Register Machines and Computable Functions in HOL4

Zhuo Chen

An ASC submitted for the degree of
Bachelor of Philosophy (Science)
The Australian National University

July 2019

– 14 July 2019

Except where otherwise indicated, this thesis is my own original work.

Zhuo Chen
14 July 2019

to my parents

– 14 July 2019

Acknowledgments

Great thanks to my supervisor Michael Norrish for his kind guidance, inspiring ideas and great help throughout my project. Thanks to my friend Yiming Xu for her kind help with using HOL4.

Register Machine in HOL4

An interactive theorem prover(ITP), or proof assistant, is a software tool which assists human-computer collaborated formal proofs. HOL4 is the newest version of HOL, an ITP developed based on standard meta-language for higher-order logic.

A register machine, also known as an abacus machine, is an abstract machine which simulates computation by manipulating the numbers inside an infinite number of registers.

It has been proved that recursive functions, Lambda calculus and Turing machines are able to simulate each other. To convert the proof into a mechanised form, it has been proved in HOL4 that (1). recursive functions and Lambda calculus can simulate each other; (2). recursive functions can simulate Turing machines. The direction from Turing machines to the other two has not yet been proven so we can not establish the equivalence between three of them.

In this paper, we try to further complete the above proofs in HOL4. We developed and verified a register machine model, explored how register machines are able to simulate recursive functions and suggested that if Turing machines is able to simulate register machines can be proven, we will be able to close the whole proof and establish the equivalence relation between recursive functions, Lambda calculus, Turing machines and register machines.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mathematical proving has existed for a long time in human history. However, using computers to help with such proofs is a relatively new field compared to using pen and paper. We call such technique theorem proving. The computer software that is used to do theorem proving is called an interactive theorem prover(ITP), or proof assistant. IPSs usually have libraries of existing theorems and are able to reason about the proof given assumptions, conclusions and the theorems to be used. Some of which allows users to use tactics to apply the knowledge. HOL4 is an ITP developed based on standard meta-language for higher-order logic.

There are well-known proofs showing that recursive functions, Lambda calculus, Turing machines and register machines are able to simulate each other. To convert the proof into a more mechanised form, it has been proven in HOL4 that Lambda calculus and recursive functions can simulate each other. Meanwhile, recursive functions has also been proved to be able to simulate Turing machines. To prove the equivalence between these three models (Lambda calculus, recursive functions and Turing machines), we need to prove that Turing machines can simulate recursive functions or Lambda calculus. However, it is difficult to do so straight away. We need to find another way out.

It has also been proved in theory that register machine and the above three functions/machines are able to simulate each other. This means that alternatively, we can prove that Turing machines can simulate register machines and register machines can simulate recursive functions, then derive the equivalence between these four models.

In this paper, we developed and verified a register machine model in HOL4 and discussed how this could link to solving the above problem.

Firstly, we constructed a general model of the register machine and different register machines representing a different type of computations derived from that model. Secondly, we established and proved the equivalence between these machines and the arithmetic operations they represent. Lastly, we showed how more complicated computable functions can be simulated by register machines, suggested how in theory recursive functions can be simulated by register machines and linked this back to how this helps with transforming the well-known proofs about recursive functions, Lambda calculus, Turing machines and register machines into more mechanised form.

Chapter 2

Background and Related Work

This paper is motivated by the well-known proof about recursive functions, Lambda calculus, Turing machines and recursive machines and the progress of transforming them into more mechanised form in HOL4.

This paper uses HOL4 as the proving platform to transform their proofs into a more mechanised form.

2.1 Motivation

"Any real-world computation can be translated into an equivalent computation involving a Turing machine." - Church Turing

In the 1930s, the equivalence relation between Lambda calculus, recursive functions, Turing machines and register machines has been proved. A lot of books have relavant materials and the one we used while writing this paper is Boolos et al..

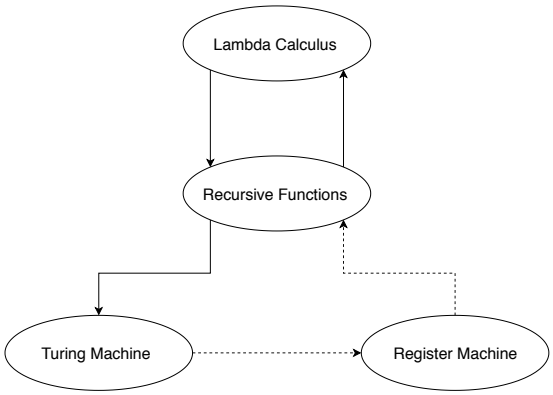


Figure 2.1: Lambda Calculus, Recursive Functions, Turing machines and Register Machines

In HOL4, the equivalence between recursive functions and lambda calculus has been proven by Michael Norrish. Recursive functions can simulate Turing machines has been proven by Elliot Catt. These are shown in figure 2.1 as the solid lines. We

can see that the proof from Turing machine to register machine and from register machine to recursive function is unfinished.

