Register Machines and Recursive Functions

Zhuo (Zoey) Chen (u6363182) under the supervision of Dr Michael Norrish
The Australian National University

Abstract

Turing machine, recursive functions, Lambda calculus and their equivalence play a central role in computer science. However, there does not yet exist a mechanised proof of this equivalence in HOL4. HOL4 is a well-known proof assistant built upon simple type theory. Therefore, the formalisation of this equivalence can both support existing proofs and contribute to further research in related topics in HOL4.

Existing theorems in HOL4 have established the equivalence between Lambda calculus and recursive functions. Furthermore, recursive functions are proved to be at least as strong as Turing machines. The final step is to prove that Turing machines are at *least* as strong as recursive functions. In our previous report, we chose the pathway of proving this via register machines and laid the groundwork for proving that register machines can simulate recursive functions.

In this report, we detail the continuation of that project. In particular, we have developed a programming logic for register machines resembling Hoare logic. Using this framework, we prove the correctness for a variety of simple example machines. In addition, we show the correctness of machines that emulate some of the constructions of unary recursive functions.

1 Introduction

The invention of Turing machines, Lambda calculus and recursive functions along with the proofs of their equivalence was a significant breakthrough for logic, math and computer science fields. However, each of these three models come with their own unique set of advantages and disadvantages. Under different circumstances, one can often work better than the others. For instance, Turing machines are good at expressing computer programs but really tedious to manipulate. Meanwhile, the abstract nature of Lambda calculus and recursive functions enables them to solve the same problems in a much more elegant manner.

Being able to use these three models interchangeably is very important for computer science research. Interactive theorem proving is one such area where this applies.

Interactive theorem proving is the formalisation of pen and paper proofs in a computational system, called a theorem prover. Just like we need axioms in mathematical proving, interactive theorem provers (ITP) also need to be built upon some ”axioms”, which are usually some small sets of theories. For instance, HOL4, the ITP we are using for our project, is built upon simple type theory. By trusting simple type theory to be correct, we can then use HOL4 to formalise proofs. In the process of the formalisation of existing proofs, we can sometimes identify the flaws of the proofs, or we can prove them to be correct and trust the proof. It also makes the reuse of old proofs a lot easier. Interactive theorem proving plays an important role in both mathematical, logical proving verifications and formal methods in computer science where computer systems are verified.

In HOL4, we also want to be able to use the three models, Turing machines, Lambda calculus and recursive functions interchangeably. In that way, we can choose the right tool to make certain proofs a lot simpler.

Previously, some work has been done for the proof of equivalence of these three models. What is left to do is to prove that Turing machines can simulate recursive functions. In our previous report, we suggested the pathway of proving Turing machines can simulate register machines and then proving register machines can simulate recursive functions. We laid some groundwork for the proof from register machines to recursive functions and in this project, we continue the unfinished work.

In this project, we first propose a new way of defining correctness of register machines, prove a series of lemmas to help work with it and then prove the correctness of some simple sample machines to show that this new verification technology works. We then propose the idea

of breaking large machines into smaller pieces and construct as well as prove a series of helper machines to help implement the idea. Lastly, we make the high-level strategy decision of proving unary recursive functions instead of n-ary recursive functions. We constructed additional machines and proved some of them to show that register machines allow this change. We then rewrite the n-ary composition function into unary and proved it to be correct.

2 Background

2.1 Motivation

Turing machines, Lambda calculus, recursive functions and their equivalence are crucial parts of computability theory.

Turing machines play an essential part in theoretical computer science as they are one of the simplest computer models and can be used to determine if any given question is solvable by computer or not. Lambda calculus is used and researched in mathematics, linguistics, philosophy and computer science. It can express computations by function abstraction and application thus played an important role in the theory of programming languages. Functional programming languages are implementations of Lambda calculus. Recursive functions are used a lot in both mathematics and computer science as an important tool to represent or solve many problems. One of the most important computer science topic, recursion, is realised by recursive functions.

However, proving that Turing machines can simulate recursive functions directly is difficult because of the nature of Turing machines - a tape with 0s and 1s on it. As a result, many people prove this via another computable model called register machines. The new equivalence relation is shown in Figure 1.

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 1 as the solid lines. We can see that the proof from Turing machines to register machines and from register machine to recursive function is unfinished.

In our previous report, we laid some groundwork for the proof from register machines to recursive functions and in this report we continue the unfinished work.

2.2 ITP

Interactive Theorem Prover(ITP) is a software tool that does math proofs by providing a human-computer collaborative 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 a given goal by using the tactics and theorems provided by the 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 number of such errors.

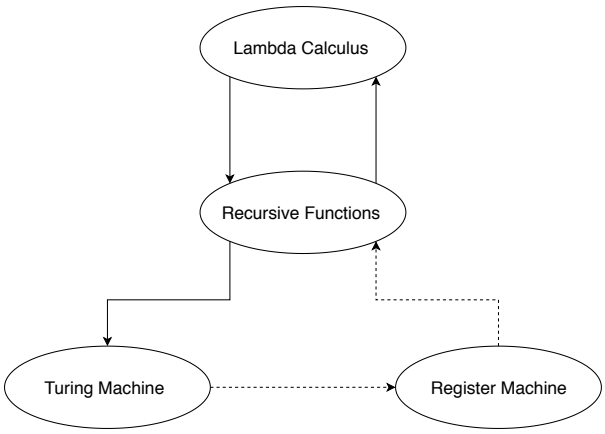

Figure 1: Lambda Calculus, Recursive Functions, Turing machines and Register Machines

2.3 HOL4

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. We will explain some basic concepts in HOL4 to help understand the code snippets provided in the report.

Tactics In HOL4, tactics are tools used to simplify proofs. They can also take additional theorems to apply during the simplification. The tactics we used in this report are *rw* [],*simp* [], *metis_tac* [], all of them allow users to add theorems. For example, *rw* [$theorem_1$] 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.

npair $(\otimes)$ is a function in HOL4 which takes in two natural numbers in order and return a unique value in terms of natural numbers to represent this "tuple".

