs of natural numbers their Godel Numbering representations is a building block that will be used to create the other converters.
Note: encode_pair1$(x, y) \stackrel{\text{def}}{=} \langle\langle x, y\rangle\rangle$ and encode_pair2$(x, y) \stackrel{\text{def}}{=} \langle x, y\rangle$

As an aid, we give the function signatures:

```rust
fn encode_pair1(x: u128, y: u128) -> u128 {
    unimplemented!();
}

fn encode_pair2(x: u128, y: u128) -> u128 {
    unimplemented!();
}
```

The next step is to encode a list of natural numbers into its Godel Numbering representation:

```rust
pub fn encode_list_to_godel(l: &[u128]) -> u128 {
    unimplemented!();
}
```

And finally, encode every Instruction in a program into it's Godel Numbering representation:

```rust
fn encode_program_to_list(program: &[Instruction]) -> Vec<u128> {
    unimplemented!();
}
```

With all of these functions, a program can be converted into a natural number. This is useful if you want to instruct a Universal Register Machine what Register Machine program to run.

However, we also would like to convert between these representations in reverse as well. If not, then what is the whole point in having these bijections!?:

```rust
fn decode_pair1(a: u128) -> (u128, u128) {
    unimplemented!();
}

fn decode_pair2(a: u128) -> (u128, u128) {
    unimplemented!();
}

pub fn decode_godel_to_list(g: u128) -> Vec<u128> {
    unimplemented!();
}

pub fn decode_list_to_program(program: &[u128]) -> Vec<Instruction> {
    unimplemented!();
}
```

Testing

For this lab, we have made some simple unit tests in order to show how unit tests are formatted, as well as test small elements of your code's functionality. You may run these tests by opening a terminal in this lab's root directory and typing:

```
$ cargo test
```

The tests given for this lab are quite sparse and do not tests edge cases. We highly encourage you to extend the test suite to account for any edge/corner cases your code] may come accross.

Extensions

For this lab we have elected to run three extensions

Extension A

Extend the code to make use of the num_bigint and num_traits crates. As of now, the maximum values that can be stored in Registers, and manipulated by the encoding/decoding functions, is $2^{128} - 1$. This is far too small and makes doing anything ***useful*** almost impossible. BigInts allow us to manipulate arbitrary-sized integers (or unsigned integers, as is for this lab) limited only to the size of your main memory.
An updated skeleton for this task would look like:

```rust
use std::collections::HashMap;
use num_bigint::BigUint;

pub type Label = usize;
pub type Register = u128;
pub type State = (Label, HashMap<Register, BigUint>);

#[derive(Clone, Copy, Debug, PartialEq)]
pub enum Instruction {
    Add(Register, Label),
    Sub(Register, Label, Label),
    Halt
}

use Instruction::*;
pub fn eval_program(program: &[Instruction], state: &State) -> State {
    unimplemented!();
}
fn encode_pair1(x: &BigUint, y: &BigUint) -> BigUint {
    unimplemented!();
}
fn encode_pair2(x: &BigUint, y: &BigUint) -> BigUint {
    unimplemented!();
}
pub fn encode_list_to_godel(l: &[BigUint]) -> BigUint {
    unimplemented!();
}
fn encode_program_to_list(program: &[Instruction]) -> Vec<BigUint> {
    unimplemented!();
}
fn decode_pair1(a: &BigUint) -> (BigUint, BigUint) {
    unimplemented!();
}
fn decode_pair2(a: &BigUint) -> (BigUint, BigUint) {
    unimplemented!();
}
pub fn decode_godel_to_list(g: &BigUint) -> Vec<BigUint> {
    unimplemented!();
}
pub fn decode_list_to_program(program: &[BigUint]) -> Vec<Instruction> {
    unimplemented!();
}
```

Note that BigUint does not implement the Copy trait. If we don't want to move the BigUint when passing them as parameters to function calls, then we must borrow the BigUint by passing the function call a reference to the BigUint. BigUints do implement the Clone trait.

Extension B

The code in this state is certainly useful for someone who knows how to use Rust, but what about the non-programmers who have just learned abour Register Machines in their Intro to English Literature course? How are they meant to interact with your code?
Of course this is in jest, but it's an important idea to make your code usable by the general public. Use the clap crate to create an interface with which users may use functionality in your code. Feel free to be creative here! Do you want your users to pass in a text file that has the program? How about a way for the user to just pass in a Godel Number for something, and encode it/ decode it into another representation? Maybe you want to allow the user to step through the execution of some program instruction-by-instruction. Really, go wild!

Extension C

This extension is definitely meant as more of a flex. Towards the end of the COMP50003 course, we were introduced to the idea of a Universal Register Machine. That is a Register Machine which can take the Godel Numbering for any other Register Machine and execute it. The link to the video is here. Try to write such a program that can be executed with your evaluator. See what interesting things you can come up with. What happens if you encode the Universal Register Machine into its Godel Numbering, and pass that into the Universal Register Machine?

Submission

Once you've finished working and you want the piece to be marked, you can