What we really want to show is that Turing machines are able to simulate recursive functions or Lambda calculus. However, it is difficult to complete such proof. Alternatively, we choose a pathway by proving that Turing machine can simulate register machines and register machines is able to simulate recursive functions to achieve the same proving goal. It is easier to do such a proof because (1). Turing machines and register machines are both abstract machines which simulate computation and have similar structures; (2). Register machine is able to simulate recursion by using registers to store counter, accumulator and using states to simulate base case and recursive steps.

2.2 Related work

2.2.1 Interactive Theorem Prover

Interactive Theorem Prover(ITP) is a software tool that does math proof by providing a human-computer collabrative service. ITPs have existing libraries which stores proved theorems and can be easily accessed during proving steps. The previous theorems proved by users locally can also be borrowed for new proofs easily. Most ITP proves given goal by using the tactics and theorems provided by user to guide each proving step and is able to warn the user when the proving step given by the user is not rigorous enough. While pen and paper proofs may sometimes have logic errors which users are not able to notice, ITP reduces the amount of such errors.

2.2.2 Higher Order Logic Interactive Theorem Prover

Higher Order Logic(HOL) is an ITP written on top of Standard Meta Language and HOL4 is the newest version of it. HOL4 allows new types and new constants and proofs of theorems about those types and constants. Working in HOL4 is similar to working in Haskell. HOL4 also uses the same math operators and logic operators such as arithmetic operators and logic quantifiers. Lambda can also be printed in the same way as the greek letter in HOL4 but in this paper we will refer to it as \, a backslash. We will explain some basic concepts in HOL4 to help understand the code snippets provided in the paper.

Tactics Tactics are able to simplify the proofs. Some of also allow users to add theorems to apply during the simplification by giving the tactics the name of the theorems. The tactics we used in this paper are `rw[], simp[], metis_tac[]`, all of them allow users to add theorems. For example, `rw[theorem1]` will add theorem1 to `rw[]`'s simplification.

Record `p1 = <| Person := "Mary"; Fruits:= ["Apple", "Orange"]; Age := 21 |>` defines a record type which can record information of different fields (different

types are allowed) of one variable.

Set Set in HOL4 is same as a mathematical set (all elements are of same type, no duplicates). For example, $\{1, 293, 45\}$ is a set.

List List defines a list of elements of the same type where duplicates are allowed. For example, `[1;2;3]` is a list.

Function Composition `o. f o g x` is the same as `f(g(x))` where x is the input.

2.2.3 Register Machines and Recursive Functions

Register Machines A register machine, also known as an abacus machine, is an abstract machine which simulates computation by manipulating the numbers inside an infinite number of registers. Similar to Turing machines, register machines use states to represent different actions and use transition functions to move from state to state.

Imagine each register as a basket and the number in it as the amount of stones inside the basket. We are only allowed to add or remove one stone at a time. In the other words each action will either increase or decrease the number inside one of the registers by one.

The graph notation we used in this paper is in the same style as figure 2.2 where sx stands for state x, rx stands for register x and rx-/rx+ stands for removing/adding one stone from/to register x. The general form of this notation is borrowed from Boolos et al.

There is no solid development of register machines in HOL4's database before this paper.

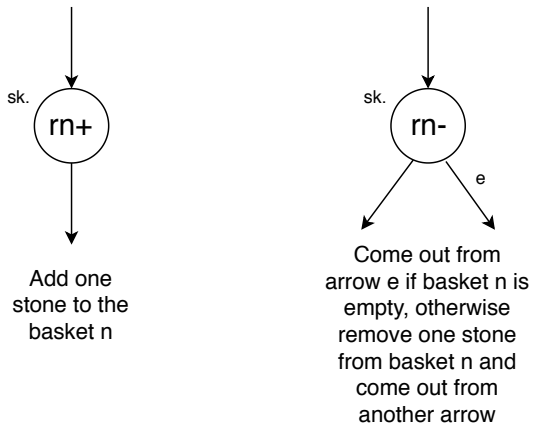


Figure 2.2: Elementary Operations in Register Machines

Recursive functions Recursive functions can be split into two types: primitive recursive functions and recursive functions with no end. Primitive recursive functions have a base case for when the guard is 0 and applies the recursive step repeatedly while minusing the guard by one each time. Each recursive step has to lead closer to the base base. Guard needs to be non-negative. For the non-primitive recursive functions, their recursive step does not lead towards the base case, which means that they will never terminate. Recursive functions in general always perform recursive steps but may or may not terminate depending on whether or not it gets closer to the base case.

Chapter 3

Register Machines in HOL4

Register machines have not been established and included in HOL4 formally. In order to find the connection between register machines and recursive functions, we first need to construct a register machine model and define some concrete register machines. We also proved the correctness of some concrete register machines to show that our construction is correct and that register machines are able to simulate some computable functions.

3.1 Register Machine Model and General Functions

Before creating register machines and using them to perform computations, we first need to define what a register machine is and how to run it. We also defined several helper functions to assist with constructing complicated register machines and proving of register machines.

3.1.1 Definition

Each register machine is a collection of different fields: set of states, the arrows(transition function) betweem states, set of registers and special states/registers. The register machine model need to be able to describe all different register machines, which is why we defined it as below, using record type to record each field.

```
rm = <| Q : state set;
        tf : state -> action ;
        q0 : state ;
        In : reg list ;
       Out : reg |>
```

`Q` is a set of numbers, representing the set of states of the register machine.