2.4 Register Machines

Register Machine: A register machine, also known as an abacus machine, is an abstract machine which simulates computation by manipulating the numbers inside an arbitrary 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 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 report is in the same style as figure 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 the textbook Computability and Logic [1]

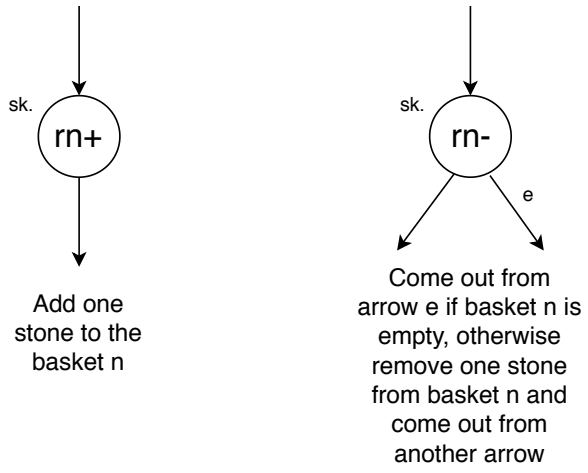

Figure 2: Elementary Operations in Register Machines

2.5 Recursive Functions

Recursive functions are functions that call themselves. Traditionally, to prove the correctness of recursive functions, we need to prove all 6 kinds of recursive functions to be correct: 0, successor, projection, composition, primitive recursive functions and minimisation.

This is what we attempted to do in our previous report. However, such proof is very tedious when it comes to composition, primitive recursive functions and minimisation because we need to keep track of all the inputs of different length. It has been proved that we can also prove recursive functions by proving the following 8 components: 0, successor, pair, first, second, composition, primitive recursive functions and minimisation. Pair, first and second allows the packing and unpacking of the inputs. In this way, the proof of composition, primitive recursion and minimisation has been made a lot easier.

2.6 Hoare Logic

Hoare logic is a formal system with a set of logical rules for reasoning rigorously about the correctness of computer programs. The central feature of Hoare logic is the Hoare triple. A triple describes how the execution of a piece of code changes the state of the computation. A Hoare triple is of the form $\{P\}$ C $\{Q\}$ where P and Q are constraints and C is a command. For any initial state that satisfies P, after executing C, we can get to a final state which satisfies Q.

3 Register machines in HOL4

In our previous report, we developed a formal register machine model in HOL4 and showed that register machines can simulate some computable functions which are discussed in section 3.3. We also defined some recursive functions (0, successor, projection and composition) using register machines.

3.1 Definition

```
rm = <|
  Q : num → bool;
  tf : num → action;
  q0 : num;
  In : num list;
  Out : num
|>
```

Each register machine is a collection of different fields: a set of states, the arrows(transition function) between states, set of registers and special states/registers. The register machine model needs to be able to describe all

different register machines, which is why we defined it as below, using a record type to record each field. We describe the values inside different registers using a function which takes in a register number and returns the value inside that register. We call this function register state.

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

```
action =
  Inc num (num option)
| Dec num (num option) (num option)
```

- tf stands for the transition function. It returns the action that happens at the given state. In the case of an increment action, we will increment the number in the specified register by one and move on to next state. In the case of a decrement action, if the specified register is not empty, then we decrement it by one and move on to state option 1, else we move straight to state option 2.
- q0 is the initial state.
- In is a list of numbers, representing the input registers.
- Out is a number, representing the output register.

3.2 Computation

For a given register machine and inputs, the computation is done by calling the function RUN with the two arguments being this register machine and these inputs. RUN uses several helper functions to initialise the machine with the given inputs and then starting from q_0, walk through the states according to the machine's transition function as well as perform the action inside the current state each time.

However, RUN causes a lot of troubles for proving the correctness of register machines because of the non terminating lambda expantion caused by either the initialisation or the action checking.

3.3 Simple Sample Machines

We constructed and proved some register machines to be correct, such as register machines which represent addition, multiplication, exponential and factorial. A register machine is said to be correct if, given the same inputs, it outputs the same result as the function it represents. In this report, proving a machine and proving a machine to be correct means the same thing,

The graphical model of addition and exponential are shown in Figure 3 and Figure 4. As we can see from the graph, for simple operations like addition, we can still use a relatively small register machine to represent it. However, when it comes to more complicated machines like exponential, it will be really tedious if we write it out from the first principle like shown in the figure.

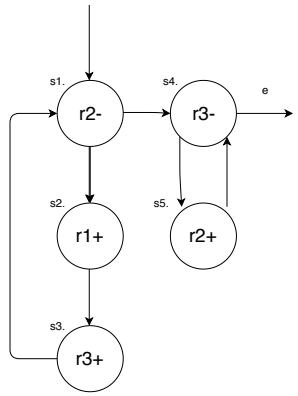

Figure 3: The Addition Machine

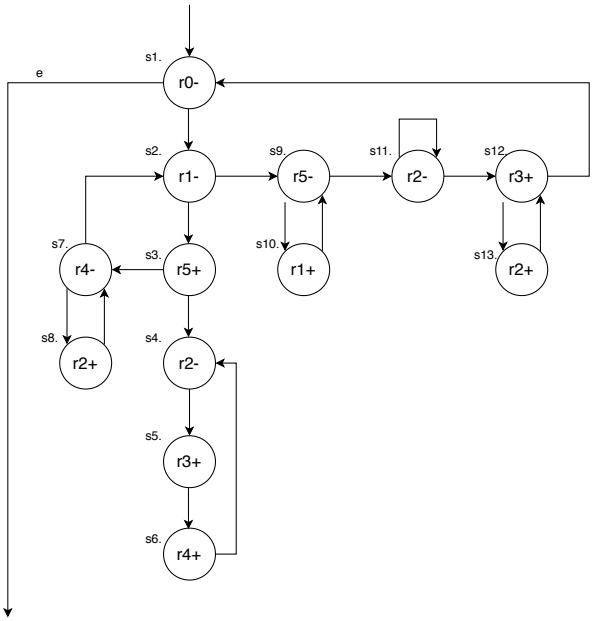

Figure 4: The Exponential Machine

4 Verification Technology

In our previous work, we defined correctness in terms of RUN, which made the proofs very tedious and causes

a lot of nonterminating computations while proving. In order to deal with that, we defined run_step, which works similar to RUN but with everything inside wrapped safely. We then built our primary proving tool, the Hoare triple, using run_step. All lemmas involving the Hoare triple used lemmas of run_step, but we will omit the lemmas for run_step in this report for brevity.