`tf` stands for transition functions. It returns the action that happens in the given: increasing the number in a register by one and move on to another state, or decreasing the number in a register by one and move on to another state or goes to `NONE`(exits).

`q0` is the initial state.

`In` is a list of numbers, representing the input registers.
`Out` is a number, representing the output register.

3.1.2 Computation

For a given register machine and inputs, the computation is done by calling the function `RUN` with the two arguements being this register machine and these inputs. `RUN` uses several helper functions to initialise the machine with the given inputs and then starting from `q0`, walk through the states according to the machine's transition function as well as perform the action inside the current state each time.

3.1.3 Helper Functions

We implemented a series of helper functions to help with the proofs of register machines and the construction of more complicated register machines. You might want to skip this section and come back to check these functions when they are mentioned in the later chapters.

correct1 `correct1` checks if the given register machine computes the same result as the given unary function;

correct2 `correct2` checks if the given register machine computes the same result as the given binary function.

dup `dup` makes a duplicate of the given register in the other indicated register.

link `link` links two register machines one after another by connecting the ending state of the first machine to the initial state of the second machine.

link_all `link_all` links a list of machines together using `link`.

mrInst `mrInst` renames the given register machine's registers. It achieve renaming by pairing up the given machine number with the given machine's register numbers.

msInst `msInst` renames the given register machine's states. It achieves renaming by pairing up the given machine number with the given machine's states.

Pi Projection. Returns the n-th element of the list ns, indexing from 0 .

3.2 Simple Register Machines: Construction and Proofs

After defining the register machine model and the computation method, we are able to construct some register machine examples (shown in table 3.1) where each machine represents a different operation. We then proved the correctness of some interesting machines: `simp_add`, `addition`, `multiplication` and `exponential`. To make the notation of the theorems more structured, we used helper function `correct2` from section 3.1.3.

Table 3.1: Simple Register Machines

Machine Name	Input Register	Output Register	Description
`constant`	r0	r1	Returns a register of whose number is same as the given constant(r0)
`identity`	r0	r0	A machine which does not change any value in any register
`empty`	r0	r0	Empties r0
`transfer`	r0	r1	Transfers all stones in r0 to r1
`double`	r0	r1	Returns 2*r0 in r1
`simp_add`	r2, r1	r1	Performs addition operation by transfering the stones
`addition`	r1, r2	r1	Performs addition operation by transfering and restoring the stones
`multiplication`	r0, r1	r2	Performs multiplication operation
`exponential`	r1, r0, r2	r2	Reads in r1, r0 and r2 (r2 is accumulator and has to be 1). Calculates r1 ** r0.
`factorial`	r0	r1	Calculates r0!

The proofs of the register machines are similar because most of them use loops of the similar structures. Let's start from one of the most basic register machines: `simp_add`.

simp_add : As we can see from figure 3.1, `simp_add` is a simplified version of addition machine. It transfers all stones from one register (r2) into another one(r1) to achieve addition. Theoretically it should compute the same result as the arithemtic addition.

Theorem 1. *`correct2 (+) simp_add` (simp_add is equivalent to arithmetic addition).*

Proof. Theorem is same as proving at the end the result register will hold the sum of the two input registers' initial value. Followed by proof by mathematical induction:

Base case: r2 is empty - return number of stones in r1;

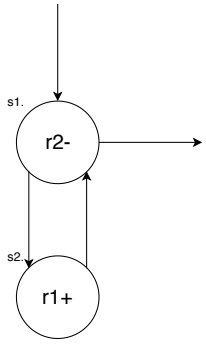


Figure 3.1: simp_add machine

Inductive step: Prove `correct2 (+) simp_add` is true for r2 = SUC k is true if `correct2 (+) simp_add` is true for r2 = k holds.

By looping through the states we end up back as the beginning state with a smaller r2 value (k) and a bigger r1 value(increase by 1)

By induction hypothesis we have for r2 = SUK k, `simp_add` returns (1 + (`simp_add` r2)), which is same as the result returned by (+).