For small register machines, we can use the Hoare triple and its helper lemmas to prove them quite easily. However, as discussed in section 3.3, when we have more complicated computable functions, we will need larger register machines to describe them. To make proving these large machines easier, we developed and proved some glue machines in section 4.2 to help both the construction and proving of large machines.

After that, we defined a new correctness theorem: correct1_rmcorr in section 4.3. The difference between correct1_rmcorr and the correctness theorems in our previous report, correct, is that correct1_rmcorr assumes all the functions and machines are unary and it uses run_step instead of RUN. This involves the strategical change of proving unary recursive functions instead of n-ary recursive functions, which is further explained and implemented in section 6.2.

4.1 Hoare Triples for Register Machines

In this section, we will introduce the Hoare triple, helper lemmas built using it and some examples of simple programs proved using the Hoare triple and these helper lemmas.

4.1.1 Definition

$\{P, q_0\}\, M\, \{Q, qf\}$ (Figure 1) takes five parameters: a register machine M, starting state number q_0, pre condition P, ending state number qf and post condition Q. The Hoare triple returns true if, for each register state which satisfies the precondition P, the machine is able to walk from q_0 to qf and end up with an updated register state which satisfies the postcondition Q in finite steps.

Description 1 *Hoare Triple Definition*

$$\begin{array}{l}\vdash \{P,q\}\, M\, \{Q,qf\} \iff \\ \quad \forall rs. \\ \qquad P\, rs \Rightarrow \\ \qquad \exists n\, rs'.\ \mathsf{run\_step}\, M\, (rs, \mathsf{SOME}\, q)\, n = (rs', qf) \wedge Q\, rs'\end{array}$$

4.1.2 Helper Lemmas

In this section, we will introduce all the helper lemmas built using the Hoare triple. We have four lemmas in total, respectively representing sequential composition, increment correctness preservation, decrement correctness preservation, lemma weakening and loop correctness.

Theorem 1 *Sequential Composition Hoare Triple Proof*

$$\begin{array}{l}\vdash (\forall rs.\ Q\, rs \Rightarrow Q'\, rs) \wedge \{P, q_1\}\, m\, \{Q, \mathsf{SOME}\, q_2\} \wedge \\ \quad \{Q', q_2\}\, m\, \{R, q_3\} \Rightarrow \\ \quad \{P, q_1\}\, m\, \{R, q_3\}\end{array}$$

The Hoare triple allows sequential composition. If we can move from a register state which satisfies P to Q and we can move from any weaker condition Q' than Q to R, then we can move from P to R correctly.

This lemma helps separate huge proofs into smaller parts.

Theorem 2 *Increment Correctness Preservation*

$$\begin{array}{l}\vdash m.tf\, q_0 = \mathsf{Inc}\, r\, (\mathsf{SOME}\, d) \wedge q_0 \in m.Q \wedge \\ \quad \{(\lambda rs.\ P\, rs(\!| r \mapsto rs\, r - 1 |\!) \wedge 0 < rs\, r), d\}\, m\, \{Q, q\} \Rightarrow \\ \quad \{P, q_0\}\, m\, \{Q, q\}\end{array}$$

The action Inc preserves the correctness of the machine as long as the update of register state is performed correctly. If the machine is correct from the future step then it is correct for the current step.

This lemma helps to deal with states with Inc action.

Theorem 3 *Decrement Correctness Preservation*

$$\begin{array}{l}\vdash m.tf\, q_0 = \mathsf{Dec}\, r\, (\mathsf{SOME}\, t)\, (\mathsf{SOME}\, e) \wedge q_0 \in m.Q \wedge \\ \quad \{(\lambda rs.\ P\, rs(\!| r \mapsto rs\, r + 1 |\!)), t\}\, m\, \{Q, q\} \wedge \\ \quad \{(\lambda rs.\ P\, rs \wedge rs\, r = 0), e\}\, m\, \{Q, q\} \Rightarrow \\ \quad \{P, q_0\}\, m\, \{Q, q\}\end{array}$$

The action Dec preserves the correctness of the machine as long as the update of register state is performed correctly. If the machine is correct from both the future steps then it is correct for the current step.

This lemma helps to deal with states with Dec action.

Theorem 4 *Lemma weakening*

$$\vdash (\forall s.\ P\, s \Rightarrow P'\, s) \wedge (\forall s.\ Q'\, s \Rightarrow Q\, s) \wedge \{P', q_0\}\, m\, \{Q', q\} \Rightarrow \{P, q_0\}\, m\, \{Q, q\}$$

Pre-condition strengthening and post-condition weakening preserve the correctness of the program.

This lemma helps proofs of weaker lemmas when we have the stronger version.

Theorem 5 *Loop Correctness*

$$\begin{array}{l}\vdash (\forall N.\\ \quad \{(\lambda\, rs.\\ \quad\quad INV\ rs(\!|gd \mapsto rs\ gd + 1|\!) \wedge\\ \quad\quad rs\ gd = N), body\}\ m\\ \quad \{(\lambda\, rs'.\ INV\ rs' \wedge rs'\ gd \leq N), \mathsf{SOME}\\ \quad\ q\}) \wedge (\forall rs.\ P\ rs \Rightarrow INV\ rs) \wedge\\ (\forall rs.\ INV\ rs \wedge rs\ gd = 0 \Rightarrow Q\ rs) \wedge\\ m.tf\ q = \mathsf{Dec}\ gd\ (\mathsf{SOME}\ body)\ exit \wedge\\ q \in m.Q \Rightarrow\\ \{P, q\}\ m\ \{Q, exit\}\end{array}$$

This is an implementation of a classical definition of loop correctness. A loop is said to be correct if the post-condition P implies the loop invariant, loop invariant stays true throughout the loop, the guard keeps decreasing and when the guard hits zero we end up at the desired register state which satisfies the post-condition Q.

This lemma helps in proving the correctness of loops.

4.1.3 Examples

We then use the lemmas from last section to prove some simple machines such as the register machines which represents addition, multiplication and exponential. The advantage of using the Hoare triple does not quite show off for proofs of addition machines as they have really simple structures to begin with. However, the Hoare triple does make the proof of multiplication and exponential machines a lot easier.

In our previous report, we did the proof for each machine by proving smaller lemmas describing each loop and subloop from first principle. This time, with our lemmas describing sequential decomposition and correctness of loops, we can write all the proofs together in a more condensed way.

We will only show the exponential machine proof here because it's the most complicated one.

Previously, we need to prove 6 lemmas about loops and subloops inside the exponential machine to assist the correctness proof. These 6 lemmas have a similar structure so we will only show one of them in Theorem 6 and omit the rest. For each lemma we need to describe the behaviour of that loop and prove it to be correct. We then finish the final proof(Theorem 7). This made the whole proof very long and all over the place.

Theorem 6 *Exponential Subloop Correctness Proof*

$$\begin{array}{l}\vdash \mathsf{WHILE}\ (\lambda\, (rs, so).\ so \neq \star)\ (\mathsf{run}_1\ \mathsf{exponential})\\ \quad (rs, \mathsf{SOME}\ 4) =\\ \quad \mathsf{WHILE}\ (\lambda\, (rs, so).\ so \neq \star)\ (\mathsf{run}_1\ \mathsf{exponential})\\ \quad (rs(\!|\\ \quad\quad 2 \mapsto 0;\ 3 \mapsto rs\ 3 + rs\ 2;\ 4 \mapsto rs\ 4 + rs\ 2\\ \quad |\!), \mathsf{SOME}\ 7)\end{array}$$

Theorem 7 *Exponential Old Proof*

$$\vdash \mathsf{RUN}\ \mathsf{exponential}\ [a;\ b] = a ** b$$

Now with the new verification technology, we are able to split up the proof using sequential composition (Theorem 1) and deal with the change of register states. Within each subgoal, we can use loop correctness (Theorem 5) and then increment correctness preservation (Theorem 2) and/or decrement correctness preservation (Theorem 3) to proved them. The detailed proof is shown in our code, in this report we will only show the theorem.

Theorem 8 *Exponential Correctness Proof*

$$\begin{array}{l}\vdash RS\ 2 = 0 \wedge RS\ 3 = 0 \wedge RS\ 4 = 0 \wedge RS\ 5 = 0 \Rightarrow\\ \quad \{(\lambda\, rs.\ rs = RS), 14\}\ \mathsf{exponential}\\ \quad \{(\lambda\, rs.\\ \quad\quad rs\ 2 = RS\ 1 ** RS\ 0 \wedge rs\ 0 = 0 \wedge\\ \quad\quad \forall k.\ k \notin \{\ 0;\ 2\ \} \Rightarrow rs\ k = RS\ k), \star\}\end{array}$$

The new proof (Theorem 8) is shorter than the previous version, however, it is still quite long as we still need to walk through every state. In the following section, we will discuss how this could be improved.

4.2 Machine Building Tools

As we can see from the last section, machines built primitively are difficult to write and prove. In our previous report, we came up with the idea of breaking the large machines into smaller pieces and using glue machines (shown in Table 1) to stick them back together to form the large machine. This is very helpful because many register machines share the same small machines and now we only need to prove the same parts once. The decomposition of large machines makes the construction and proof for them a lot easier. In this section, we talk about how we prove all the glue machines.

Theorem 9 *Duplication Hoare Triple Proof*

$$\begin{array}{l}\vdash r_1 \neq r_2 \wedge r_1 \neq r_3 \wedge r_2 \neq r_3 \wedge\\ \quad P =\\ \quad (\lambda\, rs.\\ \quad\quad rs\ r_3 = 0 \wedge rs\ r_1 = N \wedge\\ \quad\quad INV\ (\lambda\, r.\ \mathtt{if}\ r \in \{\ r_1;\ r_2;\ r_3\ \}\ \mathtt{then}\ 0\ \mathtt{else}\ rs\ r)) \wedge\\ \quad Q =\\ \quad (\lambda\, rs.\\ \quad\quad rs\ r_2 = N \wedge rs\ r_1 = N \wedge rs\ r_3 = 0 \wedge\\ \quad\quad INV\ (\lambda\, r.\ \mathtt{if}\ r \in \{\ r_1;\ r_2;\ r_3\ \}\ \mathtt{then}\ 0\ \mathtt{else}\ rs\ r)) \Rightarrow\\ \quad \{P, 0\}\ \mathsf{dup}\ r_1\ r_2\ r_3\ \{Q, \star\}\end{array}$$

Table 1: Glue Machines

Machine Name and Parameters	Description
dup r_1 r_2 r_3	Duplicates the value of r1 into r2 using r3 as scratch register
$m_1 \rightsquigarrow m_2$	Sequencial composition of m1 and m2 (link m2 onto the end of m1)
mrInst *mnum m*	Rename all the registers r used in machine m to $mnum \otimes r$
msInst *mnum m*	Rename all the states s in machine m to $mnum \otimes s$

dup is defined to be correct if given r3 starts with empty inside, it can duplicate the value of r1 into r2 without changing anything. The *INV* function here plays a really important role because it enables us to add conditional constraint on the register state without giving out details about every register in the register state. The actual proof is not complicated.

Theorem 10 *Link Hoare Triple Proof*

$$\begin{array}{l}\vdash \mathsf{wfrm}\ m_1 \wedge \mathsf{wfrm}\ m_2 \wedge \mathsf{DISJOINT}\ m_1.Q\, m_2.Q \wedge \\ \quad q = m_1.q0 \wedge \{P, m_1.q0\}\ m_1\ \{Q, \star\} \wedge \\ \quad \{Q', m_2.q0\}\ m_2\ \{R, opt\} \wedge (\forall rs.\ Q\ rs \Rightarrow Q'\ rs) \Rightarrow \\ \quad \{P, q\}\ m_1 \rightsquigarrow m_2\ \{R, opt\}\end{array}$$

link_correct_V is similar to sequential composition (Theorem 1) because link is a register machine describing sequential compositions. link does not involve any actual states or registers, rather, it talks in terms of m_1 and m_2. This decided that its proof will be a lot different from the previous proofs we have done. For this proof, we need to dive into run_step level and prove theorems about link in terms of run_step in order to prove *link_correct_V* because link actually inspects every state and swap m_1's $\star$ state with m_2's starting state while keeps everything else the same. We first prove that in the first half of $m_1 \rightsquigarrow m_2$, if for m_1 we can start from any state and reach $\star$, then in $m_1 \rightsquigarrow m_2$ we can start from any state and reach m_2's starting state. For the second half of $m_1 \rightsquigarrow m_2$, we prove that if we can start from any state and reach $\star$ in m_2, then we can do the same thing in $m_1 \rightsquigarrow m_2$. We then use them to prove that given m_1, m_2 being wellformed, correct and have no intersecting states, we can (1). walk from $m_1 \rightsquigarrow m_2$'s starting state to $m_1 \rightsquigarrow m_2$'s intermediate state (which is the same as m_2's starting state); (2). we can walk from there to $\star$ and end up with a correctly updated register state. We then put them together and prove *link_correct_V*.