□

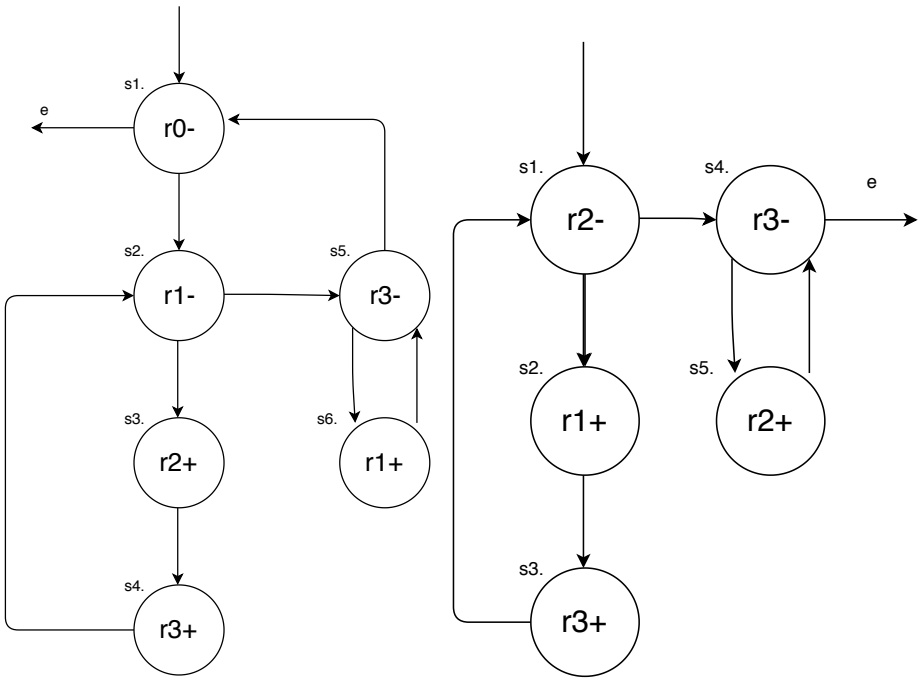
addition : `addition` transfers all stones from one register into another one to achieve addition as well as restore the register we transferred stones from with the same amount of stone at the end. Proof for `addition` is similar to `simp_add`, except after proving the transfer, we also need to prove that the restore of r2 does not affect the number of stones in r1. We can see from figure 3.2 that `addition` has the same structure as the inner loop of `multiplication`. Thus the additional proof is similar to one of the lemmas for theorem about `multiplication`, so we will skip the proof for `addition` and go to `multiplication`'s.

multiplication : Multiplication machine is similar to an addition machine with an extra big loop wrapped outside. Let the two inputs be a and b. `a * b` is represented as `a+a+...+a+a` (there are b number of a's in total) in the multiplication machine. In order to prove multiplication, we need to prove two other lemmas for the subloops inside multiplication.

Lemma 2. *s2 will eventually transit into s5; the loop consists s2 is always doing addition correctly.*

Lemma 3. *s5 will eventually transit into s1; the loop consists s5 is always transfering r3 into r1 correctly.*

Lemma 2 is proved in the same manner as simp_add. Their proof in HOL4 are not exactly the same because of the state number, register number difference as well as the expression difference.



(a) multiplication (b) addition

Figure 3.2: Multiplication Machine and Addition Machine

Lemma 3 is of similar structure as lemma 2 but one less state. The proof method goes the same but modify details to compromise with the state number and register number difference.

Theorem 4. *correct2 $∗ multiplication*

Proof. Induct on r0. If r0 = 0 then return r2, which is 0. 0*r1 = 0 so for the base case the theorem is correct. Assume it is correct for r0 = k. In the case of r0 = k+1, by using lemma 2 and lemma 3 we are back to s1 with r2 added by 1*r1. By using the induction hypothesis we have `multiplication` for (k+1) = `multiplication` for k + r1 = k*r1 + r1 = (k+1) * r1. □

exponential : From the figure 3.3, we can see that `exponential` is a big loop wraps around four small loops. From left to right, the small loops are seperately doing: multiplication(s2-s8), transfering(s9-s10), emptying(s11) and transfering(s12-s13).

Theorem 5. ∀ *a b. RUN exponential [a;b;1] = a ∗∗ b*

The proof of theorem 5 includes the proofs of 4 seperate lemmas for the four subloops, where the multiplication subloop contains another 2 subsubloops. All the proofs are in the same manner but with different details, which are unnecessary and painful. 161 lines were written to prove the small loops inside `exponential` while the main part of `exponential`'s proof is only 20 lines. To help with avoiding

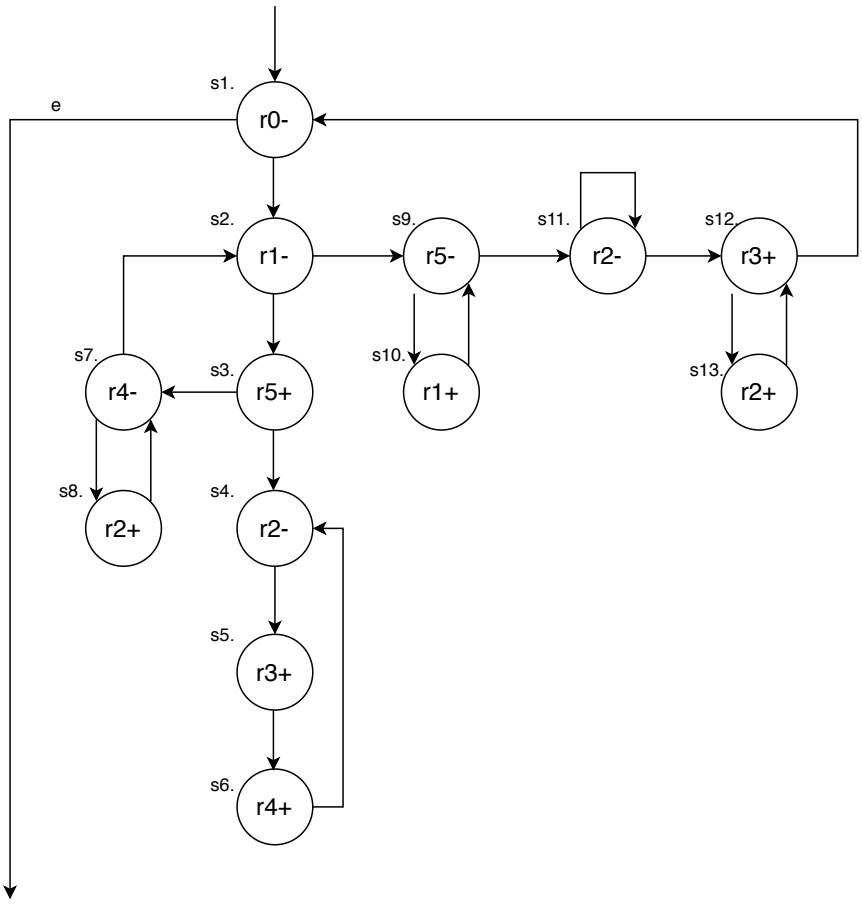


Figure 3.3: exponential machine

unnecessary repeated proofs, we will introduce composition and primitive recursion machines in the next chapter.

3.3 Summary

We have designed the register machine model, designed and proved several examples of simple register machines where each of them stands for a simple operation. We have shown that register machines are able to simulate some simple computable functions such as arithmetic operations. That leads to the question: what is the connection between register machines and computable functions? Can register machines simulate even more computable functions?

Chapter 4

Connection to Computable Functions

In this chapter, we further explore the relation between register machines and computable function. We first show that our register machine model is able to simulate composition, then discuss the idea of how primitive recursion can be simulated and eventually connect this to the idea of how recursive function can be simulated by register machines.

4.1 Put it all together: Composition

In math formulae, we need to use multiple operations in order to get the desired result. For example, $E = mc^2 = m * c * c$, we need to use multiplication twice here (or one multiplication and one squaring operation). Similarly, we want register machines to be able to do the same thing. Further more, we want a new machine which is equivalent to a new formula, rather than writing out different combined machine expressions each time when the input changes, which makes both constructions and proofs of the new machine unnecessarily long. This brings the need of a composition function which composes the given machines together and return a composed machine. We will call it `Cn` from now on. Assume main machine takes `n` inputs, `Cn` will be able to compose this main machine with `n` sub machines. Then we have the general idea of `Cn` as shown below.

```
Cn main_machine sub_machines inputs =
       main_machine (sub_machines inputs)
```

So far this seems simple enough. However, for the actual implementation of `Cn`, there is more to concern other than running the machines one after another.

The biggest problem is that the main machine and the sub machines might have used the same state number or same register number in their definition, which means if we leave the machines to be how they were defined, they might overwrite each other's register or even find multiple transition functions for the same state and stop working. Another thing to concern is that the output registers of the sub machines

and the input registers of the main machine are not the same, which means main machine has no idea what inputs it should take.

The problems are solved by renaming the registers and states and copying the values inside the registers around. The renaming and copying are done by using the helper functions from section 3.1.3 Let's take a closer look at the actual implementation of `Cn`. To help understanding the code, figure 4.1 is how copying and computing work in `Cn`. (renaming is not included in the diagram as it is pretty straight forward by itself)