Theorem 11 *Register Renaming Hoare Triple Proof*

$$\begin{array}{l}\vdash \mathsf{wfrm}\ M \wedge q \in M.Q \wedge \\ \quad P' = \\ \quad \mathsf{liftP\_V}\ mnum\ P \\ \quad\ (\lambda rs.\ \forall k.\ \mathsf{nfst}\ k \neq mnum \Rightarrow rs\ k = RS\ k) \wedge \\ \quad Q' = \\ \quad \mathsf{liftP\_V}\ mnum\ Q \\ \quad\ (\lambda rs.\ \forall k.\ \mathsf{nfst}\ k \neq mnum \Rightarrow rs\ k = RS\ k) \Rightarrow \\ \quad \{P, q\}\ M\ \{Q, opt\} \Rightarrow \\ \quad \{P', q\}\ \mathsf{mrInst}\ mnum\ M\ \{Q', opt\}\end{array}$$

We proved 4 different versions of correctness for mrInst and this one is the one we ended up using in the correctness proof of composition.

Generally speaking, mrInst works correctly as long as it changes all the registers of the given register machine to the $(\otimes)$ product of them and the machine number. However, in the proof for composition, we also need the information that the other registers stay the same. Similar to the proof of link, we put an additional constraint on the register states, which is $\lambda rs.\ \forall k.\ \mathsf{nfst}\ k \neq mnum \Rightarrow rs\ k = RS\ k$, all the new registers whose npair decomposition is not part of the (npair mnum x) family holds the same.

Theorem 12 *State Renaming Hoare Triple Proof*

$$\begin{array}{l}\vdash \mathsf{wfrm}\ M \wedge q \in M.Q \Rightarrow \\ \quad \{P, q\}\ M\ \{Q, opt\} \Rightarrow \\ \quad \{P, mnum \otimes q\}\ \mathsf{msInst}\ mnum\ M \\ \quad \{Q, \mathsf{npair\_opt}\ mnum\ opt\}\end{array}$$

msInst is a lot easier to prove because we are only changing the state numbers and they are more or less just some labels. After showing the new machine works the same as the old machine on run_step level, we use it to prove the correctness of msInst.

4.3 correct1_rmcorr

We define correctness in terms of the Hoare triple, shown in definition 2. To prove a register machine M to be correct, we first prove that it satisfies a certain Hoare triple, then use it to prove M to be correct.

We assume all the functions to be unary to make the proof simpler. The reasoning for this will be discussed in more details in section 6.2

Description 2 *Correctness Definition*

$$\begin{array}{l}\vdash \mathsf{correct1\_rmcorr}\, f\, M \iff \\ \quad \exists min. \\ \quad\quad M.In = [min] \wedge \mathsf{wfrm}\, M \wedge \\ \quad\quad \forall inp. \\ \quad\quad\quad \{(\lambda\, rs.\, rs\, min = inp \wedge \forall k.\, k \neq min \Rightarrow rs\, k = 0), M. \\ \quad\quad\quad q0\}\, M \\ \quad\quad\quad \{(\lambda\, rs.\, rs\, M.Out = f\, inp), \star\}\end{array}$$

5 Regster Machines to Recursive Functions

Our ultimate goal is to show that register machines are able to simulate recursive functions and this section contributes to this goal.

As discussed in section 2.5, we can prove the same goal by showing that register machines can simulate unary recursive functions. The decreases the difficulty of proving composition, primitive recursive functions and minimisation a lot lower. All we need to do before the actual proofs are to lay some groundwork to show that register machine can simulate pair, first and second functions.

In this section, we talk about why it is okay to assume all the inputs are unary and how we constructed or proved some recursive functions.

5.1 Unary Functions

By making all functions unary does not make them less effective because the unary inputs still contain all the essential information. For example, we pass $3 \otimes 7$ rather than 3 and 7 separately into a unary addition, and then unary addition decomposes $3 \otimes 7$ down to 3 and 7 by using nfst and nsnd. So we know that making functions unary is fine.

Then we need to show that it also works for register machines. In order to achieve that, we constructed register machines (in Figure 2) working similarly to $(\otimes)$, nfst and nsnd. These machines rely on some smaller machines including Tri and invTri (Table 3) and glue machines (Figure 1). Most of these helper machines have been constructed and proved in previous sections so we will skip them and talk about the two helper machines newly constructed and proved: Tri (Theorem 13) and invTri (Theorem 14).

Theorem 13 *Triangular Function Hoare Triple Proof*

$$\begin{array}{l}\vdash \{(\lambda\, rs.\, rs = RS \wedge \forall k.\, k \in \{2; 3\} \Rightarrow rs\, k = 0), 1\}\ \mathsf{Tri} \\ \quad \{(\lambda\, rs.\, rs\, 2 = \mathsf{tri}\, (RS\, 1)), \star\}\end{array}$$

Theorem 14 *Inverse Triangular Function Hoare Triple Proof*

$$\begin{array}{l}\vdash \{(\lambda\, rs.\, rs = RS \wedge \forall k.\, k \in \{2; 3\} \Rightarrow rs\, k = 0), 1\}\ \mathsf{invTri} \\ \quad \{(\lambda\, rs.\, rs\, 2 = tri\, (RS\, 1)), \star\}\end{array}$$

The proofs for both Tri and invTri are similar to the proofs for simple example machines. We just need to show that give that same input they can give the same result as tri and tri functions in HOL4 respectively.