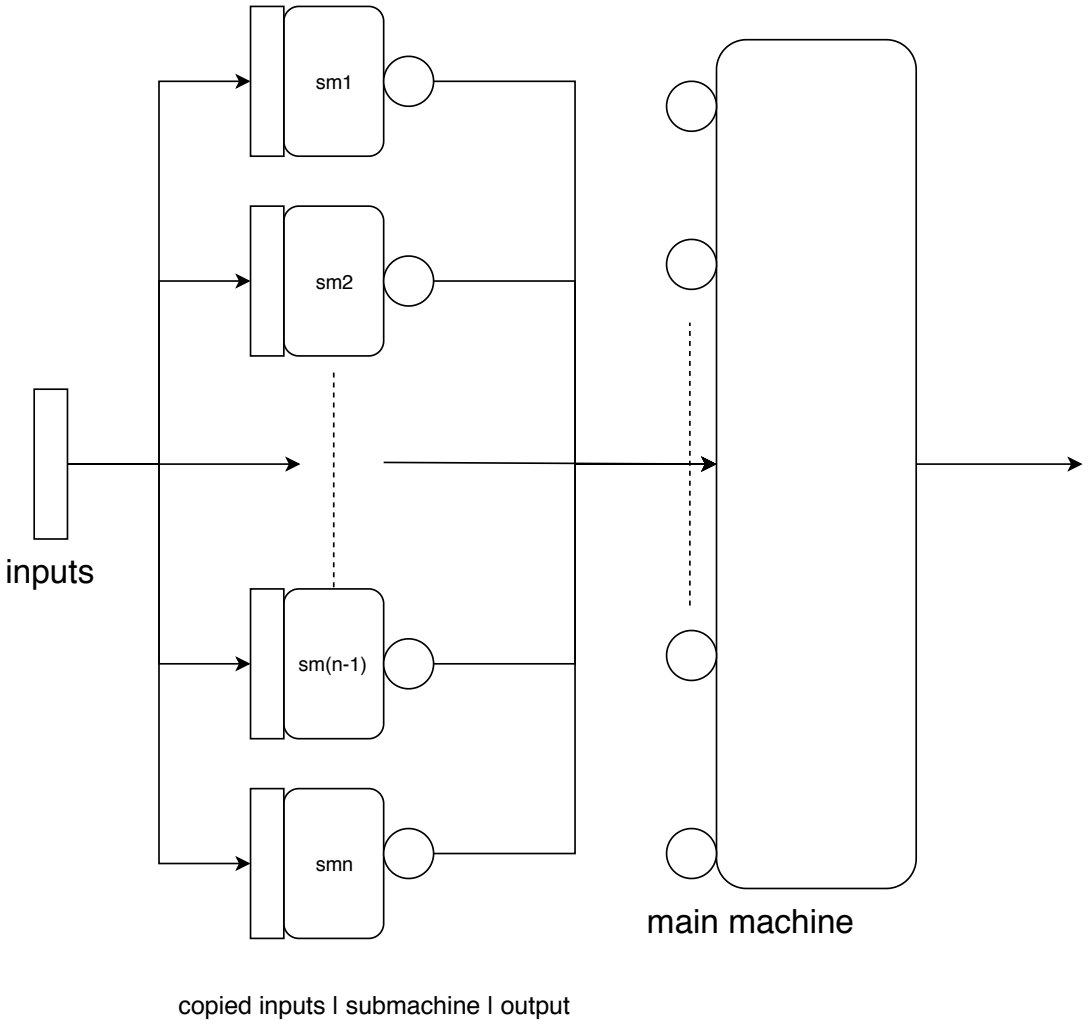


Figure 4.1: Composition

```
Cn m ms =
let isz = LENGTH (HD ms).In;
    mms = MAPi (\i mm. mrInst (i+2) mm) (m::ms);
```

```
    m'  = HD mms;
    ms' = TL mms;
    ics = FLAT (MAP (\mm. MAPi (\i r. dup0 (npair 0 i) r (npair 1 0)) mm.In) m
    ocs = MAPi (\i mm. dup0 mm.Out (EL i m'.In) (npair 1 0)) ms';
    mix = ics++ms'++ocs++[m'];
    mix' = MAPi msInst mix;
in
  link_all mix' with In := MAP (npair 0) (GENLIST I isz)
```

`Cn` renames all the machines' registers to a (machine number, register) pair, saved as `mms` . `Cn` also renames states of all machines (including the copying machines) in the same manner, new machines saved as `mix'`. This ensures that none of the machines will affect each other in an unexpected way. When `Cn` runs, it first copies the same inputs into all renamed submachines' input registers (saved as `ics`) and run the sub machines(`ms'`) seperately. After all submachines finished computing their results, their outputs are then copied into the renamed main machine's input registers(`ocs`). Main machine(`m'`) then computes the final result.

4.2 Primitive Recursion Machines

As we addressed in last chapter, the construction and proving of register machines can sometimes get annoying. It might be repeating the same process over and over again. To make this easier, we need to implement primitive recursion machines.

A lot of register machines constructed in Chapter 3 have primitive recursion inside their construction. `addition` recurses on one of the registers and stops when it is empty. `multiplication` recurses on one register in the outer loop, performs addition in the inner loop and stops when the register in the outer loop is empty. `exponential` recurses on the power in the outer loop and performs multiplication in the inner loops. This means that a lot of machines we explictly constructed in Chapter 3 are actually primitive recursion wrapped around by primitive recursions. So if we have a function which can simulate primitive recursion, alone side with composition, we can describe a lot machines in a simpler way. For example, instead of writing out all the states and register numbers, `addition` can be defined as a primitive recursion which adds one to the output register each time. The more complicated register machines can be simplified in the same way. Further more, by rewriting the register machines in such a way simplifies proofs of them a lot. Currently, we need to prove loops of the same structure every time when it appears in a different machine because they share different state number and register numbers. However, if we manage to rewrite them with primitive recursion and composition, the loops will be the same and we only need to prove one lemma for each loop structure in order to use it in the proofs of all different machines.

The psudo code for a primitive recursion machine would be like below.

```
Pr base step = Recurse counter accumulator guard step

Recurse counter accumulator guard step =
    if (guard = 0) then return(accumulator)
    else Recurse (counter+1) (step counter accumulator) guard step
```

For more intuitive understanding, the general idea of how a primitive recursion machine should be constructed is shown in the graph below. `Pr` has knowledge of counter, accumulator, guard and inputs. `base` expects inputs. `step` expects counter, accumulator and inputs.