We then use Tri and invTri to implement Pair, FST and SND. The implementation follows the definition of $(\otimes)$, nfst and nsnd in HOL4.

The proof for Pair, FST and SND are yet to be written but they will be similar to the proof of composition machine, which will be discussed in section 5.3.

5.2 Base Cases: 0 and Successor

We prove that const 0 is equivalent to 0 in Theorem 15. We then prove that add1 is equivalent to the successor function in Theorem 16. The proofs for these two functions are relatively simple because both of them only have one state.

Theorem 15 *0 Correctness Proof*

$$\vdash \mathsf{correct1\_rmcorr}\ (\lambda\, i.\, 0)\ (\mathsf{const}\ 0)$$

Theorem 16 *Successor Correctness Proof*

$$\vdash \mathsf{correct1\_rmcorr}\ \mathsf{SUC}\ \mathsf{add1}$$

5.3 Composition - Cn

5.3.1 Definition

After showing that we can assume that all machines are unary, we then constructed a unary version of the composition machine and proved it to be correct. The unary composition machine (Cn) is shown in definition 3. The graphical representation of it is shown in figure 5

Table 2: Unary Functions Simulated by Register Machines

Machine Name and Parameters	Description
Pair f g	Pair up unary machines f and g such that for any given input n, it will return $fn \otimes gn$. (fn is the output of machine f given n and gn is the output of machine g given n)
FST	For any given input n, compute nfst n
SND	For any given input n, compute nsnd n

Table 3: Helper Machines

Machine Name and Parameters	Description
Tri	Calculates the triangular number at the given index
invTri	Calculates the lowest triangular index of a given number

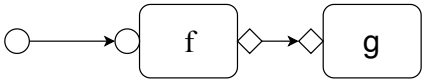

Figure 5: New Composition Graph Model

Description 3 *Composition Definition*

$$
\begin{array}{l}
\vdash \mathsf{Cn}\, f\ g\ = \\
\quad (\texttt{let} \\
\quad\ f' \ = \ \mathsf{mrInst}\ 1\, f\ ; \\
\quad\ g' \ = \ \mathsf{mrInst}\ 2\ g\ ; \\
\quad\ d_1 \ = \ \mathsf{dup}\ g'.\mathit{Out}\ (\mathsf{HD}\, f'.\mathit{In})\ (0 \otimes 1)\ ; \\
\quad\ \mathit{mix} \ = \ [g';\, d_1;\, f']\ ; \\
\quad\ \mathit{mix}' \ = \ \mathsf{MAPi}\ \mathsf{msInst}\ \mathit{mix} \\
\quad \texttt{in} \\
\quad\ \mathsf{link\_all}\ \mathit{mix}'\ \mathit{with}\ \mathit{In}\ := \ g'.\mathit{In})
\end{array}
$$

Comparing to the old version (what we did in the previous report), this new definition makes the structure of composition machines a lot simpler. For the reference, the old version is shown below in definition 4 and in figure 6. This improvement directly decreases the difficulty of proving the correctness of composition machines.

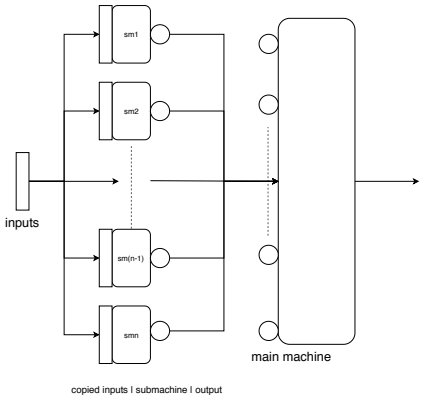

Figure 6: Old Composition Graph Model

Description 4 *Old Composition Definition*

$$
\begin{array}{l}
\vdash \mathsf{Old\_Cn}\ m\ ms\ = \\
\quad (\texttt{let} \\
\quad\ \mathit{isz} \ = \ \mathsf{LENGTH}\ (\mathsf{HD}\ \mathit{ms}).\mathit{In}\ ; \\
\quad\ \mathit{mms} \ = \ \mathsf{MAPi}\ (\lambda\, i\, \mathit{mm}.\ \mathsf{mrInst}\ (i \ + \ 2)\ \mathit{mm})\ (m :: \mathit{ms})\ ; \\
\quad\ m' \ = \ \mathsf{HD}\ \mathit{mms}\ ; \\
\quad\ \mathit{ms}' \ = \ \mathsf{TL}\ \mathit{mms}\ ; \\
\quad\ \mathit{ics} \ = \\
\quad\ \ \mathsf{FLAT} \\
\quad\ \ (\mathsf{MAP} \\
\quad\ \ \ (\lambda\, \mathit{mm}. \\
\quad\ \ \ \ \mathsf{MAPi}\ (\lambda\, i\, r.\ \mathsf{dup}\ (0 \otimes i)\ r\ (1 \otimes 0))\ \mathit{mm}.\mathit{In}) \\
\quad\ \ \ \mathit{ms}')\ ; \\
\quad\ \mathit{ocs} \ = \\
\quad\ \ \mathsf{MAPi}\ (\lambda\, i\, \mathit{mm}.\ \mathsf{dup}\ \mathit{mm}.\mathit{Out}\ (\mathsf{EL}\ i\ m'.\mathit{In})\ (1 \otimes 0))\ \mathit{ms}'\ ; \\
\quad\ \mathit{mix} \ = \ \mathit{ics} \ +\!\!+ \ \mathit{ms}' \ +\!\!+ \ \mathit{ocs} \ +\!\!+ \ [m']\ ; \\
\quad\ \mathit{mix}' \ = \ \mathsf{MAPi}\ \mathsf{msInst}\ \mathit{mix} \\
\quad \texttt{in} \\
\quad\ \mathsf{link\_all}\ \mathit{mix}'\ \mathit{with}\ \mathit{In}\ := \ \mathsf{MAP}\ ((\otimes)\ 0)\ (\mathsf{GENLIST}\ \mathsf{I}\ \mathit{isz}))
\end{array}
$$

5.3.2 Proof

We prove the correctness of Cn by first proving it's true in terms of Hoare triple. We then use it and the lemmas for glue machines to prove the correctness of composition machines.

Theorem 17 *Composition Hoare Triple Correct*

$$\begin{array}{l}\vdash \mathsf{wfrm}\, g \wedge \mathsf{wfrm}\, f \wedge g.\mathit{In} = [\mathit{gin}] \wedge \\ \quad f.\mathit{In} = [\mathit{fin}] \wedge \\ \quad \{(\lambda\, rs.\ rs\, \mathit{gin} = M \wedge \forall k.\, k \neq \mathit{gin} \Rightarrow rs\, k = 0), g.\\ \quad \mathit{q0}\}\ g \\ \quad \{(\lambda\, rs.\ rs\, g.\mathit{Out} = N), \star\} \wedge \\ \quad \{(\lambda\, rs.\ rs\, \mathit{fin} = N \wedge \forall k.\, k \neq \mathit{fin} \Rightarrow rs\, k = 0), f.\\ \quad \mathit{q0}\}\ f \\ \quad \{(\lambda\, rs.\ rs\, f.\mathit{Out} = \mathit{Op}), \star\} \Rightarrow \\ \quad \{(\lambda\, rs. \\ \qquad rs\, (2 \otimes \mathit{gin}) = M \wedge \\ \qquad \forall k.\, k \neq 2 \otimes \mathit{gin} \Rightarrow rs\, k = 0), (\mathsf{Cn}\, f\ g).\mathit{q0}\}\ \mathsf{Cn} \\ \quad f\ g \\ \quad \{(\lambda\, rs.\ rs\, (\mathsf{Cn}\, f\ g).\mathit{Out} = \mathit{Op}), \star\}\end{array}$$

As we can see, the construction of Cn includes the use of a series of smaller machines: f, g and glue machines. The running order of the machines inside Cn is first g, then dup and last f. In previous sections, we have proved that all the glue machines are correct; f and g are assumed to be Hoare triple correct. This means we have the assumption that all of the composition machines' components are Hoare triple correct.

We use sequential composition (Theorem 10) to separate the proof into smaller parts, in which only one machine operates at one time. In each part, we then unpack the renamed machine using the lemmas about renaming machines (Theorem 11) and Theorem 12). After that, we can then use the fact that f, g and dup are correct to finish the proof of these parts.

The tricky part of the overall proof is that we need to be able to show that no machines touch other machines' registers (except for dup, it should not touch any register other than the three arguments we pass in). For f we don't care if it changes anything else because we only need the output to be correct. For g we can easily prove it by using renaming lemma (Theorem 11) because after renaming, it is impossible for g to mess with registers starting with a different *mnum*. For dup, however, we are not able to do the same thing as what we did for g because dup's registers are not renamed. That's where the lemma about dup (Theorem 9) comes in handy. It makes sure that the relevant registers for the future calculation are untouched by setting the *INV* constraint in the lemma to say that all registers are 0, except the input register of f (where we are coping the output value of g into) and the register of g. We then have the proof for composition Hoare triple correct.

After we get this, we just need to show that the composition of two well-formed machines will still be well-formed and then finally prove composition to be a correct unary machine.

Theorem 18 *Composition Correctness Proof*

$$\vdash \mathsf{correct1\_rmcorr}\, f\, M_1 \wedge \mathsf{correct1\_rmcorr}\, g\, M_2 \Rightarrow \mathsf{correct1\_rmcorr}\, (f \circ g)\, (\mathsf{Cn}\, M_1\, M_2)$$

The correctness proof is done by using the Hoare triple proof and the definition of correctness directly.

5.4 Primitive Recursive Functions

In our previous report, we discussed how n-ary primitive recursive functions can be represented by register machines. We realised its construction in this project, which is named *Pr_def* in our code. However, after the strategical change of swapping to unary functions, this construction is not useful to us anymore. Thus we will omit the explanation for n-ary primitive recursive functions in this report.

The pseudo-code for unary primitive recursive functions are similar to the one for n-ary primitive recursive functions except that all the inputs now are unary, which are nested pairs using the pairing functions we discussed in section 6.2.

```
acc = accumulator;

Recurse step (guard, input) acc counter =
  if (guard = 0) then
    return(acc);
  else do
    acc <- step (acc, (counter, input));
    guard--;
    counter++;
    return (Recurse step (guard, input) acc counter);

Pr base step (guard, input) =
  do
    acc <- base input;
    counter <- 0;
    return (Recurse step (guard, input) acc counter);
```

The actual code of the unary primitive recursive function is still under development, we have built a skeleton but need to define the helper functions guard and counter. We also need to add packing. The design of the unary primitive recursive function is shown in the Figure 7. This figure looks more complicated than the one for n-ary primitive recursive functions (Figure 8) but the actual implementation will actually be easier. The unary graph already contains everything that we will need to implement, while the n-ary graph omitted most of the copying processes.