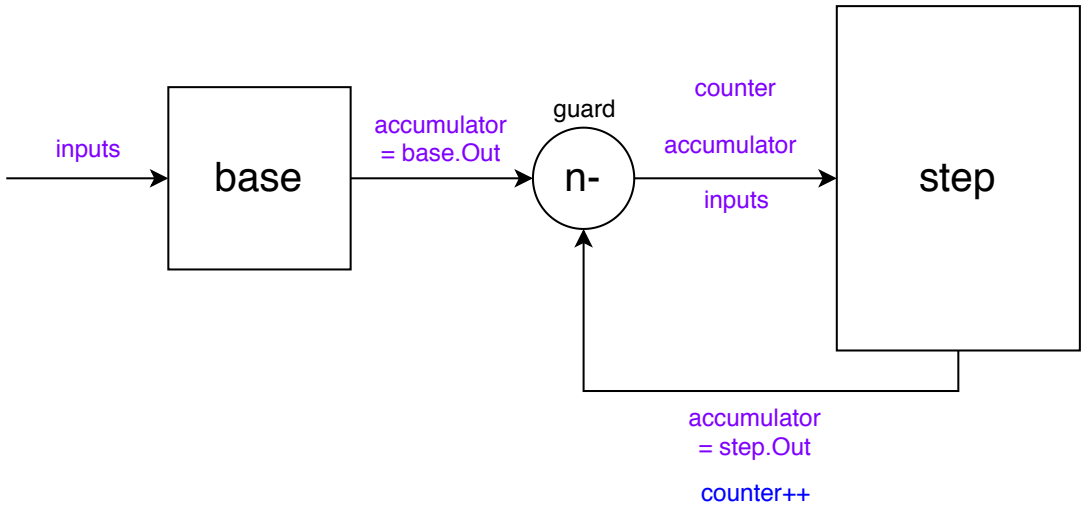


Figure 4.2: Primitive Recursive Machine

The renaming part is not included in the graph for the same reason as for `Cn`. Purple words stand for a copying machine. Blue words stand for a machine that does the described operation. For the computation steps, first `Pr` feeds the inputs into `base`. Then we compute base and update `Pr`'s accumulator by the output from `base`. Recursive steps now starts. `guard` decreases guard register by one and check if it has hit zero. If guard is 0 then return the accumulator, otherwise feed counter, accumulator and inputs into `step` and compute `step`. Add counter by one, update accumulator by `step`'s output and go back to `guard` again. Keep looping until guard returns.

The actual implementation of a primitive recursive machine is not yet finished. The current progress and suggested future work is discussed in Chapter 5.

4.3 Register Machines and Recursive Functions

In order to show that register machines are able to simulate recursive functions, we decompose recursive functions down to primitive recursive functions and minimisation function which reprensents the non-terminating recursive functions. We can show that register machine is able to simulate primitive recursive functions by showing that it is able to simulate 0, SUC, projection, composition and primitive recursion.

`0` Register machine can simulate 0 by calling the constant machine with argument 0.

SUC Register machines are able to simulate SUC by using the `add1` machine.

Projection Projection is represented by the register machine `Pi m n` where the m-th element from the list n will be returned.

Composition Composition is represented by the register machine (`Cn m ms` where m is the main machine and ms is the list of submachines that the main machine will be composed with.

Primitive Recursion Primitive Recursion can be represented by the register machine `Pr base step` where `base` is the register machine for the base case and `step` is the register machine for the step case.

Minimisation Minimisation can be represented by a function `Mu f n` which finds the least n such that f(n) = 0.

We have successfully finished the first four (0, SUC, projection and composition), yet have primitive recursion and minimisation to complete. The current progress and future work for these two will be discussed in the next chapter. After having all the above components, by putting primitive recursive functions and minimisation together we have a complete definition of recursive functions.

Chapter 5

Conclusion

This paper brings the proof for the equivalence between recursive functions, Lambda calculus, Turing machines and register machines closer to completion. This paper covered two separate parts which helps with the greater proof. The first part is the ground work for register machine in HOL4, include defining a register machine model and how to run them, constructing concrete register machines for arithmetic operations, testing and proving their computability. The second part is about composition and recursion, then recursive functions. We constructed more advanced machines to represent 0, SUC, projection and composition. We then presented the method to construct a primitive recursion machine and mentioned how never ending recursions can be represented by minimisation function. We also linkd to recursive functions and explained how they could be simulated by register machines once we finished writing primitive recursion and minimisation machines.

5.1 Future Work

5.1.1 Primitive Recursion Machines

The primitive recursion functions is very crucial, since with it we will be able to construct different primitive recursion machines conveniently. The current version of primitive recursion function is close to finish, however have some run time issues which means that in the actual implementation of the primitive recursion function we will have some errors that need to be fixed.

We have helper functions implemented to help with writing primitive recursion machines. Helper functions include `guard`, `count` and `plink`. `guard` and `count` are designed as seperate machines which can be reused inside each Pr machine as default constants. `guard` checks the guard register and decreases it by one each time. `count` increases the counter by one each time to help with the calculation for the recursive step. Because we start calculation from the innermost operation (base case) to the outermost operation, we are not able to perform calculations with the guard register value as it counts from the most outer operation to the most inner operation. The sum of the counter register and the guard register after each recursive step is always equal to the original guard register input. `plink` is a new linking function. It acts similar to `link`, except that instead of linking all the out arrows from m1 to

m2, `plink` only links one of the out arrows of r1 to r2. This is because the `guard` machine by design returns to `NONE` regardless whether or not the guard register is empty (equals zero). While linking `guard` to `step`, we want `guard` to still return if the guard register is empty and link to `step` otherwise.