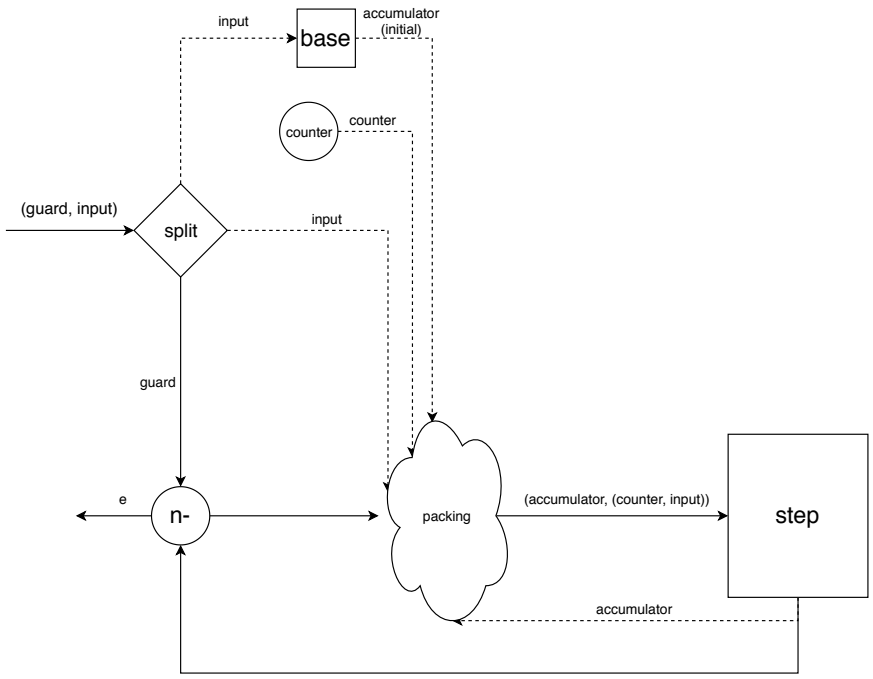

Figure 7: Unary Primitive Recursive Function

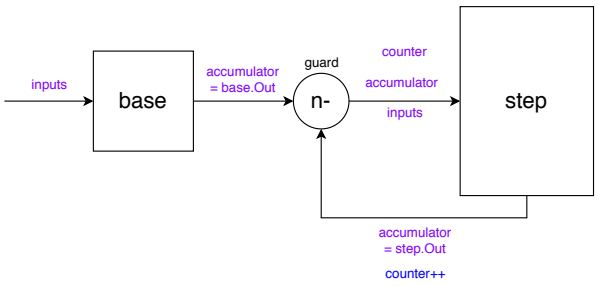

Figure 8: N-ary Primitive Recursive Function

5.5 Minimisation

Minimisation can be represented by a function Mu f n which finds the least n such that f(n) = 0. The minimisation machine is yet to be constructed and proved.

6 Conclusion and Future Work

6.1 Conclusion

We have accomplished different things surrounding the main goal of proving that register machines can simulate recursive functions.

We have developed new tools and definitions for register machine correctness proof in this project, which helps the proofs in this project as well as provides a good proving environment for further research on the same topic.

We have tested our new tools by proving some simple sample machines from our previous report, which shows the reliability of the new tools and further confirm that these simple sample machines are constructed correctly.

We have made some progress on n-ary recursive functions, which can provide a direction for future research if anyone wants to prove them.

We have made the strategical decision of proving unary recursive functions instead of n-ary recursive functions, constructed most of the recursive functions, proved 3 of them and presented the ideas about the construction of unary primitive recursive function and minimisation.

We only have a few things left to do to finish proving that register machines can simulate recursive functions.

6.2 Future Work

Unary Function We need to prove Pair, FST and SND. The proof for them will be similar to the proof for Cn. They are just large machines made of smaller machines: simp_add, simp_sub, Tri, invTri and glue machines. The only difference is that they have way more steps than Cn. All we need to do will be separate the proof into smaller parts by the transitivity law (sequential composition) and then unwrap the renaming machines, then use the existing proofs of smaller machines to prove them.

Unary Primitive Recursive Function The current skeleton we have is shown below:

Description 5 *Unary Primitive Recursive Function Definition*

$$
\begin{array}{l}
\vdash \mathsf{Pr\_unary}\ \mathit{base}\ \mathit{step}\ \mathit{guard}\ \mathit{countt}\ = \\
\quad (\mathtt{let} \\
\quad\ \mathit{base}' = \mathsf{mrInst}\ 1\ \mathit{base}\ ; \\
\quad\ \mathit{guard}' = \mathsf{mrInst}\ 2\ \mathit{guard}\ ; \\
\quad\ \mathit{step}' = \mathsf{mrInst}\ 3\ \mathit{step}\ ; \\
\quad\ \mathit{countt}' = \mathsf{mrInst}\ 4\ \mathit{countt}\ ; \\
\quad\ \mathit{d0\_}_1 = \mathsf{dup}\ 0\ (\mathsf{HD}\ \mathit{base}'.\mathit{In})\ 2\ ; \\
\quad\ \mathit{d1\_}_2 = \mathsf{dup}\ \mathit{base}'.\mathit{Out}\ (\mathsf{HD}\ \mathit{guard}'.\mathit{In})\ 2\ ; \\
\quad\ \mathit{d2\_}_3 = \mathsf{dup}\ \mathit{guard}'.\mathit{Out}\ (\mathsf{HD}\ \mathit{step}'.\mathit{In})\ 2\ ; \\
\quad\ \mathit{d3\_}_4 = \mathsf{dup}\ \mathit{step}'.\mathit{Out}\ (\mathsf{HD}\ \mathit{countt}'.\mathit{In})\ 2\ ; \\
\quad\ \mathit{d4\_}_2 = \mathsf{dup}\ \mathit{countt}'.\mathit{Out}\ (\mathsf{HD}\ \mathit{guard}'.\mathit{In})\ 2\ ; \\
\quad\ \mathit{d4\_}_2' = \mathsf{swap\_NONE\_state}\ \mathit{d4\_}_2\ \mathit{guard}'.\mathit{q0}\ ; \\
\quad\ \mathit{mix} = \\
\quad\quad [\mathit{d0\_}_1;\ \mathit{base}';\ \mathit{d1\_}_2;\ \mathit{guard}';\ \mathit{d2\_}_3;\ \mathit{step}';\ \mathit{d3\_}_4; \\
\quad\quad\ \mathit{countt}';\ \mathit{d4\_}_2'] \ ; \\
\quad\ \mathit{mix}' = \mathsf{MAPi\ msInst}\ \mathit{mix} \\
\quad \mathtt{in} \\
\quad\ \mathsf{link\_all}\ \mathit{mix}\ \mathit{with}\ \mathit{In} := [0])
\end{array}
$$

We need to finish is to construct *guard* and count that are able to deal with nested input so Pr_unary can use them instead of taking two extra parameters which work as guard and counter.

We also need to modify Pr_unary slightly and add *packing* in it (same as in figure 7) because the current code is not correctly implementing the model in figure 7.

Minimisation We have an unfinished n-ary version of minimisation machine. However, we now don't need to finish it anymore as we should develop a unary version of it instead.

Rewrite Sample Machines After we constructed and proved primitive recursive functions in terms of register machines, we can write computable functions using them. For instance, we can write a multiplication machine using Pr and addition by recursing on one of the inputs; we can also write exponential using Pr and multiplication in a similar manner.

References

[1] BOOLOS, G. G., BURGESS, J. P., AND JEFFREY, R. C. *Computability and Logic*. Cambridge University Press, 1974.