The current implementation of primitive recursion function is shown below, which was written in order to achieve the process described in figure 4.2. We used the original `link` function to do all the linking except the link from `guard` to `pts0'` (`pts0'` is `pts0` with renamed states where `pts0` is the copying machine which copies counter, accumulator and inputs into `step`'s input registers).

```
    Pr base step =
    let base' = mrInst 2 base;
        step' = mrInst 3 step;
        ptb   = MAPi (\i r. dup0 (npair 0 (i+3)) r (npair 1 0)) base'.In;
        btp   = dup0 base'.Out (npair 0 1) (npair 1 0) ;
        pts0  = dup0 (npair 0 0) (EL 0 step'.In) (npair 1 0);
        pts1  = dup0 (npair 0 1) (EL 1 step'.In) (npair 1 0);
        pts   = MAPi (\i r. dup0 (npair 0 (i+3)) r (npair 1 0)) (DROP 2 step'.In);
        stp   = dup0 step'.Out (npair 0 1) (npair 1 0);
        pts0'  = msInst 1 pts0;
        guard' = plink guard pts0';
        stp'   = msInst 2 stp;
        mix1   = ptb ++ [base'] ++ [btp];
        mix2   = [pts1] ++ pts ++ [step'] ++ [count];
        mix1'  = MAPi (\i m. msInst (i+3) m) mix1;
        mix2'  = MAPi (\i m. msInst (LENGTH mix1 + i + 3) m) mix2;
        stp''  = link stp' guard' ;
        mix    = mix1'++[guard']++mix2'++[stp''];
  in
    link_all mix with In := MAP (\r. npair 0 (r+2)) (GENLIST I $ LENGTH base.In + 1)
```

While theoretically `Pr` should be able to construct primitive recursion functions, in reality the primitive recursion machine constructed by it is not currently able to compute result. It keeps printing out the new machines it is linking to when it is supposed to fail the guard and drop out of the machine and return the results.

Possible reasons for this to happen could be `plink` is not working as supposed to; error in the renaming process; some machines are linked in the wrong order while making `mix` in `Pr`; not able to do the computation because the Lambda abstraction inside the machine definition makes HOL4 confused; etc.

5.1.2 Register Machine to Recursive Functions

After completing primitive recursive functions, we only need to write minimisation function to finish proving that register machine is able to simulate recursive functions.

As mentioned in Section 4.3, minimisation can be represented by a function `Mu f n` which finds the least n such that f(n) = 0.

5.1.3 More General Formed Composition

The current combination function is only able to compose a main machine with submachines of whose inputs are the same length. This is definitely not practical enough because we would want to use be able to use functions of different dimensions.

To make it possible for submachines to have different dimensions (different number of inputs), we need to have a selection machine. The selection machine is done and mentioned in last chapter as the projection machine `Pi`.

With the existing `Cn` and `Pi`, we just need to integrate `Pi` into `Cn` to make each submachines be able to choose different number of variables from the input list. In this way the more general formed `Cn` will be implemented.

5.1.4 Relation Between Turing Machine and Register Machine

Work on this relation hasn't been started. However, if we can prove that Turing machine is able to simulate register machine after finishing future work suggested in section 5.1.2, we can then show that the equivalence between recursive functions, Lambda calculus, Turing machines and register machines holds as illustrated in the figure 2.1

5.1.5 More Proofs

Whilst many simple register machines are proved to be true in Chapter 3, there is still a lot more left to be proved. For example, the other simple register machines such as empty can also be proved. There are also other more complicated machines that can be proved, such as `factorial`, `exponential` and `Cn`. The proofs for them will be totally different after rewriting them using `Pr` and more general form of `Cn`, but for now, the proving processes are discussed below.

factorial The proof for factorial is close to done. Similar to `exponential`, the construction of `factorial` used similar loops as `multiplication` inside it which makes the proof of `factorial` similar to `multiplication` proof and `exponential` proof. However, there is some difference between how we defined `exponential` and `factorial`. For `exponential`, it requires input numbers to do exponentiation operation on as well as the accumulator which always needs to be one. For `factorial`, we modified it so that instead of an accumulator(which represents the result of the base case), we add extra states in `factorial` to act like a base case and initialise the accumulator. We also initialised counter inside `factorial`. This modificaiton causes problem in the proving process because after the initialising states, when we are inside the recursive steps, the counter and accumulator are still assumed to be zero as they were assumed to be like that before initialising. This could be caused by problem with quantifiers restrictering during the proving process.

exponential As we discussed in the last paragraph, `exponential` requires a redundant input, the accumulator. We can simplify `exponential` and prove the new definition of it.

Cn Composition is constructed however not proved. As future work, we can first prove its helper functions such as `link`, `mrInst` and `msInst`.

Bibliography

Boolos, G. G.; Burgess, J. P.; and Jeffrey, R. C. Computability and logic. (cited on pages 3 and 5)

Catt, E. Recursive functions to turing machine. https://github.com/HOL-Theorem-Prover/HOL/tree/develop/examples/computability/turing. (cited on page 3)